

Internal vs External Events

Internal vs External events? Integration Events? Public language? What does it all mean?

Claim check pattern

Store data into DB, then enrich downstream using key.

Journey to event-driven architecture

Store data into DB, then enrich downstream using key.

EventStorming

Workshop to identify events, domains, commands and aggregates

Content enricher pattern

Enrich data before it reaches downstream consumers

Document your event-driven architecture

Notes, thoughts and examples of how to document your event-driven architectures.

At-least-once delivery

Sync vs Async Communication

Pros and cons for a producer and consumer

Choreography vs orchestration

EDA VISUALS

Small bite sized visuals about event-driven architectures

Event first thinking

Identifying events and designing them is important.

Good and hard parts of EDA

Event sourcing and change data capture to for applications.

Event Types

Different types of events used in EDA

Message Queues vs Event Brokers

Understanding queues and pub/sub

Ubiquitous Language

Define a shared language across the business

Commands vs Events

What are commands? What are events?

Derived data with Events

Message driven bounded context

Handled by reactive engines

Event-driven architecture notes by David Boyne

EDA Visuals: Small bite sized visuals about event-driven architectures

David Boyne

v0.0.24

About EDA Visuals

I started EDA visuals as way to share my thoughts, notes and designs with others online. I'm currently using Zettelkasten note taking methods to find information and connections between research. Everything I learn is shared in public, and this document shares some of these thoughts and notes with some visuals to help (myself) and hopefully you too.

This document is organic, every time new visuals are uploaded online this document will be generated and uploaded. (so make sure you keep coming back to check for updated versions).

I truly believe event-driven architectures can transform organisations and I hope you find these notes and visuals useful.

David Boyne

A handwritten signature in black ink, appearing to read 'Boyne', with a large, sweeping initial 'B'.

About David Boyne

My name David Boyne and I'm a Developer Advocate at AWS focusing on event-driven architectures and serverless technology. I dive deep into event-driven architectures and create content online to help others. I have also created many open source projects to help you manage event-driven-architectures (e.g. <https://eventcatalog.dev>).

If you want to learn more or keep up to date with updates feel free to connect with me.

- Twitter: [@boyney123](#)
- Mastodon: @boyney123@hachyderm.io
- LinkedIn: <https://www.linkedin.com/in/david-boyne/>
- GitHub: <https://github.com/boyney123>
- Website: <https://www.boyney.io/>
- My Open source projects: <https://www.boyney.io/projects>

Speaking at Events

If you are interested in having me speak at your upcoming events, I would be happy to discuss the possibility. Whether it's a conference, podcast, seminar, workshop, please feel free to contact me.

Thanks for downloading this content, I hope it can help.

Table of Visuals

Agility with event-driven architecture	7
Batch processing vs event streaming	9
Bounded context with event architectures	11
Choreography vs orchestration	13
Claim check pattern	14
Commands vs Events	15
Common issues when scaling event-driven architectures	16
Content enricher pattern	19
Document your event-driven architecture	21
Event Types	23
Event first thinking	25
Event-driven architecture and Conway's law	27
Event-driven architecture coupled with Domain-driven design	29
EventStorming	31
Events as Data	33
Explicit vs Implicit Events	34
Fundamentals of event-driven architecture	36
Good and hard parts of EDA	39
Inside event-driven architectures	41
Internal vs External Events	43
Journey to event-driven architecture	45
Message Delivery	46
Message Queues vs Event Brokers	48
Message translator pattern	49

Messages between bounded context	51
Point-to-point messaging	52
Producer and consumer responsibilities	54
Publishing events, without any consumers...	55
Schema Management	57
Splitter Pattern	58
Sync vs Async Communication	60
Things to consider when building EDA architectures	62
Ubiquitous Language	64
Understanding Eventual Consistency	65
Understanding Idempotency	67
Understanding change data capture	69
Understanding event delivery failures	71
Understanding event streaming	73
Understanding publish & subscribe messaging	75
Unlocking value from your events	77
Using events to migrate from legacy architectures	79
Why use message brokers?	81
Learn event-driven architecture today	83
Summary	85

Agility with event-driven architecture

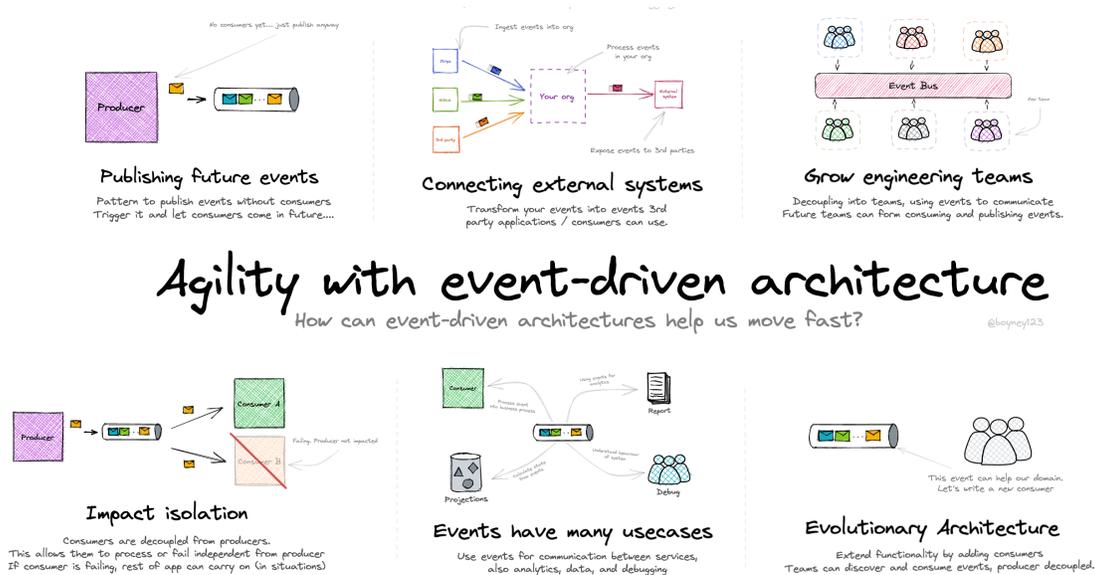


Figure 1: Agility with event-driven architecture

Event-driven architectures provide agility to engineering teams and organisations. Let's dive into more details about how and why.

Evolutionary Architecture

- Common saying with event-driven architectures, is that **producers do not know about consumers**, this allows consumers to be disconnected to producers.
- Without the coupling of producer and consumer, consumers can come and go easily.
- As business requirements change, we can add or change consumers without impacting producers or other parts of our architecture.
- Discoverability is important for consumers to know what they can subscribe to, documentation can help here.

Grow engineering teams

- Using events to communicate between teams/domains is a popular way for engineering teams to scale.
- Event-driven architectures allow us to remain decoupled, and evolve and adapt teams to meet business requirements/features.

- Event-driven architectures couple with domain driven design very well. Define boundaries in your organisation and create teams around that. Understand how Conway's law can impact this.

Connecting external systems

- Event-driven architectures are not just about the events we raise internally within our business or organisation, but events we can also consume and share with others.
- Modern applications integrate with external systems all the time, consume events from other applications and process them internally (webhooks common pattern)
- Expose events to other consumers outside your domain/business, can also provide value.
- Bi-directional events are powerful, reacting to real-time events, processing them and passing them back to the business that produced them.

Impact isolation

- Producers are often disconnected from the consumers, if consumers fail then impact is isolated.
- Event-driven architectures can give availability as components/services are isolated and use events to communicate.
- When consumers fail you have options to deal with this. Having idempotent consumers help, without them you may run into strange side effects.

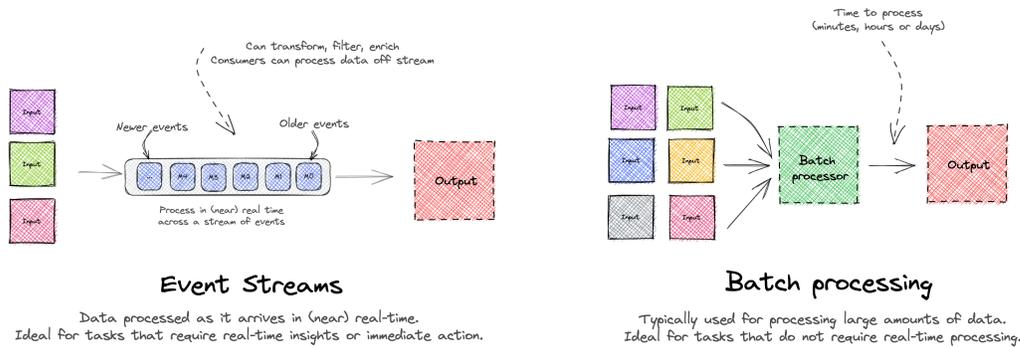
Publishing future events

- Some producers may publish events into the architecture even if no downstream consumers are listening to them.
- If you know your event is valuable to the organisation, you may want to publish it and have the option to listen in the future.
- If you use this pattern, maybe consider the costs in doing this, does your broker charge for events published?

Extra Resources

- Flow Architectures - Great book by James Urquhart about how events can and will be used to communicate between organisations.
- Bidirectional events example - Connecting organizations and events, here is an example of how Amazon EventBridge explore bidirectional events with Salesforce. Some interesting patterns here.
- Dive more into domain-driven design with EDA - Event-driven architectures, team topologies, domains? What does it all mean? Here is a visual to help you.
- How does pub/sub work in EDA applications - Want to know more about pub/sub patterns, producing events for downstream consumers with a fan out approach. Resource here to help.
- Conway's law and EDA - Our system is a reflection of the teams that build it. What does that mean for our EDA solutions? This visual can help.

Batch processing vs event streaming



Batch processing vs event streaming

Exploring different ways to process data in your architecture

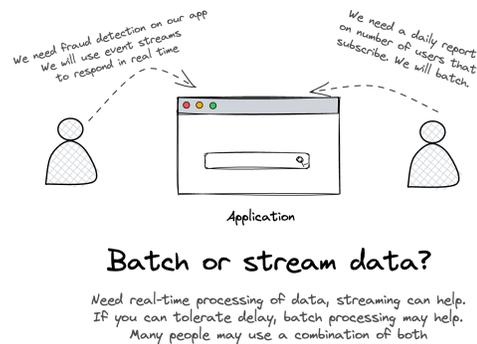


Figure 2: Batch processing vs event streaming

As data becomes an increasingly important part of modern businesses, organizations often find themselves needing to process large amounts of data. Two common approaches to processing data are **batch processing** and **event streams**.

Batch processing

- Batch processing involves processing data at once usually during a scheduled time interval such as daily or weekly.
- Commonly used for tasks that do not require real-time processing and tasks that can tolerate some delay.
- Great if you have a fixed set of input data you want to process.

- Example of batching would be AWS Batch or Apache Spark

Event streaming

- Event streams involve processing data as it happens. Event streams are a continuous flow of data that can be collected and processed in (near) real time.
- Commonly used for applications that require real-time insights. Example of this could be fraud detection, or real-time recommendations.
- Processing data in real time gives organisations information they can act on and analyse straight away, done right this could provide a competitive edge to your business.
- Events can be unbounded data (continuous data, or never ending), so if you need to process this kind of information in real-time then streaming can help.
- Example of streaming would be Amazon Kinesis or Apache Kafka

When to use batching or event streaming?

- Like always, depends on your use case. Can your data processing tolerate some delay? Then maybe batching might be a better fit. Do you need to process the data right now? Will you have an advantage to processing information straight away? Then streaming events might make more sense.
- Remember to consider the costs between batching options and streaming options.
- Use patterns to help your downstream consumers. Remember with streaming and batching you can apply integrations patterns during that particular phase. Example would be enriching events/messages as they go through your stream for downstream consumers.

Extra Resources

- Designing data-intensive applications - Book has some great insights into batching and streaming. Recommend reading.
- Understanding event streaming - Dive deeper into event streaming with this visual.

Bounded context with event architectures

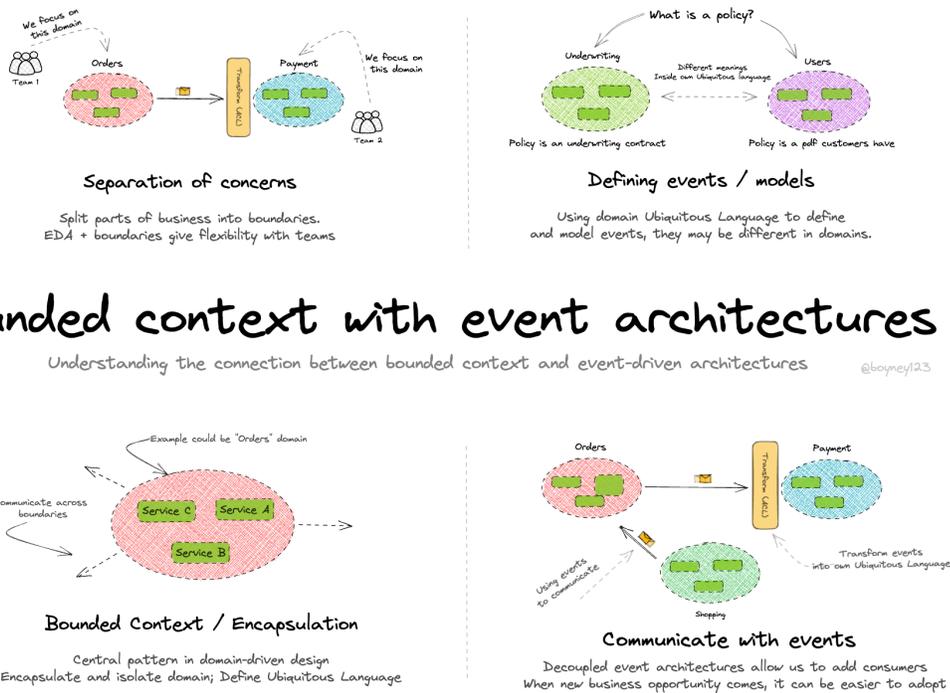


Figure 3: Bounded context with event architectures

Bounded Context

- A central pattern in domain-driven design
- A contextual boundary as part of your domain.
- Has it's own set of concepts and it's own Ubiquitous Language.
- Provides a way to isolate and encapsulate parts of your system
- Great pattern to define boundaries for event-driven architectures
- Common patterns are to use events to communicate between boundaries.

Defining events / models

- Each boundary has its own language, use this language to define events and schemas within your boundary.
- You may have internal and external events within a boundary. External events may be used to communicate with other boundaries.
- Models may differ from boundaries, it's important to understand different context mapping patterns you can use to transform events/messages before consuming them.

- **Consuming an event directly into your boundary** could couple you to the model of another boundary, think about that before you do it.

Communicate with events

- Event architectures allow us to define producers and consumers, these producers and consumers can live across boundaries in your system, and event-driven architectures give us a great ability to communicate between these systems.
- If you follow domain-driven design practices with clear boundaries, you will have decoupled areas of your system, event architectures are a great way to keep decoupled, these go hand in hand.

Separation of concerns

- A big benefit of event-driven architectures is they allow us to be decoupled. This means changes to consumers have limited impact on other consumers/producers.
- If you have clear boundaries in your system and use event-driven architectures to help you can start to form a clear separation of concerns for your organisation and tech teams, allowing to scale solutions and teams faster.

Extra Resources

- Good and hard parts of EDA - Understand the good and hard parts of EDA, defining clear boundaries is just part of the whole picture, it's important to explore more.
- What is Ubiquitous Language? - Visual here to help you understand what Ubiquitous Language is, and why it's important.
- Transforming data between boundaries - When you consume events/messages between boundaries you may not want to take it as it is (raw), do you want to translate the data into your own domain? The answer, probably yes.
- Internal and external events - Firing events within your boundary, firing events outside? Integration? What does it all mean?

Choreography vs orchestration

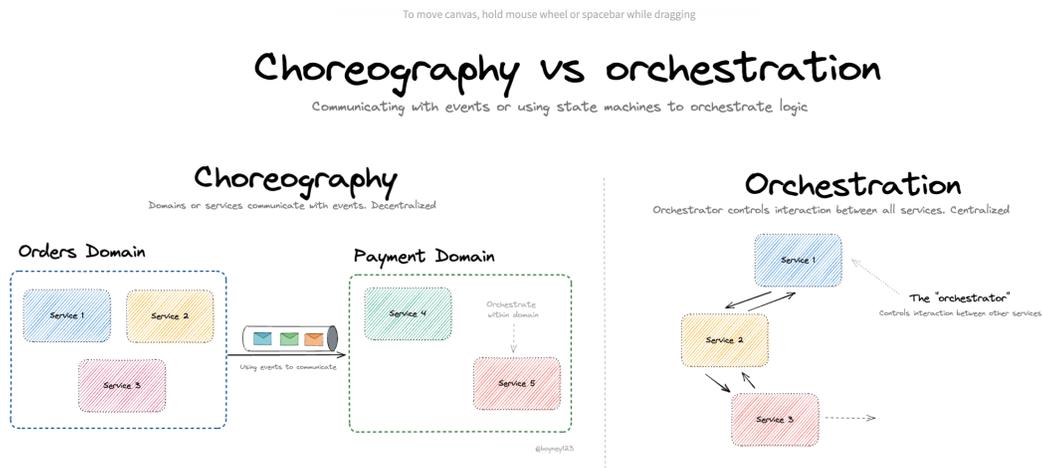


Figure 4: Choreography vs orchestration

Choreography Using events to talk between services (also bounded contexts). Services interact independently. Event-driven approach.

Example of services that can help with choreography are Amazon SQS, Amazon SNS or Amazon EventBridge.

Orchestration Flow of state that is normally controlled by the orchestrator. Think of flow chart or state machines. Might consider using orchestration within a bounded context. Normally follows request/response.

Example would be AWS Step Functions.

Extra Resources

- Choreography vs Orchestration in the land of Serverless - Great blog on choreography and orchestration through a serverless lense.

Claim check pattern

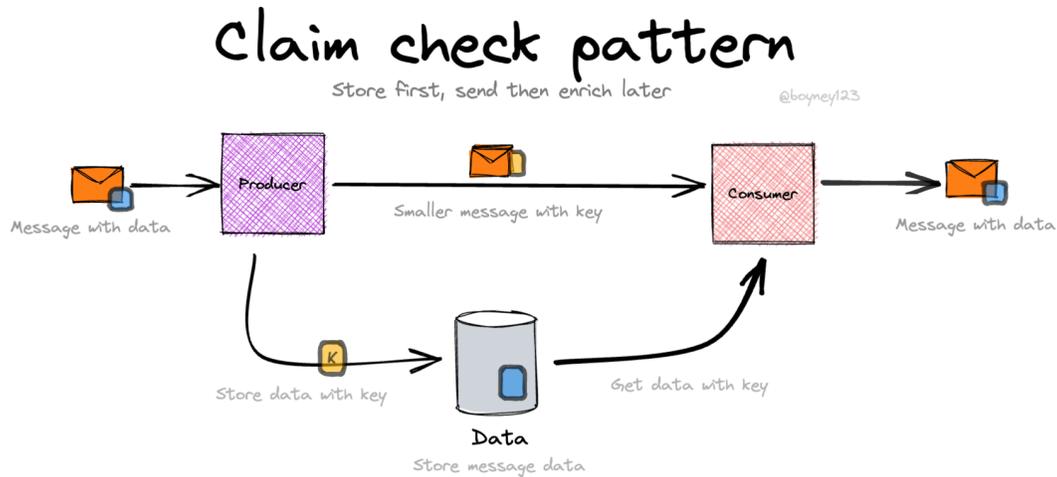


Figure 5: Claim check pattern

Sometimes you may want to store information first, and then send message downstream to consumers. This allows you to offload data or large events into a database and then send lighter events downstream.

An example use-case of this pattern would be to send large payloads (exceeding your event broker limits) to downstream consumers. First you store, then you use the key downstream to get the information back from the database.

Extra resources

- Claim Check Enterprise Integration pattern - Enterprise integration pattern for claim check, if you want to know more head here.
- Publish large EventBridge events with the claim check pattern - Blog post that I wrote, to help you use the claim check pattern with Amazon EventBridge.
- Code example using S3 and EventBridge - Pattern to show you how to use S3 and EventBridge with claim check

Commands vs Events

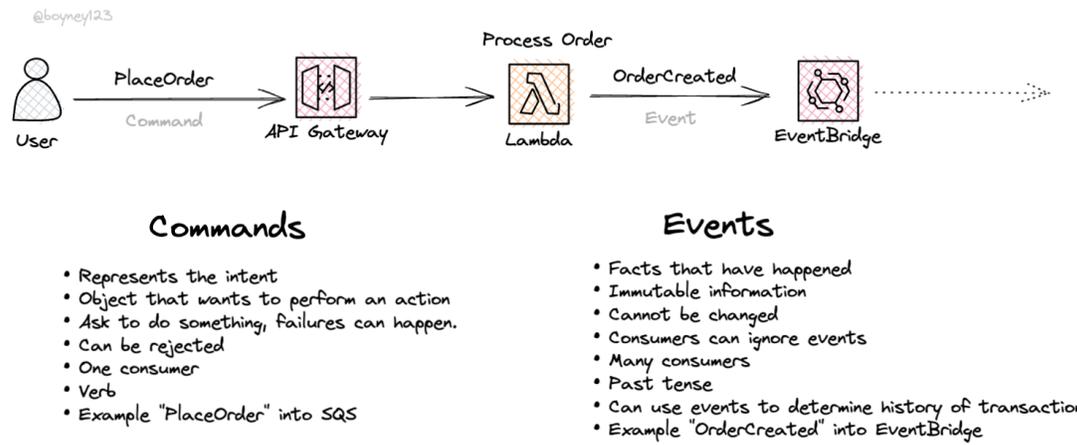
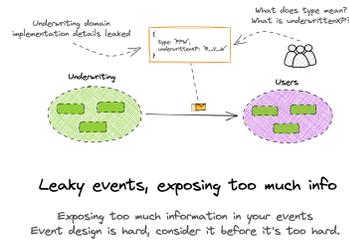
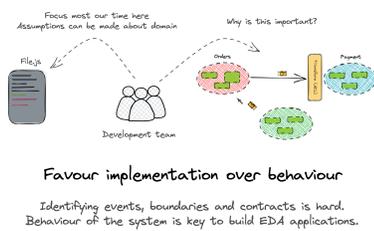


Figure 6: Commands vs Events

Understanding the difference between commands and events can be important when building event-driven architectures.

Common issues when scaling event-driven architectures



Common issues when scaling event-driven architectures

Common problems when going beyond the initial setup of event-driven architectures.

@boyney123

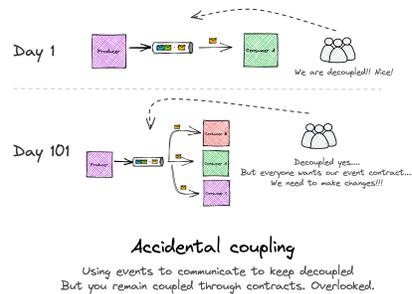
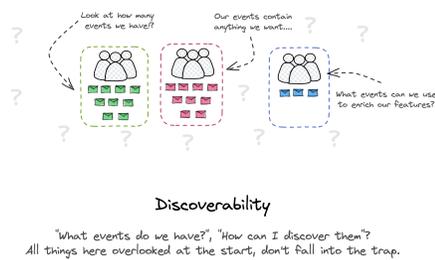


Figure 7: Common issues when scaling event-driven architectures

Talking to many folks in the community there seems to be shared problems that occur when you scale event-driven architecture solutions within your organisation.

Starting with a few producers/consumers can be great, but without considering a few fundamentals you may end up in the same place many others do, looks explore some of of these common issues.

Discoverability

- Events are added over time, 10s, 100s or 1000s. All flying through your architecture
- Many events can have many consumers, knowing **who is consuming events can help**
- Many people scale events, but **don't keep track on who is consuming what**. This can lead into event management issues (changing events, schemas etc).
- Many folks want to consume events to enhance, their functionality. But where can they find events in your system? Are they documented somewhere? Probably not?
- Event based architectures are super powerful and can help us scale and be decoupled, but without governance, standards and discoverability you will end up lost, who is coming what!? Can we change this field?

- Ways to help, documenting your events, explore tools and solutions that can help. (visual to help)
- Think about this up front, 2 years from now how will people discover your events?

Accidental coupling

- So, you are building event-driven architectures, using events to communicate between systems, you are decoupled? Maybe not.
- It's important to remember **we are still coupled by the event contract itself**; consumers rely on the contract of the event and schema (often overlooked).
- You have many ways to create events (e.g. notification, stateful events), depending on your choice depends on how coupled downstream consumers will be to the contracts.
- Keeping track of who is consuming what can help you make schema/payload changes.
- What is your schema management path? E.g no breaking changes? Forward compliable? Consider, in 2 years how are you going to make changes to these events?
- Making changes to events, do you want to produce two different versions and give time for consumers to migrate, this is a valid pattern that some follow.

Leaky events, exposing too much info

- What goes into your event? Do you just throw anything in? Probably not.
- When starting out it's often overlooked and payloads in events can be anything.
- Take time, think and consider standards in your events.
- Consider your domains and bounded context, consider what information should stay within a domain and what information can be exposed.
- Use a public interface/language between your bounded context and others.
- If you are consuming events, you might want to transform these, before external domain models leak into your domain.
- Spend time here, think about bounded context and the language used within each domain. Be mindful in what you expose.

Favour implementation over behaviour

- It's easy to get carried away with coding implementation details, and overlook the behaviour of the system, knowing what events should be fired when, and what should go inside these events.
- Events can be technical events, but also business level events, everyone having an understanding of your business domain helps (sounds simple right? So many assume they know business behaviours).

- Limit mental translations between teams, be clear on business intent and focus on behaviour then focus on implementation details.
- EventStorming can help identify your domains and bounded context.

Extra Resources

- Types of events - There are many types of events, understanding them can help.
- Learning Domain Driven Design - Some notes taken from the book “Learning Domain Driven Design” by Vladik Khononov.
- EventStorming - EventStorming is a great way to find events and domains, and also go that next level, start to map and shape your schema contracts.
- CloudEvents - A specification for describing event data in a common way. Want to standardise your events, why not use industry standard?
- AsyncAPI - Specification to help you define async APIS and event-driven architectures. Recommend looking into this.
- Amazon EventBridge: Event Payload Standards - Blog post I wrote awhile ago, inspired by community, this is an example of event standards and payloads.

Content enricher pattern

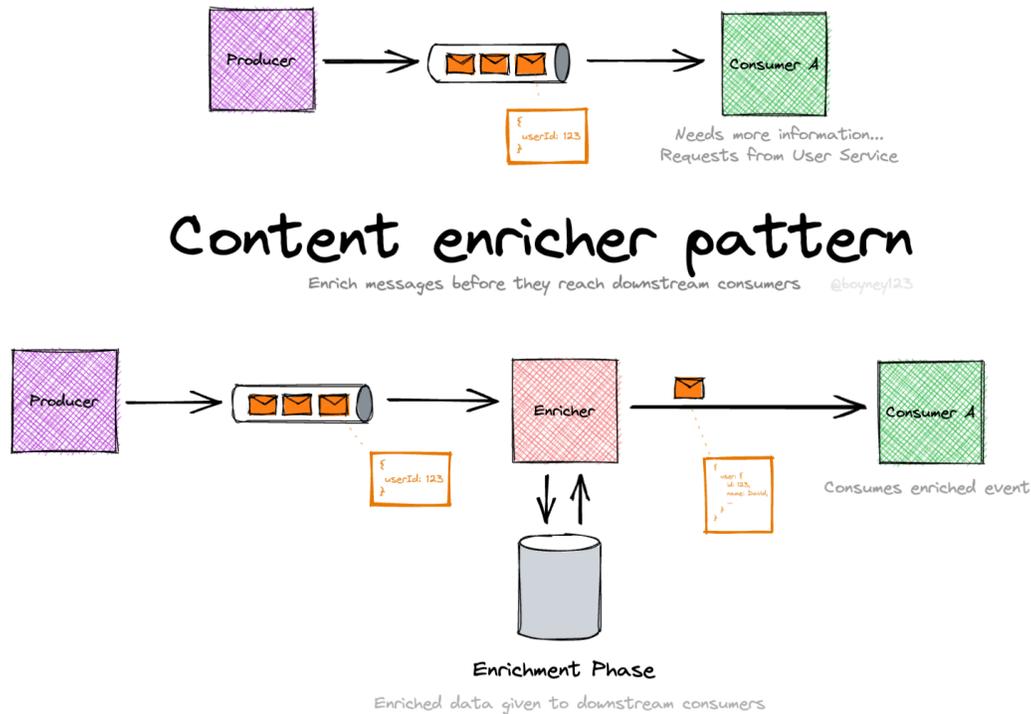


Figure 8: Content enricher pattern

Sometimes downstream consumers need more information (sometimes seen when events are notification events). Downstream consumers call external APIs or back to the producer to get information they require. Be careful with this pattern as it could lead to coupling.

With **content enricher** pattern you can add an **enricher** in the middle to pick up messages/events and enrich them before sending downstream to consumers.

Why enrich messages and things to consider

- Keeping the enrichment outside of the consumers domain, keeping consumer “pure”.
- Stop consumers fetching information they require from producer or other APIs.
- Enrichment pattern can lead to more code to maintain and manage
- If consumers need more information, are your event payloads or business boundaries, correct?

Extra resources

- Content Enricher Enterprise Integration Pattern - Great book with tons of patterns and information. Content online for free, recommend reading this.
- Enrich EventBridge events with Lambda - Blog post I wrote about enrichment pattern and using Lambda to enrich your events
- Enrichment pattern with EventBridge Pipe - Pattern that I wrote for ServerlessLand that uses EventBridge Pipes to enrich data before sending to downstream consumers.

Document your event-driven architecture

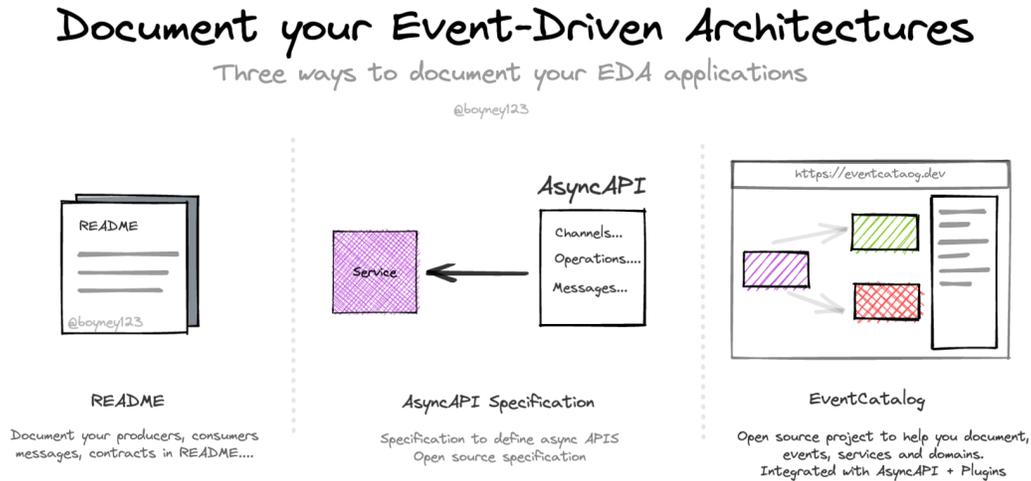


Figure 9: Document your event-driven architecture

When you dive into event-driven architecture you will see one of the main benefits is “loosely coupled” services. In fact many people will say “*producers should not know about consumers*”, this is technically true. . . . but overlooked.

Technically our *producers should not know about consumers* but operationally as humans we do. Common questions come up when building EDA applications over time:

1. What events/messages is this service producing?
2. What events/messages can I consume from this service?
3. What is the format of these events/messages?
4. What schema version do I use?
5. Who is producing what?
6. Who is consuming what?

These are common questions that will arise when building EDA applications.

Here are three areas that I consider that can help:

Using README files Simple and cheap. If you want to document your schema, or what your service is publishing or subscribing too, maybe readme files can help? Anything is better than nothing. (maybe you think nothing is OK, but as you scale your app high chance you might need some form of discovery)

AsyncAPI AsyncAPI is an open source specification for defining asynchronous APIs. The community has been growing over the past few years and many large organisations using it to help them define and write standards for producers and consumers. Community has a wide range of tools to help with integrating and documentation too.

EventCatalog I'm of course biased here, this is my own open source project called EventCatalog. I designed this to help people document their EDA applications, powered by markdown files and custom plugins. You can connect any system you want to EventCatalog and generate markdown files. This web interface gives your team a new visual way to navigate producers, consumers and domains. See example in action [here](#).

Extra resources

- AsyncAPI - Open-Source tools to easily build and maintain your event-driven architecture. All powered by the AsyncAPI specification, the industry standard for defining asynchronous APIs.
- EventCatalog - EventCatalog is an Open Source project that helps you document your events, services and domains.

Event Types



Different Event types in Event-driven Architectures

Exploring the different types of events

@boyney123

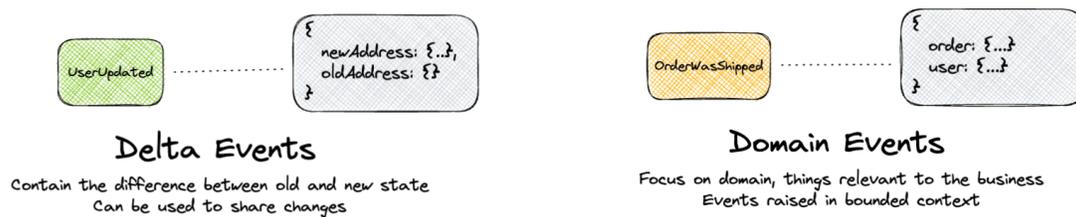


Figure 10: Event Types

When building EDA applications, it's important to know the different types of events you can publish, each has their own trade-offs.

Notification Events

- Minimal information
- Less risk of data being out of sync
- Consumers often need to fetch data
- Increases architecture coupling (callback for info)
- Producer/Consumer contracts kept minimum
- Lower risk of breaking contracts

Event-Carried State Transfer

- Enriched events (stateful)
- Higher risk of data being out of sync
- Consumers have the data
- Decreases architecture coupling

- Producer/consumer contracts more coupled
- Higher risks of breaking contracts

Delta events

- Stores difference between old/new
- Examples seen in change data capture events
- Can reduce complex in consumers needing to figure out what has changed

Domain Events

- Events that raised in the same bounded context
- Some folks reference these as internal events raised in bounded context
- Some folks reference these as business “important” events.
- People also refer to “integration” events. These are events used for integrations.

Extra Resources

- Best practices to design your events in event-driven applications - A talk I gave in 2022 around event design and trade-offs to consider.
- What do you mean by “Event Driven?” - Martin Fowler dives into event-driven architecture and talks about the different types of events
- The Event-Carried State Transfer Pattern - A great blog post on what ESCT is
- The event notification pattern - A great blog post on what notification events are

Event first thinking

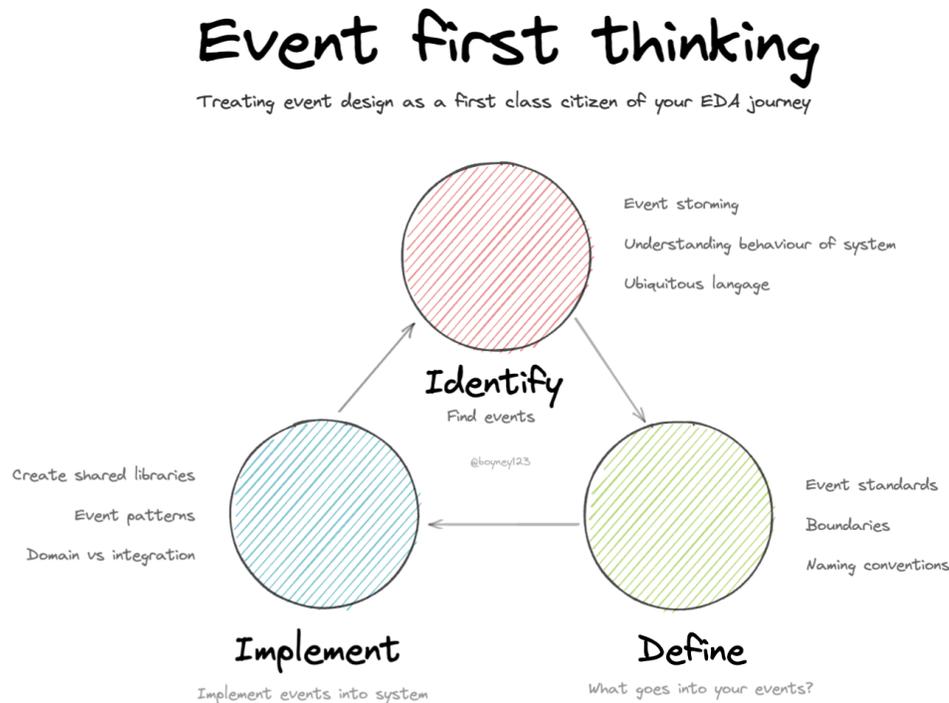


Figure 11: Event first thinking

Event identification and design within EDA applications is important, the more I researched and learnt about event design the more I believe it's a core part of an EDA application.

When we start our journey building EDA applications, we raise events from producers to downstream consumers, but what goes into our events, the structure, naming conventions, or the ability to identify our events is often overlooked.

“Event first thinking” are just some of my thoughts around all this and topics I presented last year at re:invent 2022 and EDA day in London 2022.

1. Identify

- Recommended using Event Storming with domain experts in your org to highlight the behaviour of your system and events that flow through.
- Find your bounded context and domains within your systems.
- Define a Ubiquitous language with your org to help communication between teams/parties.

2. Define

- Think about what you want in your events, do you want them to be notification, delta or event-carried state transfer or maybe something else?
- Think about documentation for your event, producers and consumers. How will people find the events you are publishing, what are the contracts? I created EventCatalog that might be able to help.
- Define naming conventions for your events, use these across your org and set standards, it can help.

3. Implement

- When you implement events in your producers think about shared libraries you might want to write. If you have standards, naming conventions or metadata for example, use shared libraries in your org to help. Can save time.
- Understand the design trade-offs when you choose different event types, understand them then implement them.
- Understand events “internal and external”. Are your events using within the same bounded context, are they used to communicate between bounded context? Depending on what kind of events they are will determine the impact of contracts and breaking changes.

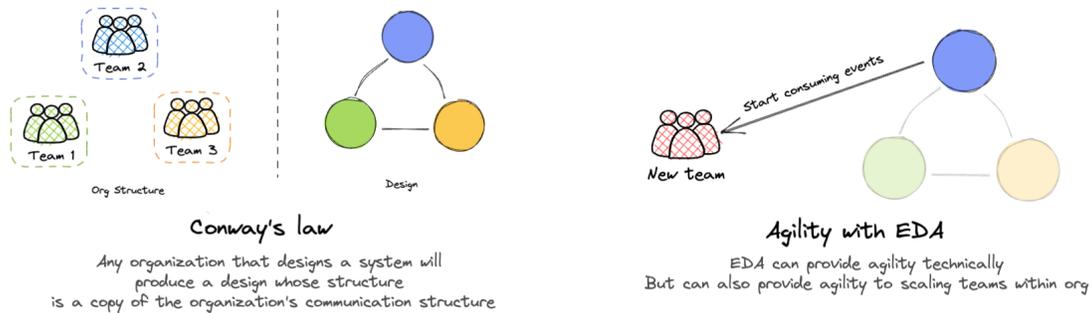
Repeat the process

I believe this process can be repeated throughout time. Business requirements change, architecture is naturally an evolution. You might want to repeat the process as time goes on.

Extra resources

- Best practices to design your events in event-driven applications - A talk I gave in 2022 around event design and trade-offs to consider.

Event-driven architecture and Conway's law



Event-driven architecture and Conway's law

Does event-driven architecture impact organization structure?

@boyney123

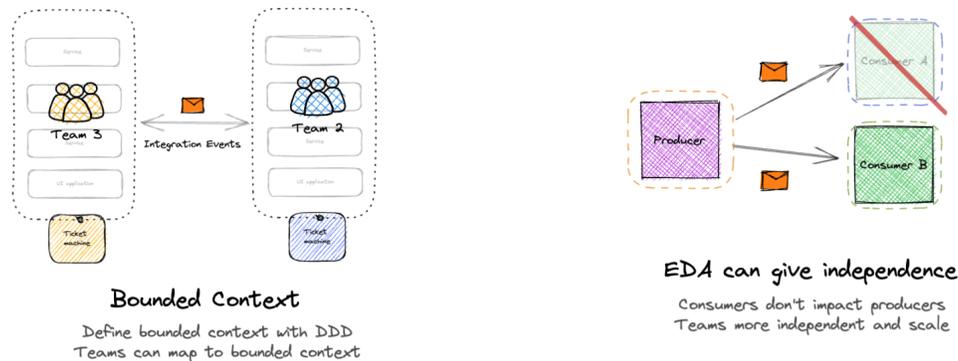


Figure 12: Event-driven architecture and Conway's law

Event-driven architectures give us the ability to create scalable, resilient and decoupled applications, coupled with domain-driven design principles we can start to use events as a form of communication between bounded contexts and teams.

Conway's law suggests that **the architecture of the system we build is a reflection of the teams that built it**, and when we couple with this domain-driven design and identify our bounded contexts and domains we start to naturally create domains and systems that exist in their own right and are decoupled from each other (sounds very similar to some of the benefits to event-driven architectures).

Agility with EDA A great benefit of event-driven architectures is the agility it can provide technical teams and also feature development. When producers raise events, consumers come and

go and choose to listen to the events (if they are interested).

EDA based architectures give organisations the ability to add more consumers to existing events, and as this catalog of events grows over time, more innovation and value can be captured.

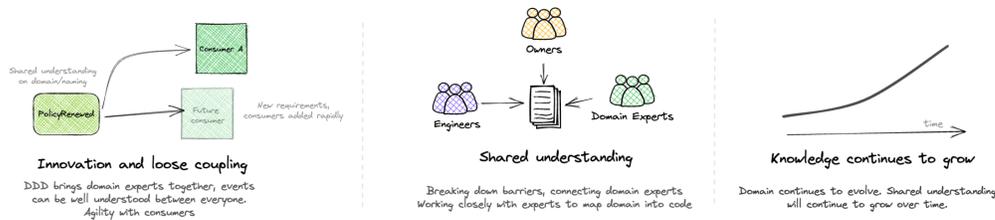
As you add more consumers, no doubt you might also add new bounded contexts which in reflect would affect your team structure (Conway's law).

Connection between EDA, DDD and Conway's Law There is a connection between domain-driven design, event-driven architectures and Conway's law, it's important to consider that when designing and implementing your applications, you might even see a natural org structure forming around your EDA solution and vice versa.

Extra Resources

- EDA coupled with DDD - Event driven architecture and domain-driven design work well together. Here is a visual to help you understand.
- Messaged between bounded context - Event design is important between bounded context, this visual explains mapping techniques for messages before you consume them.
- EventStorming - Use EventStorming to highlight your bounded context and events, this could help identify team structure and future org structure.
- EDA is a journey... you won't get it right first time - EDA is a journey as you identify domains, events, patterns, org structures, it will take time, and that's OK.
- Scaling EDA can be hard without docs - Some thoughts around documenting your EDA applications. As you grow you will need to discover events, producers and consumers.
- Conways law with Event Architectures - Some thoughts on Conways law from Martin Fowler and a mention of domain-driven design.
- Conways law and microservices - Short post on Conways law, DDD and microservices.

Event-driven architecture coupled with Domain-driven design



Event architectures coupled with Domain-driven design

Why event-driven architecture and domain-driven design work well together. @boyney123

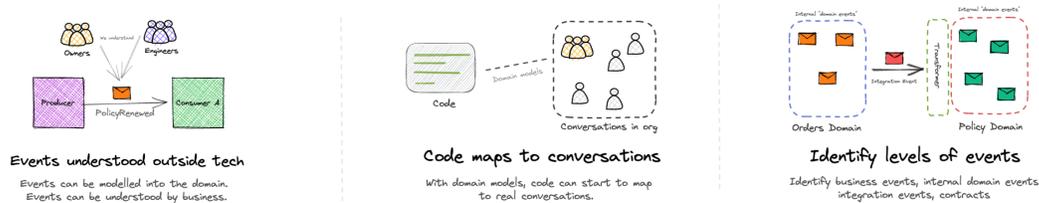


Figure 13: Event-driven architecture coupled with Domain-driven design

Innovation and loose coupling Innovation happens when we connect people together, solving issues and problems. Part of domain-driven design is connecting people, connecting domain experts with the solution that is being built and the people building it. It's only when we connect and understand our domain is we can start to innovate and build new solutions that can help. Couple this with event-driven architectures giving us a loose coupling between bounded context and services, we can reuse events and create new consumers when new requirements or innovation is required.

Shared Understanding Some of the best people I have worked with have a shared understanding of the domain they are working in, using the same terminology and modelling the domain into code, this can take time but well worth the effort. It's important to create that ubiquitous language.

Knowledge continues to grow Your domain is organic and your requirements and domain will evolve over time, once you have a map of your domain, it will continue to change. Be prepared for that.

Events understood outside tech When events are understood outside of the technical implementation new ideas can be generated. When domain experts or business owners understand domain events (events important to them/the business), they can start to come up with new features or solutions based on events that are already being dispatched vs having to write code from scratch. I have seen teams fly here.

Code maps to conversations When the domain is understood we can translate this into code, the code represents the domain, and our code can start to map to real conversations we are having. When you have well defined events these event names can be and will be brought up into conversation.

Identify levels of events When you highlight your bounded context, you will be using events inside and outside of your domain context. Internal vs External events, knowing the difference here can help. Identify, what is important for the business.

Extra Resources

- What is domain driven design? - Wiki page for DDD, start here to get a quick overview, although DDD is huge, it will take more than this page to understand.
- What is domain driven design? Continued... - A nice summary from Mathias Verraes about DDD, worth reading.
- Domain Driven Design and Event-Driven Architecture Podcast - Vaughn Vernon gives a podcast that dives into EDA and DDD. If you want to dive deeper worth a listen.
- Explicit vs Implicit events - Producers and consumers, how do they know what is in the event? Do they have a shared understanding? We are coupled but nothing stops us documenting, right?
- Eventstorming - Want to get started, trying to find events and your bounded context. Start with EventStorming.
- Document your event-driven architecture - Not really too much to DDD, but documentation can help you and your business find events and try and get that shared understanding.
- Event first thinking - Something I'm passionate about, think about events as first class citizens of your EDA design, use DDD to help model them.
- Ubiquitous Language - What is Ubiquitous Language? Visual to help.
- Messages between bounded context - How can we transform messages between contexts? Three patterns here.

EventStorming

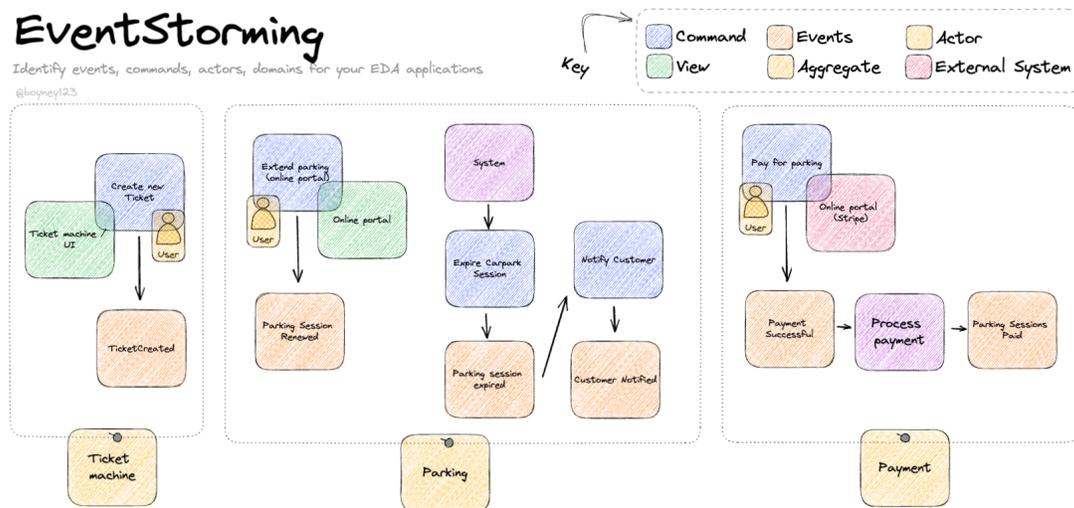


Figure 14: EventStorming

Need to figure out what events your system has? Or what commands generate events? Need to identify bounded context within your system? EventStorming can help.

EventStorming is a great method to help you identify your events, commands, aggregates and domains. EventStorming can also be used to help identify areas of duplication and complexity within your system.

It's important we understand the behaviour of the systems we try and build, event storming brings domain experts and engineers together to get a shared understanding and start to identify the behaviour of the system.

What is EventStorming? EventStorming is a workshop for collaborative exploration of business domains. You can use it on existing systems or new systems. Use EventStorming to accelerate your EDA applications.

What are the different parts / keys of EventStorming

- Orange: Events - Events written in past tense
- Blue: Commands - Command to trigger the event
- Yellow: Actors - Person who executes a command
- Green: View - View user interacts with to carry out task/command
- Pink: External System - Third party providers (e.g payment service)
- Yellow: Aggregate - Cluster of domain treated as a unit

- Purple: Business Process - Process command to generate event

Extra resources

- What is Event Storming? - Wiki link to that describes what event storming is
- EventStorming.com - Website with many resources, workshops and book about EventStorming.
- Awesome EventStorming - Huge list of resources to learn more.

Events as Data

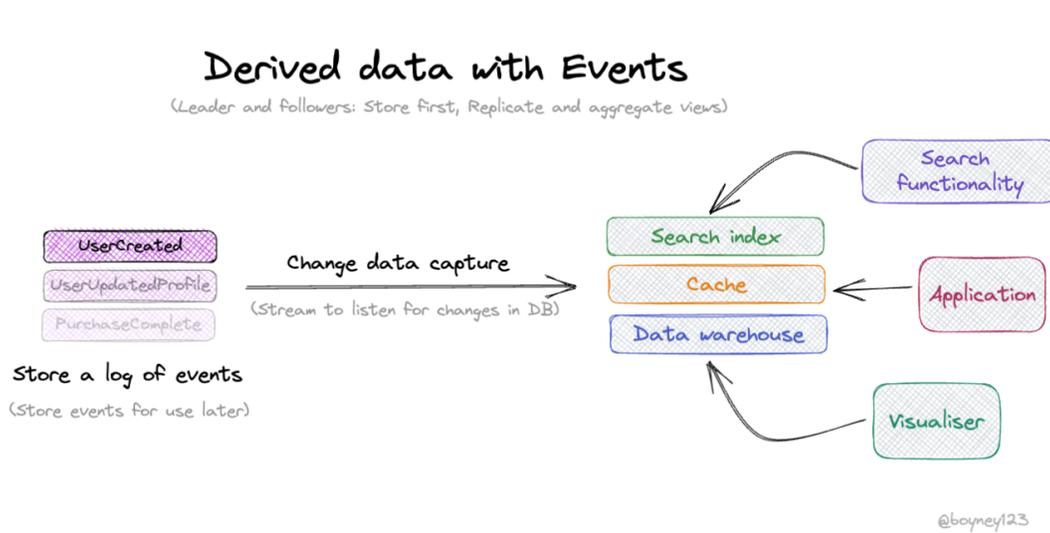


Figure 15: Events as Data

Using Event Sourcing to capture events in your system. The idea of capturing a set of events and using streams like “Change Data Capture” to process the information downstream.

Other systems have their own derived data using the events and create their own views.

Explicit vs Implicit Events

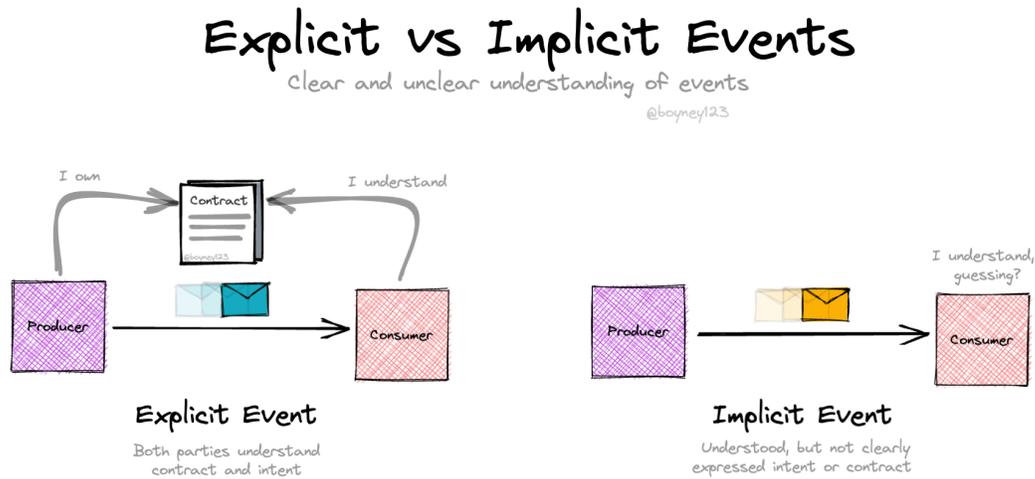


Figure 16: Explicit vs Implicit Events

When designing your events, I believe it's important to have explicit events for your event-driven-architectures. Define your events, define schemas and make sure the event itself is clear and intent can be understood. Also note producers that do not provide contracts (schemas) have increased chance of producing implicit events for downstream consumers.

Explicit Events

- Be clear on the event and it's intent
- Use clear naming conventions
- Use schemas to help define contracts
- Gives stability to both producers and consumers

Implicit Events

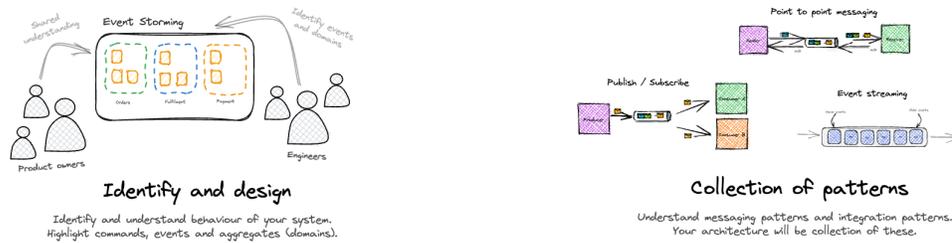
- Unclear event intent, try to avoid
- Unclear naming conventions
- Assumptions can be made
- Breaking changes possible without contract/schema.

Extra Resources

- EventStorming - EventStorming can help you identify your events in your system, using other stake holders it can be a natural way of defining explicit events, as events naming is often discussed.

-
- EDA Documentation - Documentation can help have explicit events. If consumers can discover and understand your events you are half way there.
 - Event First Thinking - Events are important, designing them, thinking about them, taking your time. Treat events as first-class citizens with Event First Thinking.
 - Event Types - Understand event types can help you along your event design path.
 - Commands vs Events - Two different things. Be clear if your message is a command or event.
 - Building Event-Driven Microservices - Thoughts generated from notes from this book.

Fundamentals of event-driven architecture



Fundamentals of event-driven architecture

Successful event-driven architecture, distilled into three main areas

@boyney123

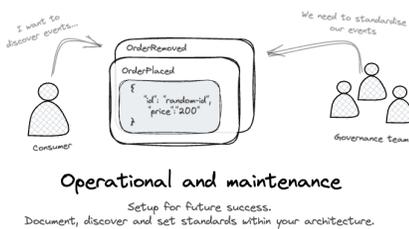


Figure 17: Fundamentals of event-driven architecture

When building event-driven architectures, how do we make sure they are successful today, tomorrow and in a few years' time?

I have distilled what I believe a successful event-driven architecture into three main areas, first is identify events and the importance in understanding the behaviour of a system. Followed by understanding messaging and integration patterns then finally spending time to understanding operational and maintenance going into the future.

Identify and design

- When you start building event-driven architectures you have two options, start to implement or stop and understand the behaviour of your system and identify your events.
- Spending time understanding your system, finding events, commands, systems and aggregates can really help you model a successful event-driven architecture.
- Event storming is a great workshop to run with your domain experts to identify events, commands and aggregates.

- Event storming brings domain experts together to discuss and negotiate how the system works and the naming of your domains (aggregates) and events. It's important to have event names that are explicit and clear.
- Once you have shared understanding of your system and identified your events, it can paint the picture you can use to start modelling your architecture and events.
- Remember event storming is not just a workshop you run at the start, come back to it in the future, keep using it to help retain the shared understanding.

Collection of patterns

- Event-driven architecture is just a collection of patterns, you may have messaging patterns and also integration patterns inside your architecture depending on what you need.
- Messaging patterns include patterns like point-to-point messaging, publish/subscribe, and event streaming. You can use these methods to pass messages/events around your architecture, and over time your architecture will likely have many of these messaging patterns.
- Integration patterns are slightly different, these are patterns you can use to help you overcome certain situations when dealing with event-driven architectures or messaging patterns. Examples of these would be claim check pattern, splitter pattern, or message translator pattern all can be found in enterprise integration patterns book.
- Understanding messaging and integration patterns can help you build successful event-driven architectures.

Operational and maintenance

- It's important to consider what your event-driven architecture will look like in 6, 12 or even 24 months. How will people find events? How will you change events? Who is producing what?
- Documenting your event-driven architecture can really help you keep maintenance under control when building your architecture. Start to think about documenting who is consuming/producing what.
- Having standards in your events can also help you scale your event-driven architecture. What information should be all events in your architecture? Maybe create a custom SDK to help you build these within your org?
- Try and think about problems you may have in the future, and spend time upfront thinking about practices you can introduce to help you mitigate away from these issues.

Extra Resources

- Event storming can help! - Event storming is a great workshop to help you identify and understand the behavior of your system. Run these workshops to help identify events, commands, systems and aggregates. Perfect way to get shared understanding and get started with event-driven architecture.
- Domain driven design and EDA - We can take so many tips from the domain driven design community when building event-driven architectures. Here is a visual to dive deeper.

- What is point-to-point messaging - Understanding messaging can help, here is a visual to help you understand what point-to-point messaging is.
- What do we mean by publish/subscribe - Notify downstream consumers that something has happened. When you build event-driven architectures no doubt you may need this pattern. This visual dives deeper.
- Discover your events - In the future people will want to know what events you have, and how you can discover them. This visual dives into event-driven architecture documentation options with some open-source projects that can help.

Good and hard parts of EDA

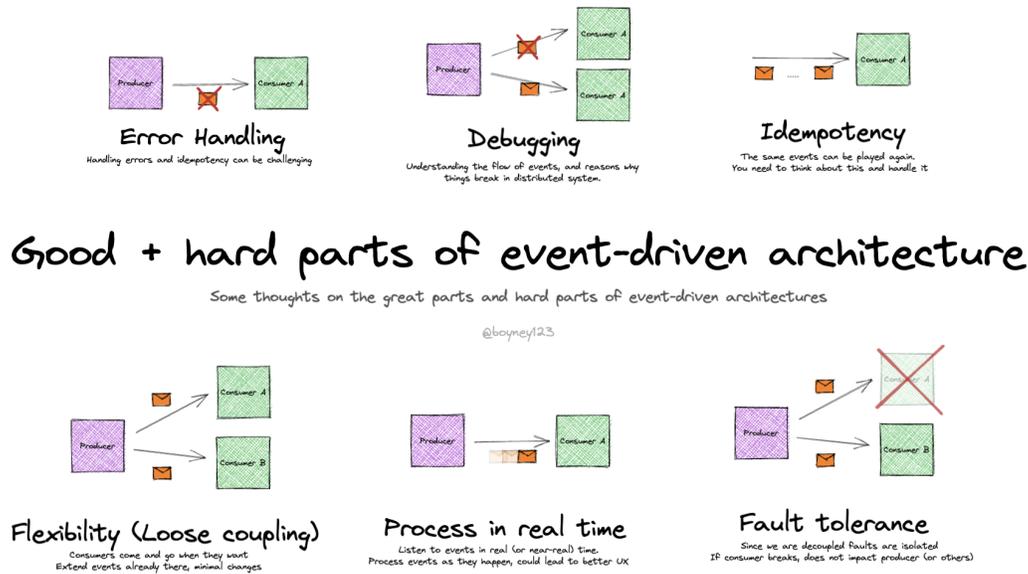


Figure 18: Good and hard parts of EDA

Good parts

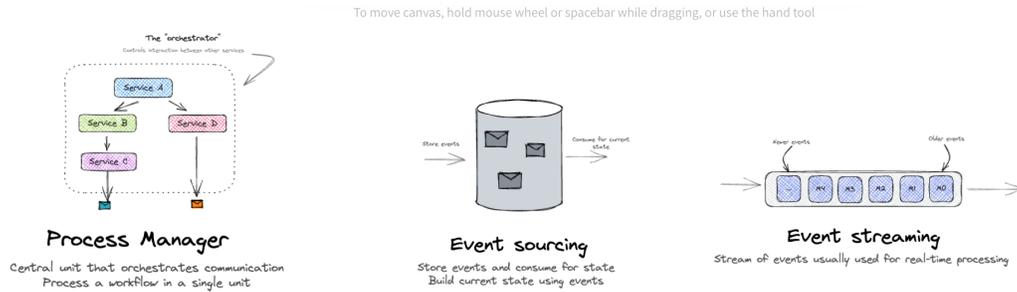
- Flexibility - The ability for consumers to come and go without impacting the producer. When new business requirements occur, existing events can be listening too and consumers can be created.
- Process in real time - Some systems can allow for events to be processed in real-time or near-real time. Thinking about the user experience, could this lead to a better UX?
- Fault tolerance - Consumers are decoupled by nature, if a consumer was to break or fail to process the event, the failure is isolated to that consumer.

Hard parts *Things to think about.*

- Error Handling - When errors occur, being able to track the error and understand what is going on. Need to consider DLQ or ways to capture events before they are gone forever... if that's important to you.
- Debugging - Producers and consumers are distributed by design, so being able to trace and debug can be hard.

- Idempotency - Key to think about idempotency. If the same event was to be replayed into your consumer, what will happen? Avoid side effects, have the same outcome if event replays were to happen.

Inside event-driven architectures



Inside event-driven architectures

What patterns will you come across when building event-driven architectures?

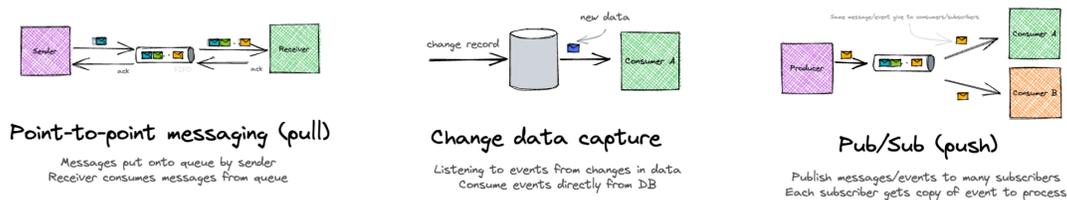


Figure 19: Inside event-driven architectures

When build event-driven architectures, you find yourself using many different patterns, not just one. When you dive into EDA applications you see a mixture of point-to-point messaging, pub/sub, choreography, orchestration, maybe some event sourcing and much more. . .

I believe using these patterns together help build a resilient, available and scalable event-driven architecture.

Process Manager

- The orchestrator that manages a workflow or process
- We have to perform business logic, this encapsulates that
- Often events are triggered from process manager to other consumers
- Example of this is AWS Step Functions

Event Sourcing

- Store events and use this information to calculate state
- Downstream projections can use this to calculate their own view of world
- Change data capture popular to listen to changes
- Audit is great here, as you can see everything that is happening.

- Example is storing events into NoSQL DB like Amazon DynamoDB

Event Streaming

- Often used to process information in real time
- Examples seen with user interactions on some form of interface (e.g. clicks)
- Messages are put onto the stream, consumers pick where they want to listen
- Offsets are used by consumers to read messages from the stream.
- Example of this is Amazon Kinesis

Point-to-point messaging

- Send messages to a channel for downstream consumers to process
- Multiple consumers can pick up messages from queue for concurrent processing
- Highly scalable pattern
- Example of this is Amazon SQS

Change data capture

- React to changes when they are made against your data
- Listen for new, delete or updates
- Attach consumers to changes to process information
- Example of a DB that supports this is Amazon DynamoDB

Pub/Sub - Fire notifications out to downstream consumers - Fan out events - Consumers get own copy of event - Create for decoupling applications and scaling teams - Example of this is Amazon EventBridge

Extra resources

- Point-to-point messaging - What is point-to-point messaging? When should you use it? Visual I created to help you
- Choreography vs orchestration - What's the difference? Using process managers to orchestrate workflows or using events to communicate between services, maybe a world of both? More than likely...
- Pub/Sub - What is Pub/Sub? When do events get pushed to consumers? Why use this pattern? Visual here to help
- Event-driven architecture with domain-driven design - How does domain-driven design help us build event-driven architectures? Take a look to understand why.

Internal vs External Events

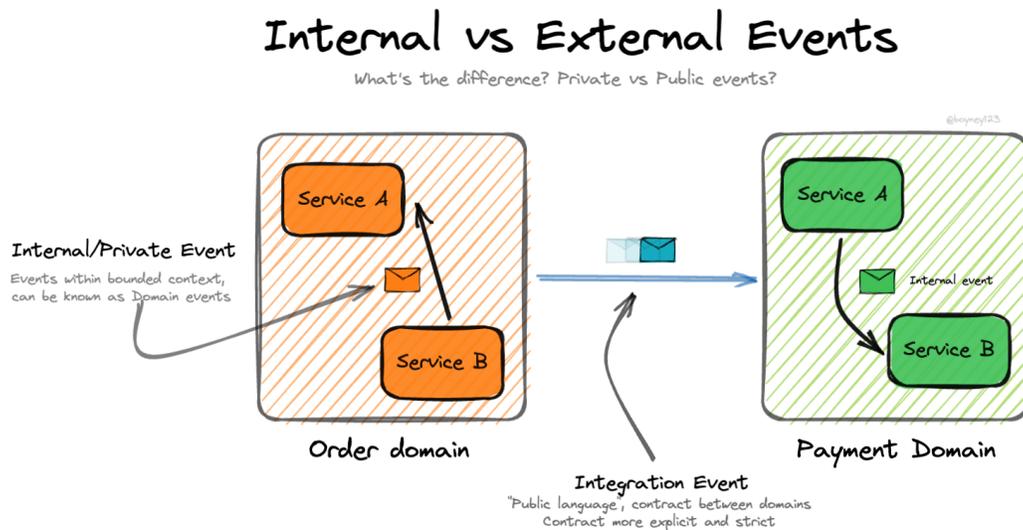


Figure 20: Internal vs External Events

When you build event-driven applications you can use events to communicate between services and boundaries. Many people use EventStorming to highlight events and business domains.

Within a bounded context you can have many different services and communicate between these services using events, sometimes these are referred to as **“private”**, **“internal”** or **“domain”** events (depending on what you read...). These events belong within the bounded context, they can be exposed to implementation details and raise events assuming that downstream consumers understand the domain language used in the boundary and implementation details, they are “private” and not meant for “public” consumption.

Using events to communicate between bounded contexts (remember this can be within in your organization or outside your organization) can be referred to as “public” or “integration” events. Event contracts are important here, and ideally you do not want to expose private or implementation details of your domain in these. Consider a public language to communicate between systems (Defining a ubiquitous Language can help here).

Private/Internal Events

- Can expose internal implementation details
- Use a language that the domain understands, external domains may not understand this
- Contract is important, but depending on how “close” the services are within boundary, could be relaxed?

Public/Integration Events

- Should not really expose implementation details of the domain
- Events use a language that is shared between the business or boundaries, no assumptions made
- Contract is extremely important. Outside your domain you have less knowledge of who is consuming your events, don't break them.

Extra resources

- Learning Domain Driven Design - Some notes taken from the book "Learning Domain Driven Design" by Vladik Khononov.

Journey to event-driven architecture

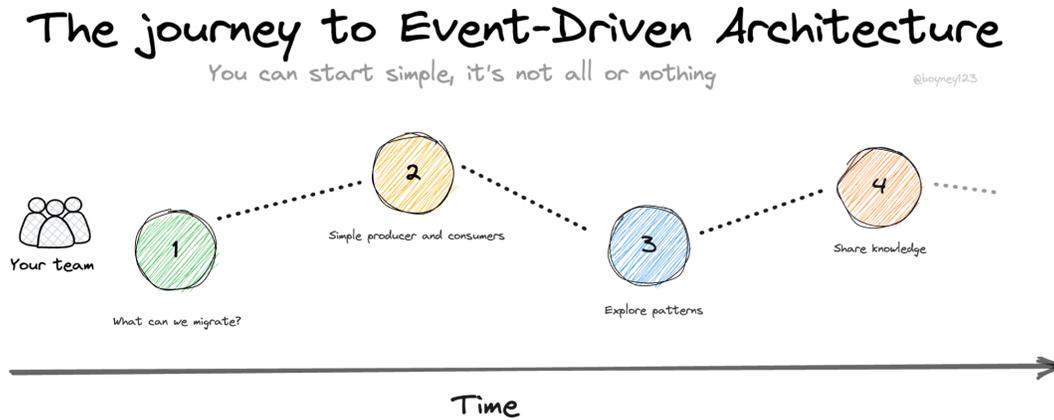


Figure 21: Journey to event-driven architecture

Software changes all the time. Requirements change, technology changes and patterns change. Using event-driven architectures can be great, as they allow you to be agile and adapt to change fast, but the road to implementing event-driven architecture does not happen overnight.

Implementing an event-driven architectures is a journey, a technical journey but also a journey for your organisation. There are many benefits to building event-driven architectures but the implementation itself is a continuous effort (like all software we write!).

If you have legacy code and you want to move to an EDA application, you can use migration strategies to slowly migrate services into an EDA landscape (if it makes sense)! If you are looking to migrate I recommend using EventStorming to identify the behaviour of the system and start raising some basic events and consumers. Grow the producers and consumers over time and start to decouple your applications. Find the natural bounded context in your business domain and start to communicate with events.

If you want to build event-driven applications or already do, I believe it's a journey and it will be a continuous journey, paired with ever changing business requirements EDA can be a great option to remain agile and adapt to change.

Start small, and grow your implementation over time.

Extra resources

- EventStorming Visual - Visual to help you learn about EventStorming.
- EDA Guide - Short guide to help you get started with event driven architectures.

Message Delivery



Message Delivery

At most once, at least once and exactly once

@boyney123

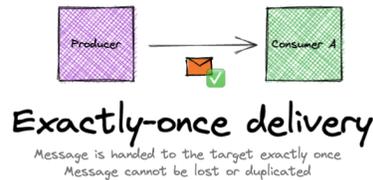


Figure 22: Message Delivery

Different systems will provide different messaging solutions, understanding these can help you understand how your messages will be given to downstream services.

At-most-once delivery

- Delivery means message will be delivered once
- If it fails it can be lost

At-least-once delivery

- Message may duplicate to consumer.
- Multiple attempts can be made to deliver message.
- Important to have idempotent consumers
- Multiple attempts can be made to deliver the message to the target until one succeeds. This means messages can be duplicated but not lost.

Exactly-once delivery

- Message is given to the target exactly once

- Message cannot be lost or duplicated

Extra resources

- At-least-once delivery - Short blog post to help.
- Exactly-once delivery - Short blog post to help.

Message Queues vs Event Brokers

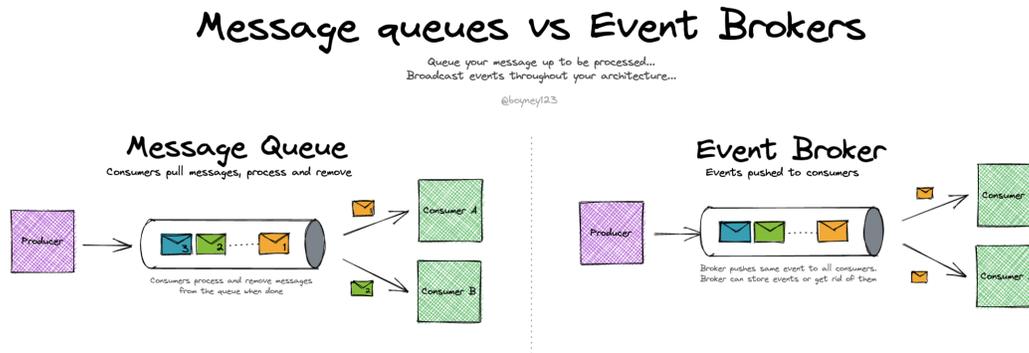


Figure 23: Message Queues vs Event Brokers

Message Queue Messages are put onto a queue and a consumer consumes the message and processes them. Messages are acknowledged as consumed and deleted afterwards. Messages are split between consumers which makes it hard to communicate system with events.

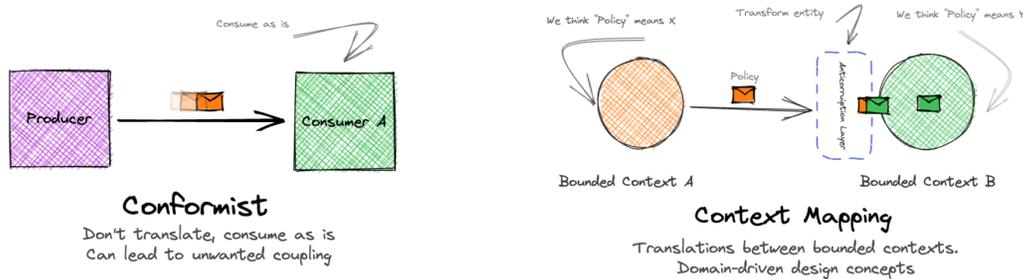
Example of this would be Amazon SQS. Publish messages to the queue and then listen to them, process them and they are removed from the queue.

Event Broker Event brokers are a push system, they push these events downstream to consumers. Example of this would be Amazon EventBridge.

Extra Resources

- Building Event Driven Microservices - Notes mainly from this book by Adam Bellemare

Message translator pattern



Message translator pattern

Why would you want to transform messages before consuming them?

@boyney123

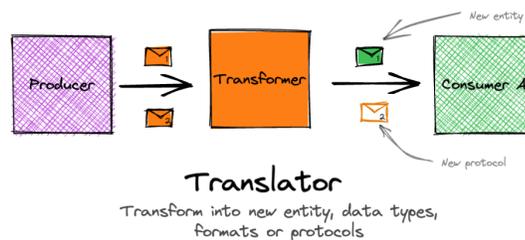


Figure 24: Message translator pattern

Use the message translator pattern to transform data format into a another one.

Examples of this might be transforming the message into a new entity (e.g. Stripe Payment into your own internal model), change data types (e.g. concatenate first name and last name into a new field) or new protocol (transform JSON into XML).

When building event-driven applications you may think you are decoupled by design, but the **message/event contract itself can couple you...**

Things to ask yourself when consuming events:

- Should my consumer conform to the event?
- What are the risks if we consume as is (this might be OK!?)
- Should we transform the message/event?
- Does this event/message conform to our understanding (ubiquitous language of our domain)

If you just think about these questions when consuming events, you are already half way there.

How does domain-driven design fit into this? When you define your bounded contexts (aggregates of services for example), you might be using messages/events to communicate with each other (integration events). There are different patterns to consider when consuming events rather than just conforming to the event payload/structure, it's worth checking them out.

Summary

- Use translator to transform messages
 - Transforming messages can help you keep your producing and consuming applications decoupled
 - When both parties conform to the structure of the message, you may run into coupling, this might be OK, but just be aware.
 - Having transformations on the edge of your bounded context can help isolate the need for change when changes occur in your schemas/messages/payloads.
-

Extra resources

- Message Translator Integration pattern - Enterprise integration pattern for message translator, if you want to know more head here.
- Transform messages between bounded context - Conform, Transform or open-host service when listening and consuming events. Patterns worth checking out if you are interested in the transformation pattern.
- Content enricher pattern - If you are after more patterns, check out the enrichment pattern. You can enrich your messages/events before sending them downstream to consumers
- Claim check pattern - Simplify your events by storing data first and a reference back to the event downstream, save on payload size.

Messages between bounded context

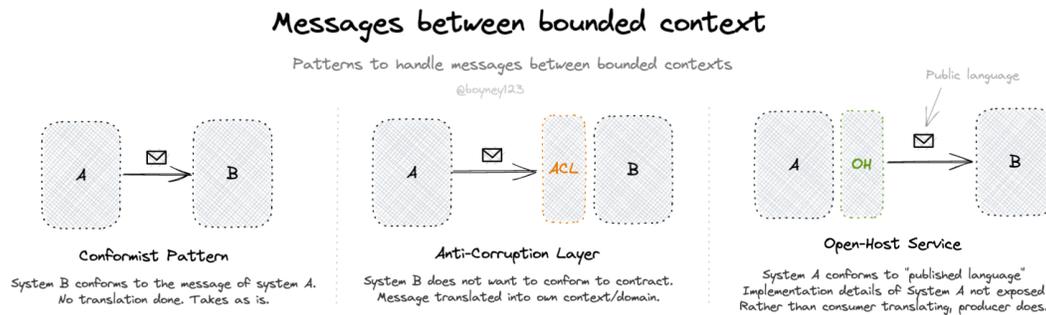


Figure 25: Messages between bounded context

When sending data between bounded context you have a few options to handle the data/contracts.

Conformist Pattern

- System A sends data to System B. System B does no translation of the data before using it within it's bounded context.
- System B conforms to the data/contract.

Anti-Corruption Layer

- System B does not want to conform to the contract of System A.
- System B translates the data into a domain model it can understand.

Open-Host Service

- System A sends data to System B but translates it before it is sent.
- Public language is used between systems.
- Integration details of System A is still locked away, but public interface/message exposed.

Extra Resources

- Learning Domain Driven Design - Notes taken from the book "Learning Domain Driven Design" by Vladik Khononov.

Point-to-point messaging

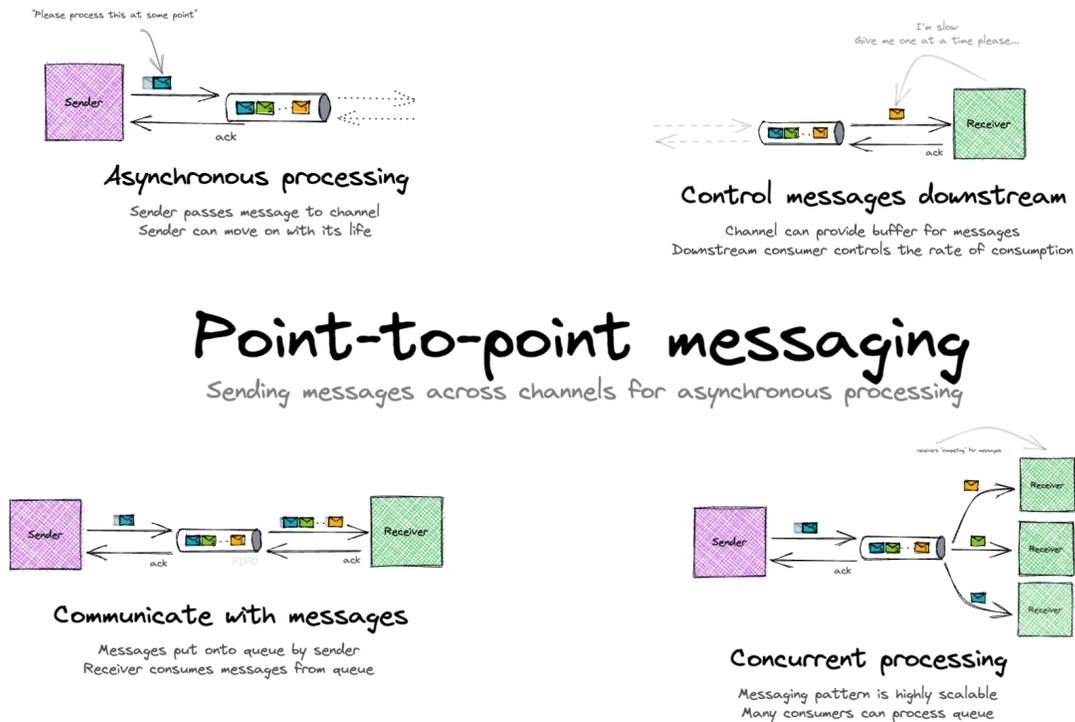


Figure 26: Point-to-point messaging

Point-to-point channels are used for handling messages between a sender and receiver. The sender puts messages onto the channel and the receiver consumes these messages from the queue (at the rate they want).

Great for scalability Point-to-point channels can have many concurrent receivers, this is what makes them a powerful for scaling, you can scale your processing of messages from the queue. Having many consumers can also be known as “competing consumers” as described by the enterprise integration patterns book.

Typically, the channel controls which consumer has which message as consumers pull messages from the queue.

Reducing pressure from consumers You can also see point-to-point messaging being used to release pressure from receivers. Receivers can control the rate of consumption; this is great if you have an API that cannot handle the throughput of messages.

Just one of many patterns Your event-driven architecture will consist of many different patterns and point-to-point messaging is just one of them. If you want to fan out messages to consumers you need to look at the pub/sub pattern.

Extra resources

- Understanding Pub/Sub - Visual and resources to help you understand pub/sub pattern.
- Competing consumers - When using point-to-point messaging you can have many receivers, this is known as competing consumers.
- EDA Guide - In our EDA guide we also have many more patterns and information to help you dive deeper into EDA.
- Eventual Consistency - When you have async message processing, the task/process downstream can take time, you need to know how eventual consistency can effect your architecture/application.

Producer and consumer responsibilities

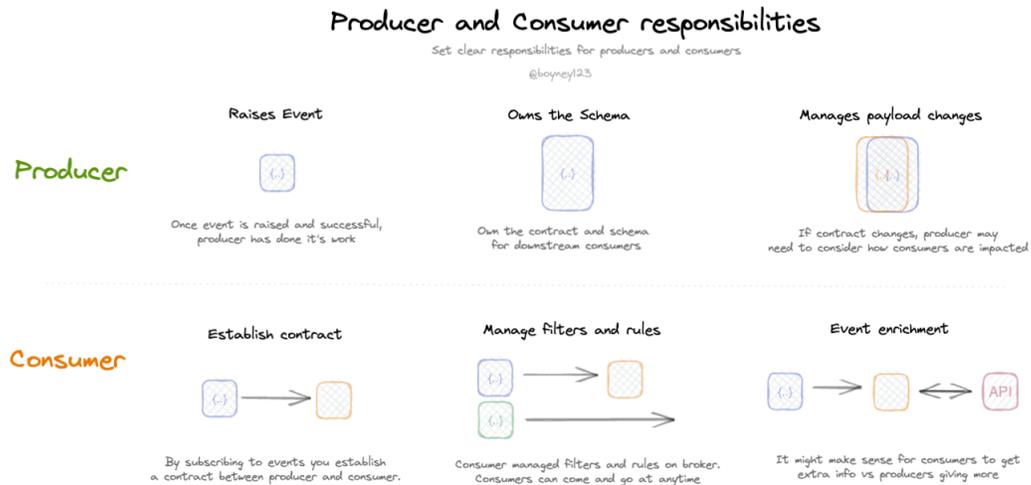


Figure 27: Producer and consumer responsibilities

When building event-driven applications it's important to know the responsibility of the consumer and producer. This diagram represents my thoughts on it, but I think from design to design it may differ.

Use this diagram as a base, but you may want to adapt it based on your architecture.

Publishing events, without any consumers...



Figure 28: Publishing events, without any consumers...

Should we publish events even though we don't have any consumers? Some say no, but others think there is value. Let's take a look.

Future extensibility

- Event-driven architecture allows us to be decoupled and have an architecture we can extend when consuming new events. If we want new events, we go back and add them, what if the events are already there? Just not consumed yet?
- Having events without consumers is something to get your head around, but it would mean you can add consumers in the future with little or no effort.
- If you had a catalog of these events, consumers could come and go as you see fit, that's the whole point isn't it?

Write the code, publish the event

- You are in the code anyway, writing the feature, why not emit the event when a business process has been done? Example would be sending an email, you write the code to send the email, then why not raise the event too.
- Publishing events without consumers or any "need" might seem counter intuitive though, this is code and events we have to maintain, so does that come at a cost we have to consider?

- Some brokers may charge for publishing events, so that is also something to consider.

Interest over time

- Today your consumers may not be interested in your events, in the future they might... if the events are already there to be consumed, the effort of integration could be lower?
- Maybe identify which events you want to publish “without consumers”, if they are internal private to that domain, the maybe that doesn’t make much sense? But if they are business domain events, it could?

Summary Personally, I’m not sure where I sit on this, I can see the value but also any code we write or events we publish we have to maintain. But this is something that I never really considered and come across, so it could be an interesting pattern to explore.

Maybe identify core business events, and raise them, there will be a higher chance of consumption vs private or technical events you may raise.

Extra Resources

- Fundamentals of Software Architecture - Idea came from this book, talking about Event broker topology. Great book to dive deeper.
- Document your events - If you are going to publish future events, it’s worth documenting them for consumers.
- Event Storming - Publishing all events might not be correct, but maybe some core business events might make sense? Use your judgement.

Schema Management

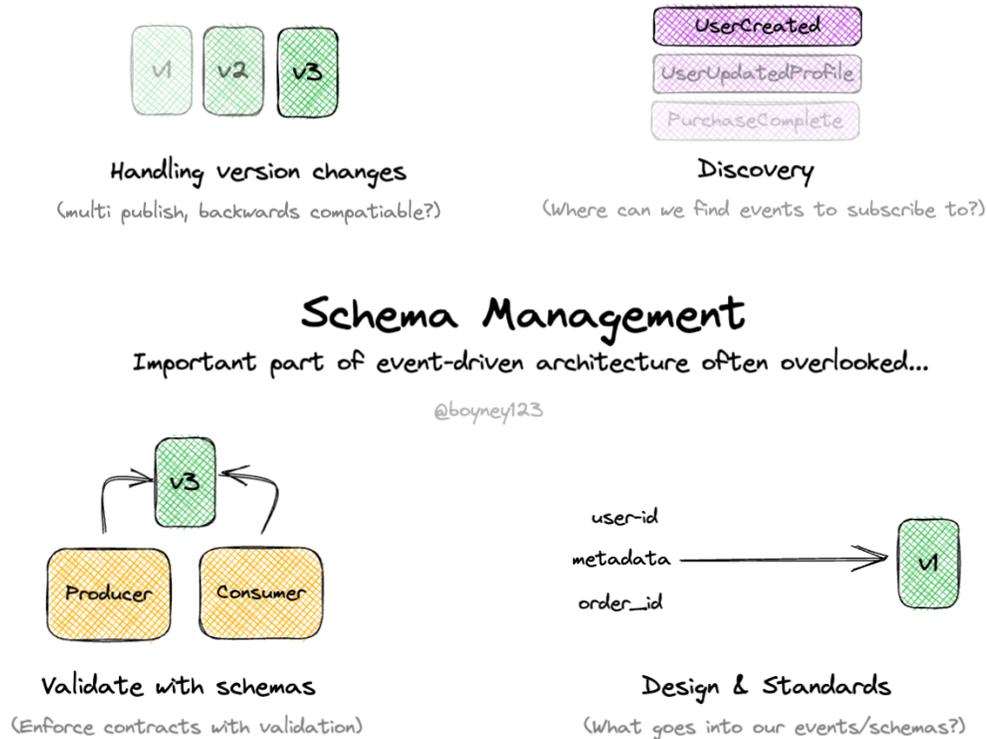


Figure 29: Schema Management

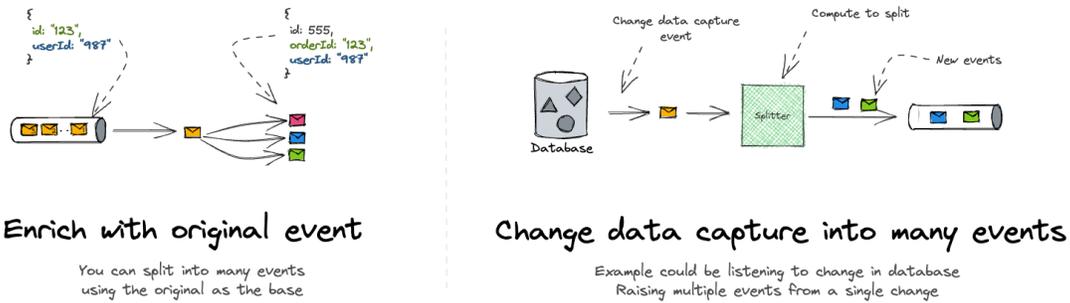
As EDA applications grow developers often find schemas to be a more important part of their EDA application.

1. What events can I listen to?
2. How are we going to manage change in our events?
3. What do the events look like?
4. What goes into our events?

Extra Resources

- AsyncAPI - AsyncAPI specification, the industry standard for defining asynchronous APIs.
- CloudEvents - A specification for describing event data in a common way
- Event Payload Standards - Blog post about Amazon EventBridge payload standards, lessons can be applied anywhere.

Splitter Pattern



Enrich with original event

You can split into many events using the original as the base

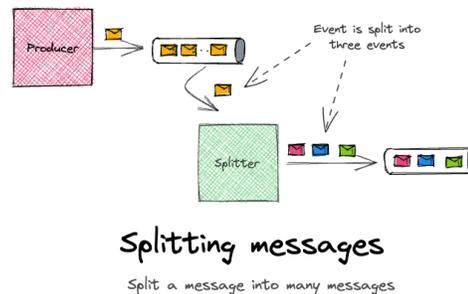
Change data capture into many events

Example could be listening to change in database Raising multiple events from a single change

Splitter pattern

@boyney123

Splitting messages/events into multiple for downstream consumers



Splitting messages

Split a message into many messages

Figure 30: Splitter Pattern

The Splitter Pattern from Enterprise integration patterns, takes an event/message and splits it into many for downstream consumers.

Splitting Messages

- If you have a large message/event you can use a splitter to split it into many messages/events for downstream consumers.
- Using a queue an example could be the splitter takes a message from a queue, splits it and puts it back onto a new queue/channel.
- This pattern can be helpful if you want to process large messages or events, or have a business use-case where it might make sense to split. Example could be taking an order and splitting into it's sections (as seen here).

Enrich with original event

- When you split your event, you may want to duplicate the information into the child events, depending on your data strategy. If you duplicate then downstream consumers may have the information they need to process, or they can go back to producer/API to get information they need.
- Splitter consumes original event, uses information to build payload for child events.

Change data capture into many events

- Databases that support change data capture events allow us to react to events when data is changed in databases.
- An example use-case could be listening for database changes (using change data capture) and transforming these events into multiple events downstream.
- An example of this could be using DynamoDB stream, into EventBridge Pipes to create many events downstream. [Example here](#).

Extra Resources

- Splitter pattern from enterprise integration patterns - This pattern is from the enterprise integration pattern book, if you want to learn more, recommended checking this out.
- Aggregator pattern - The opposite to the splitter, this takes events in, aggregates them into one message/event downstream.
- Example using DynamoDB with EventBridge Pipes - Some code examples to show you how you can do this with AWS services, all open source.

Sync vs Async Communication

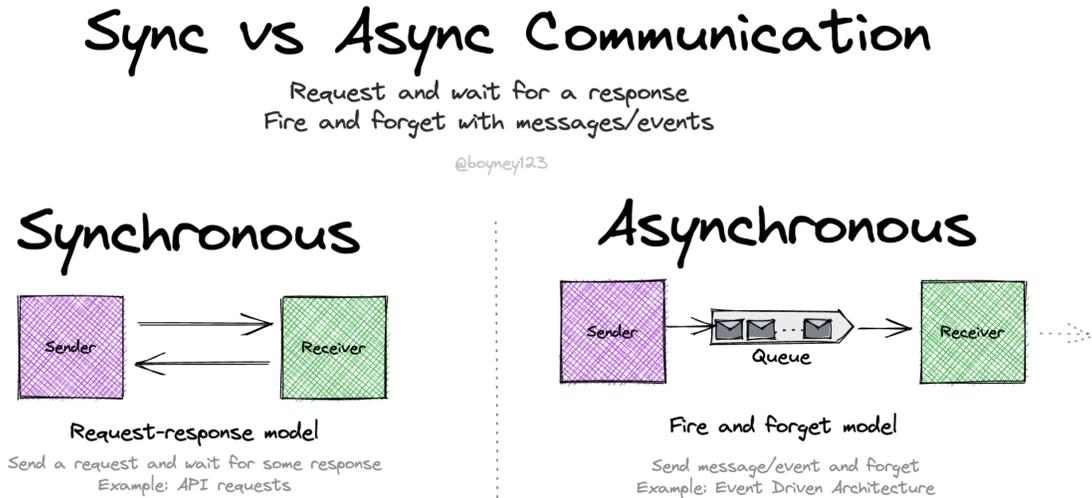


Figure 31: Sync vs Async Communication

When we design our architecture and build our services it can be useful to know the different types of communication patterns. Although pretty simple, I think it's important to know the difference between sync and async.

Sync

- Normally seen as the “request/response” pattern. Example would be taking a phone call, you need both parties there.
- An example of this could be a command, many API calls want something to happen.
- Fails fast, simple and low latency
- What happens when many requests come in? Need to scale the service, could you use Async pattern instead? Something to consider.

Async

- With EDA a producer can fire an event and forget it and allow downstream consumers to process it (if they want...)
- With messages, a message could be added to queue and downstream services control the speed of injection and process async.
- Async provides a loose coupling between services/systems.
- Receiver can control the rate of consumption of events/messages
- Example of this would be Amazon SQS.

Extra resources

- AWS: Re:invent 2022 - Building next-gen applications with event-driven architectures - Video on building EDA applications and speaker talks through communication patterns.

Things to consider when building EDA architectures

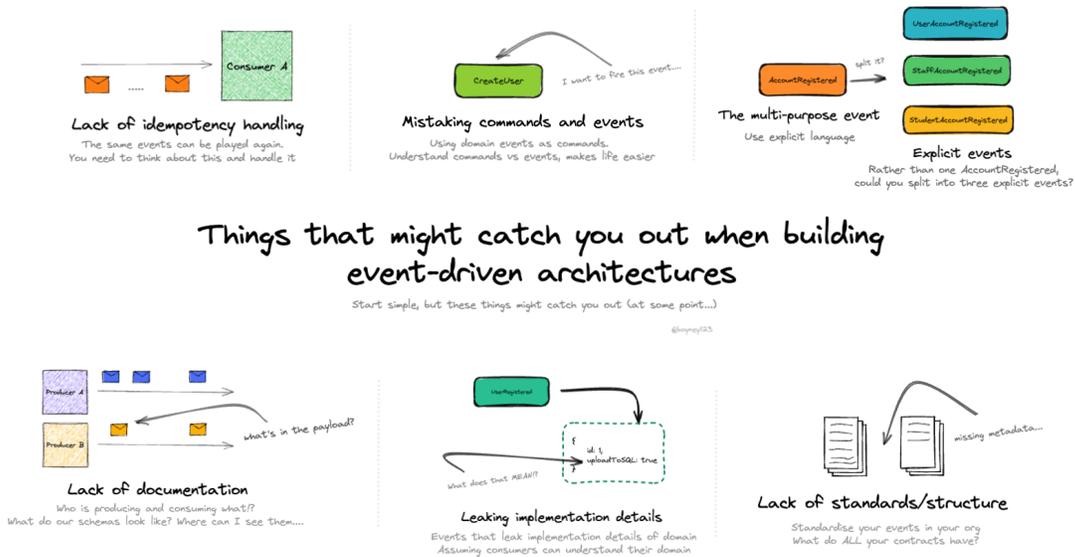


Figure 32: Things to consider when building EDA architectures

When you start building event-driven applications you start to write producers and consumers and you start to see value quite fast. Over time as more producers and consumers are added you will start to **face some common issues** (*these issues listed are all problems I have experienced and seen in the community*)

Lack of idempotency handling Consumers may process your events more than once (e.g. if events need to replay, or failures etc), your consumers need to handle this. You need to make sure your consumers produce the same outcome regardless of how many times it was called with the same event, having side effects in your consumers (different results) can lead to issues.

Mistaking commands for events An easy one when you get started, mistaking commands vs events. Here is a visual to help you understand the difference.

Multi purpose events When you start designing and implementing your events, it's easy just to add on that one extra field. . . . "what if we add a **type** field here", this works and might provide value, but over time your consumers might have a hard time understanding the **intent** of the event. Rather than having one event to rule them all, why not split them out? *Something to think about. . . .*

Lack of documentation Let's face it, many people don't like to write documentation, especially as you start to build your EDA applications. Overtime as you add producers and consumers it can

be hard to keep track of who is producing what, and who is consuming what. Adding documentation can help. Here is visual with resources to help you.

Leaking implementation details When you have clear boundaries of services, it can be easy to leak implementation details of your service in your event. Make your events explicit, don't get consumers to guess what is happening, don't confuse them with implementation details.

Lack of standards/structure Overtime you might have 10s or 100s of events in your architecture, think about what goes into these events, maybe you might use CloudEvents or define your own standards, spend time here and think about it.

Extra resources

- Document your event-driven architectures - Visual here to help you understand ways to document your event-driven architectures.
- Identify domains and events with EventStorming - Think about how you are going to identify events in your system. EventStorming can help.
- Message delivery - How are your messages/events getting to your downstream consumers? Understanding this can help.
- Events are important - Event First Thinking - Treat events as a first class citizen in your architecture, doing this from the start can help you.
- Know the difference of types of events you raise - Notification events, Delta events, Event carried state transfer... what does it mean? Important to know.
- Commands vs Events - Quickly learn the difference between commands and events, 5min here can save pain later on.
- Passing events through context - Find your business domains and use events to communicate, you might need to transform them before consumers can understand them... that's OK, look at these patterns.

Ubiquitous Language

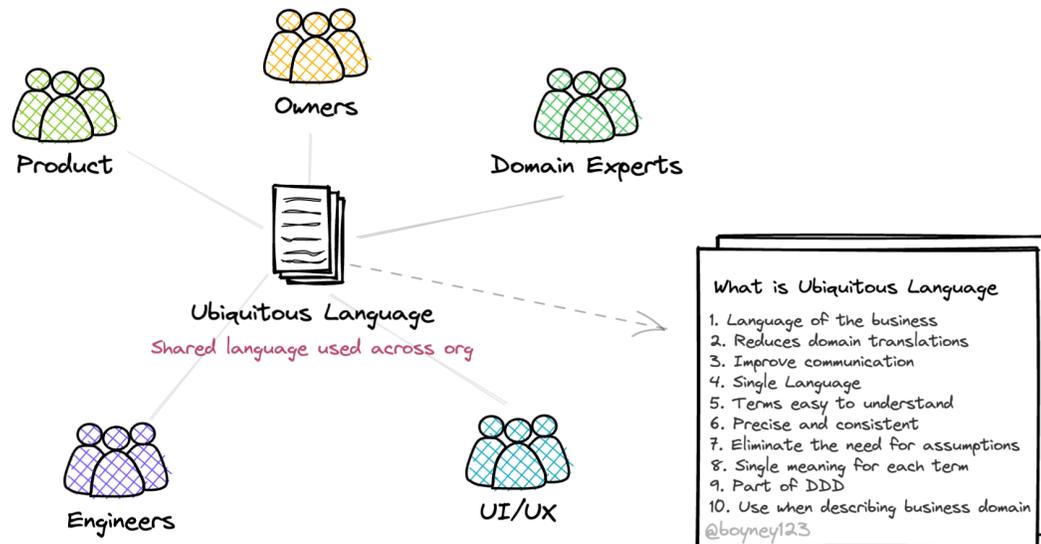


Figure 33: Ubiquitous Language

It can be time consuming or even frustrating when teams within an organisation are using terms to mean different things depending on the domain. For example a “policy” within one team could mean something completely different within another... This leads to mental translations of models/business terms and can lead to confusion and assumptions.

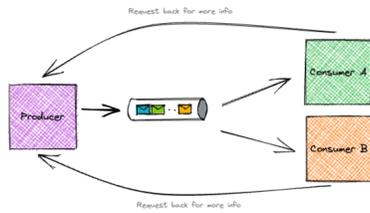
Having a “Ubiquitous Language” means parties that communicate between each other speak in the same language. Define standards for terms, share them with each other and use them to describe business domain.

This is not directly associated to “event-driven architectures” but domain driven design plays a huge part in EDA. Having a shared language can help you name events, boundaries and build your EDA applications.

Extra Resources

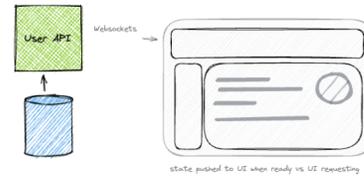
- Best practices to design your events in event-driven applications - Blog post by Martin Fowler on Ubiquitous Language.
- Domain Driven Design Book - Domain-Driven-Design book, where Eric Evans talks about Ubiquitous Language.
- Learning Domain Driven Design - Notes in this design taken from the book “Learning Domain Driven Design” by Vladik Khononov.

Understanding Eventual Consistency



Availability over consistency

If consumers don't have own version of data. They may request from producer, making callback pressure effecting the availability. So they store own version

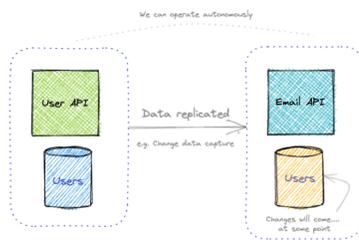


Watch out for customer UX

Does your UI have to be instant? Can you use bidirectional events to help?

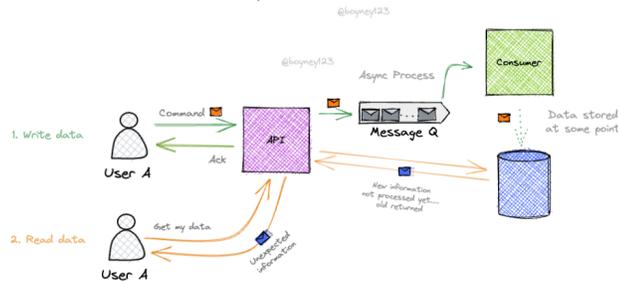
Understanding Eventual Consistency

What does it mean when data is eventually consistent?



State split across systems

Increases availability and service autonomy
Data replicated across boundaries
Latency introduced, data will be eventually consistent



Reading your own write problem

User makes command request to make changes
User then makes request to get data
Old information is returned, not updated yet.

Figure 34: Understanding Eventual Consistency

“Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.” [source]

Distributed state When we build distributed systems, there are times that state is distributed across our architecture (e.g. when we favour availability over consistency, e.g services have a copy of the data they are consuming vs requesting it from another service). This means data across your architecture in theory will be eventually consistent and at times the state will be inconsistent (as data is replicated across your architecture).

Users performing async actions Another common pattern is when users perform async operation. A command is sent to perform an action, and an async operation kicks in to process the action. The user then goes to read the information but it's not complete yet or returns old stale information, this means the task or data has not yet been replicated from where the user/api is reading it from.

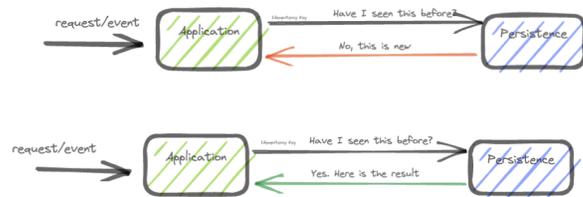
When data is inconsistent it can affect your users experience (users reading their own writes), so it's worth considering UX patterns you can introduce to help (e.g webhooks back to client). Also, it's worth asking yourself does your business mind if these tasks are async and eventually consistent? Like most things, trade-offs need to be considered.

When building EDA applications, you will naturally find yourself dealing with patterns and experiences that are eventually consistent, it's worth keeping that in mind and creating or exploring patterns to help if you need to manage it better.

Extra Resources

- CAP theorem - Interesting theory about consistency, availability and partition tolerance, and that distributed data store can only provide two of the following.
- Trade-offs of event driven - More information from AWS to tell us about the trade-offs when going event-driven, more information on Eventual consistency

Understanding Idempotency



Idempotency Keys

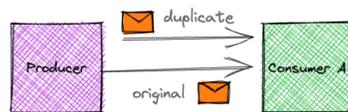
One solution to use hash/keys to understand if messages has been processed before

Understanding Idempotency

Given the same input we get the same output.

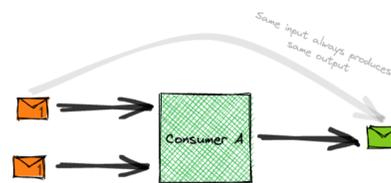
@boyney123

Avoiding side effects



At-least-once delivery

Commonly found with "at-least-once delivery"
Multiple attempts can be made to deliver message



Idempotent Consumer

Events can be retried and replayed
Avoid side effects in your consuming applications

Figure 35: Understanding Idempotency

When building distributed message/event-based architectures it's important to consider and think about idempotency.

Messages/events can be replayed to your consumers, this means you need to handle the same message being played more than once. This might happen if you have failures, network issues or maybe you need to replay a bunch of events, either way **thinking about idempotency up front can help**.

Implementing idempotent consumers can be easier in some consumers than others depending on the logic inside your applications, but when you design or write your applications just think **"What happens if this SAME event was to trigger again"**, what side effects will you have? Things will fail, helps being prepared for that.

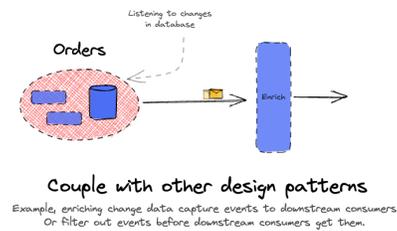
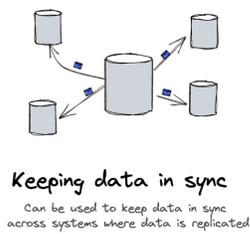
Idempotency keys Some patterns to solve this issue to store the fact that the event has been processed before using idempotency keys, this is where you hash the events or part of the event,

and use persistence storage to store the fact the events has been processed (maybe with the result of the initial process). There are tools and libraries out there to help you with this, but it's worth exploring.

Extra Resources

- Message Delivery - Will your events naturally get replayed for you? Could depend on your broker of choice and how you handle messages/events, worth understanding at-most-once, at-least-once and exactly-once delivery.
- Good and hart parts of EDA - Understanding idempotency can take awhile to wrap your head around if you are new to it, but there are also some things to consider when building EDA applications (Error Handling, Debugging).
- Idempotent Receiver - Summary from the book enterprise integration patterns.
- Open Source project Powertools Idempotency helpers - Using AWS Lambda? Powertools is a great open source project with many different utilities to help you build performant Lambda functions. The project also can utils to help you deal with idempotency, worth checking out.

Understanding change data capture



Understanding change data capture

Listening to data that has been changed in your database and taking action. @boyney123

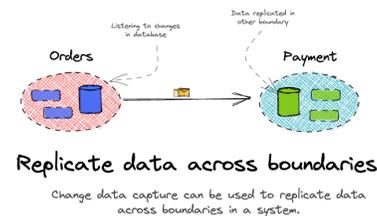
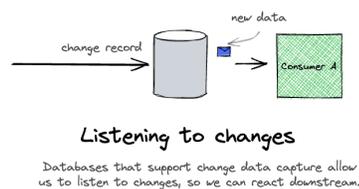


Figure 36: Understanding change data capture

Change data capture allows us to track changes made to a database and react.

We can observe changes written to a database and trigger downstream systems. This pattern is great for **event-driven architectures** as we can listen to changes and trigger downstream consumers or even replicate the data across our bounded context / microservices.

Listening to changes

- Change data capture allows us to listen to changes in a database.
- You can react to these changes to trigger downstream consumers or copy the data into other databases/views.
- It's quite a common pattern to listen to changes in a database, change data capture can help us write some resilient, and scalable event-driven architectures, raising events for internal use but also domain events for business consumption.
- Change data capture patterns can help us reduce the amount of custom code we need to write. E.g. Rather than raising events straight after a Database insert, we can insert and let services/databases give us the change data capture event.

- Great example of this is using DynamoDB streams. Data is streamed to consumers in near real-time.

Keeping data in sync

- Change data capture can be used to keep data in sync across many boundaries of your architecture.
- Many architectures have many data sources, change data capture allows us to listen to a change on the “leader” and push these changes to “followers”.

Couple with other design patterns

- When consuming change data capture events, you may want to add layers in between the system consuming it, to enrich or filter the data.
- Example: if your consumer of the event requires additional information you can use enricher pattern to do this.

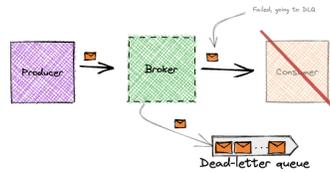
Replicate data across boundaries

- Your architecture may be split into many boundaries. You can use change data capture pattern to replicate data across boundaries making systems more resilient favouring availability over consistency.
- Replicating data may introduce eventual consistency, given no changes are made your database all listener’s will be eventually consistent. This means data changes across your architecture is async and can take time, it’s important to understand this.

Extra Resources

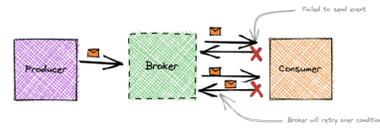
- Designing data-intensive applications - Book has a great insights into streaming and has a chapter around change data capture.
- Change data capture with DynamoDB - DynamoDB is a great NoSQL database that supports change data capture pattern. Many people are using this pattern to help scale their distributed architectures.
- Understanding bounded context - Your data may be replicated across your architecture, have you got the correct bounded context defined? Are you going back to producers to get information? Understanding bounded context can help.
- Migration with Change data capture - Maybe you are looking to migrate to an EDA architecture? Can you use change data capture pattern to help? Visual here to help you understand how you can use events to migrate.
- Unlocking value from your events - There is a huge amount of value in your events in your architecture, these include events from change data capture pattern. Can your business utilise these?
- Understand eventual consistency - If you are replicating data you may be favouring availability over consistency. It’s important to understand eventual consistency and what it means for your architecture.

Understanding event delivery failures



Dead-letter queues

Event fails to reach its target, broker can send event special queue for debugging/retry



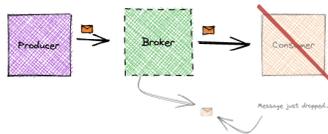
Redrive policies

Some brokers may redrive/retry event delivery for you. When events fail in given condition broker can retry

Understanding event delivery failures

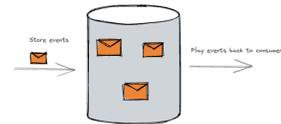
What options do we have when events fail to be delivered in our architecture?

@boyney123



Dropping events

Depending on your broker, some events may just be dropped if cannot reach targets. This may or may not be an issue for you



Archive / Replay

Store events (e.g. archive), and retry them (when needed) Make sure consumers are idempotent, avoid side effects

Figure 37: Understanding event delivery failures

When building event-driven applications it's good to understand what happens if your **event fails to reach consumers**, what your broker may do for you or what you need to manage yourself.

Dead-letter queue

- Place where events can go when failed to be delivered to consumer
- Events can fail to be delivered, do you want to store them? (before they are dropped)
- Use queue to debug errors or replay events back into system
- For more information you can read Dead letter channel enterprise integration pattern.

Redrive policies

- Some brokers may redrive (retry) events for you if they fail
- Example of failures may be due to network conditions or permissions
- Example of this would be Amazon EventBridge that retries event delivery for 24 hours with back off.

- Events could be dropped/lost (setup dead-letter queue to capture them)

Archive/Replay

- Store events that were raised and replay them into broker.
- Worth making sure consumers are idempotent as messages may be replayed more than once
- Outbox pattern can be used to store events before sending them.

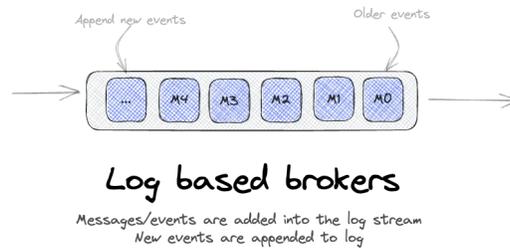
Dropping events

- Some event brokers may just drop the event if it cannot process it
- Many brokers will retry given conditions, but worth checking and handling failures

Extra Resources

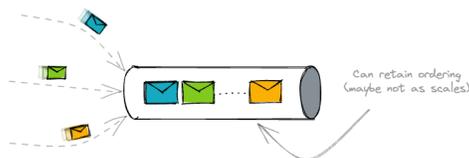
- Dead-letter channels - Interested to know more about dead-letter channels? Worth reading the enterprise integration pattern.
- Understanding Idempotency - Depending on your broker depends if events will get resent to consumers. Understanding idempotency can help.
- Things to consider when building EDA applications - Dead-letter queues / management is just one of them, there are more things to consider when building EDA applications, visual here to help.
- Message Delivery - What delivery method is your broker using? Understanding can help.
- Good and hard parts of EDA - What are the good and hard parts of building EDA applications? Visual for more information.

Understanding event streaming



Understanding event streaming

What is event streaming and when to use them? @boyney123



Processing real-time data

Use streams to collect events in real time
Consume streams for fast processing, e.g. real-time analytics



Flow of events over time

Data arrives over time, in unknown quantities
(e.g. tracking user interactions on website)
Streams can help process this information

Figure 38: Understanding event streaming

What is event streaming? Think of event streams as a flow of events traveling through a river. Each event is captured and can be processed by downstream consumers.

Event streams can be great for processing real-time data.

When we build applications there are times where the amount of data to process is unknown and we want to capture it (e.g. user interactions with a shopping cart online), we can collect this data and then process it in real-time downstream (e.g. for analytics or reporting for example).

Data loses value over time. Many companies want to process information in real-time to make decisions. Event streaming can help.

Event streaming use cases Some common examples where you may find event-streaming is handling IOT events, gaming, real-time applications or using streams to collect user information for analytics/metrics downstream.

You can also connect streams together to process information (Event stream processing), streams

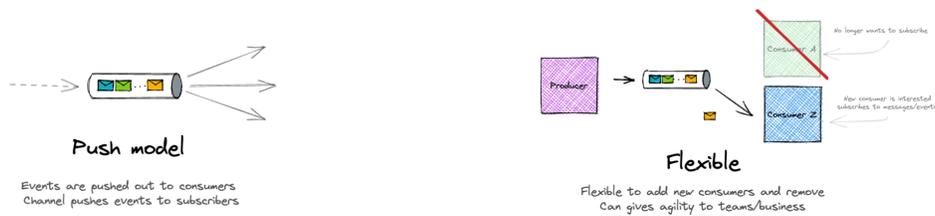
read output from other streams and can process this into new information.

Log based brokers Some brokers use a log-based approach to capture messages/events in an append only system. Think of this as a ledger, new information is added onto the stream, and consumers can consume this information from either the start or choose a starting location (usually an offset). Examples of log-based brokers are Amazon Kinesis Data Streams or Apache Kafka.

Extra Resources

- Designing data-intensive applications - Book has some great chapters on event streaming.
- Building an EDA application, streaming is just part of it - There are many other parts to building event-driven applications, streams are just part of it. This visual helps you identify others.

Understanding publish & subscribe messaging



Understanding Publish/Subscribe (Pub/Sub)

Publishing messages to many downstream subscribers/consumers

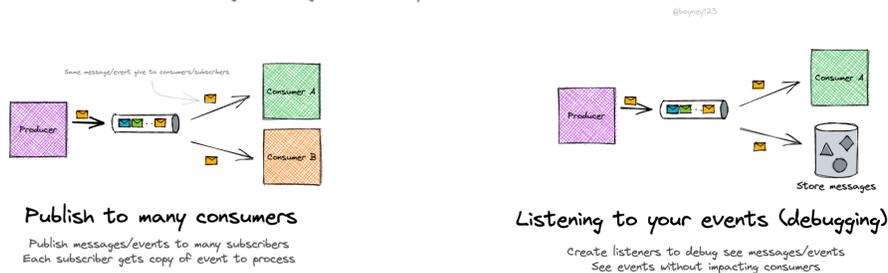


Figure 39: Understanding publish & subscribe messaging

Pub/Sub channels allow you to publish messages to many consumers that are interested in that message. Producer publishes and consumers subscribe.

Publish to many consumers This pattern allows your producer to publish messages to many downstream consumers. Consumers get their own copy of the message to process.

Listening to your events As consumers are independent from each other (own copy of the message), you can create new subscribers onto the channel and use this for debugging. An example of this would be to create a new subscriber to an event, store the information somewhere or log it out, this way you can see all events going through the producer's channel.

Flexible Consumers come and go, high chance your producer does not care. This gives teams the flexibility to add consumers as business requirements change or even remove them.

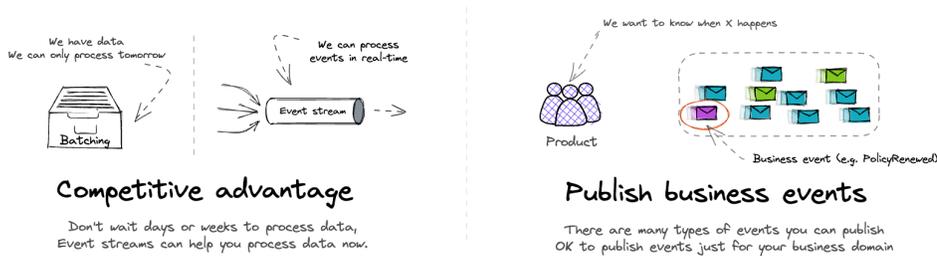
Push model With pub/sub messages/events are often pushed to downstream consumers (rather than consumers having to listen and pull events of a queue/channel).

Durable Subscriptions At times you might have subscribers that are still interested but “inactive” (no longer receiving messages), and gets the messages back when they reconnect, this is known as the durable subscriber, if you are interested it’s worth checking out that pattern.

Extra resources

- Publish-Subscribe Channel - Publish Subscribe channel from Enterprise Integration Patterns, worth a read to dive deeper.
- Durable Subscriber - Interesting pattern when subscribers are “inactive” and want to process messages they missed from a pub/sub point of view. Worth reading and understanding.
- Conway’s Law and EDA - When creating pub/sub patterns you will have many producers and consumers, understand how Conway’s law can effect all this with this visual.
- Message translator pattern - Before consuming events in your subscribers/consumers do you want to transform your data? Might be worthwhile? Dive deeper with this visual.
- Claim check pattern - When sending events, sometimes they can be too big for your broker, or you want to keep them small. Store the information with a key and let consumers get it back with the Claim check pattern.
- Content enricher pattern - Before consumers get your events can you implement some middle-ware to enrich them?
- Event Types - You will be sending messages/events throughout your application, know the difference between the types of events you are sending, your contract will impact your architecture choices.
- Message Brokers vs Event Brokers - Message Queues, Event Brokers what does it mean? Where does pub/sub fit into the picture of EDA. Visual to help you.

Unlocking value from your events



Unlocking value from your events

How can event-driven architectures give business opportunities?

@boyney123

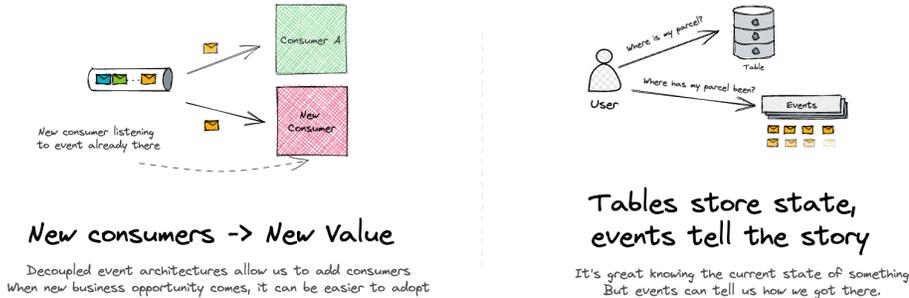


Figure 40: Unlocking value from your events

When we build event-driven architectures we use events to communicate between services and domains, these events are often used to trigger async processes downstream, and we can use event-driven architectures to create decoupled and scalable solutions.

It's also important to know that the events we use to communicate can also be **valuable for businesses** and create new opportunities (e.g. Using events to process data in real-time).

Competitive advantage Many companies rely on batching to process data and generate reports **after events occur**, event-driven architectures can give organisations options to process information **when events occur**. For example, companies are using event streaming to process events in real-time for downstream consumers (e.g. Reporting). If we move data processing closer to the time it occurs, we can get more value from it.

Publishing business events There are many types of events when building event-driven architectures, and business events are part of that. We use events to communicate between systems, but we can also use events to notify parties of critical business metrics we can use to make decisions.

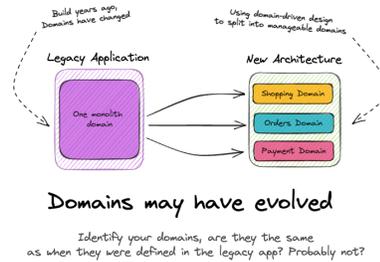
New Consumers = New Value Producers are decoupled from consumers, when new business opportunities arise we can add consumers onto events already there. Today you may not see business value in certain events, in the future the value may unlock, having this producer/consumer relationship can help.

Tables store state, events tell the story Many applications query some database for the state of a given entity, for example a query to find the location of a package right now for a given user. This is great, but how we can determine where the parcel has been? We can use events to tell the story. Keeping an event log can help us understand how things ended up the way they are.

Extra Resources

- Designing data-intensive applications - Book has some great chapters on event streaming.
- Types of events - There are many types of events, understanding them can help.
- Learning Domain Driven Design - Some notes taken from the book “Learning Domain Driven Design” by Vladik Khononov.

Using events to migrate from legacy architectures



Using events to migrate from legacy architectures

What is event-driven migrations? How can we use events to help us migrate.

@boyney123

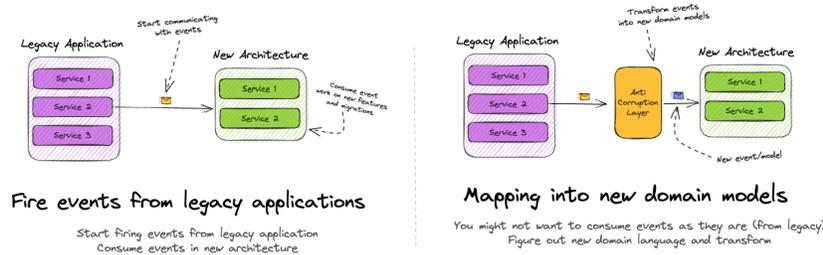


Figure 41: Using events to migrate from legacy architectures

When we build event-driven architectures we use events to communicate between services and domains, these events are often used to trigger async processes downstream, and we can use event-driven architectures to create decoupled and scalable solutions.

It's also important to know that the events we use to communicate can also be **valuable for businesses** and create new opportunities (e.g. Using events to process data in real-time).

Rather than going big bang migrations that could take months or years, slice up your architecture/features and delivery them over time (commonly seen in strangler pattern). We can use events here to help us, and move us towards an event-driven architecture.

Fire events from legacy applications

- Integrate your legacy application with your event/message broker
- Requires up front cost, but allows you to start publishing events from application
- Downstream consumers (new architecture) can consume events
- Cloud example pattern could be using `PutEvents` from Amazon EventBridge to publish events from legacy application into AWS, where new consumers can react.

Mapping into new domain models

Visuals and thoughts by David Boyne (@boyney123)

- Careful not to take “legacy events” as they are, you may want to map them into a new format/domain for your architecture
- Make sure legacy implementation details in events are not integrated within your new architecture (unless you identify that the models are similar)
- Use context mappings to map events from a legacy language into a new one. You have options here: Conformist pattern, anti-corruption Layer, open-host service.

Domains may have evolved

- Your legacy application may have been written years ago, things change. Your models/domains may have changed from inception years ago.
- Use techniques like Event Storming to highlight existing behaviours of your system and new domains you want to evolve into.
- You don't have to mimic the same language in your new architecture. You may want to start using bounded contexts and ubiquitous language away from your legacy application (if it makes sense)

Extra Resources

- Minimum Variable Migrations - Some of these ideas were inspired by the work Ben Ellerby is doing with Minimum Variable Migrations.
- Find events in your existing application - EventStorming can help you identify events for new applications, but also systems that already exist, this can help find events in your legacy application.
- To conform or not conform to legacy events? - Careful taking events from legacy applications as they are (conformist pattern), you may want to map. This visual can help

Why use message brokers?

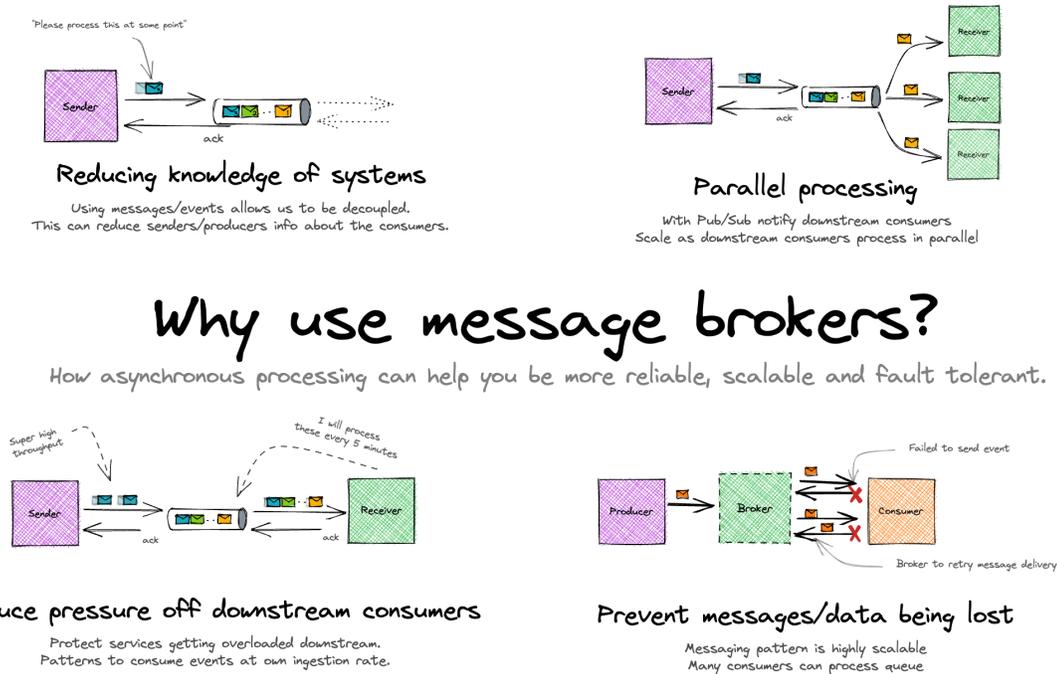


Figure 42: Why use message brokers?

Reduce pressure off downstream consumers

- Many people use message brokers to reduce back pressure from downstream services. The sender puts messages on the queue, and consumers can pick up these messages and process them in the batch size they want and the time they like.
- You can use message brokers to make sure that downstream services do not get too overloaded, which can improve system reliability.
- Example of this could be to use an SQS queue, to handle many messages, and a consumer to process these messages and delivery information to a third-party API.

Prevent messages/data being lost

- Many brokers offer the ability to retry messages/events if they fail to get processed or delivered. This can help prevent any information being lost.
- If your broker does offer retry and replay, you want to consider idempotency as your consumer may be triggered more than once, and you don't want any unwanted side effects.

- This could be a great pattern, as you broker may do the heavy lifting for you (retry/reprocess), many are configurable and if all fails you can decide what you want to do with the messages after the retry period (example drop them or store them for later processing)

Parallel processing

- Some brokers offer pub/sub patterns allowing you to send events and notify downstream consumers. This pattern allows you to notify downstream processes that something has happened. Many downstream systems can listen to these events.
- If you are using message queues, you can still have many consumers processing messages from the queue, this allows you to scale processing downstream.

Reducing knowledge of systems

- Using messages/events to communicate between systems/services can help us decouple our architecture.
- Producers may not need to know about downstream consumers, this gives the producers the ability to isolated and focus within its own domain/boundary.

Extra Resources

- Designing data-intensive applications - Book has some great insights into streaming, and messaging. Highly recommended if you want to dive deep.
- Fundamentals of Software Architecture - A great book to learn more about the fundamentals of architecture, goes into event architecture and many patterns.
- 65 messaging patterns - Great resource to dive deeper into many different patterns. Highly recommended.

Learn event-driven architecture today

If you want to learn more and get started building event-driven architectures on AWS, here are a few resources that can help you.

EDA Guide on Serverlessland.com Serverlessland.com hosts over 400 AWS Serverless patterns, 60 Step Functions workflow patterns and over 40 serverless code snippets, we also have a new EDA section to help the community learn more about building event-driven architectures. If you want to know more you can visit <https://serverlessland.com/event-driven-architecture/intro>

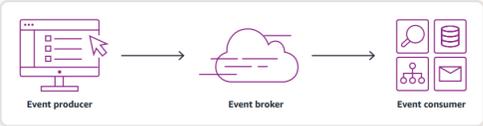
Introduction to Event Driven Architecture

What are Event Driven Architectures ?

Event-driven architectures are an architecture style that uses events and asynchronous communication to loosely couple an application's components. Event-driven architectures can help you boost agility and build reliable, scalable applications.

Serverless services like [Amazon EventBridge](#), [AWS Step Functions](#), [Amazon SQS](#), [Amazon SNS](#), and [AWS Lambda](#) have a natural affinity with event-driven architectures - they are invoked by events, emit events, and have built-in capabilities for building with events

[Start Learning →](#)



This guide introduces you to event-driven architectures. Understand the patterns within event-driven architectures and the AWS services that are commonly used to implement them. Learn best practices for building event-driven architectures, from designing event schemas to handling idempotency. Find getting started resources for building event-driven architectures on AWS.

Figure 43: Learn event-driven architecture on Serverlessland.com

30 Days of Serverless and EDA For those who want to get hands on building serverless event-driven-architectures on AWS, we have created a new program for 30 days of Serverless. These are a collection of videos we have put together to help you learn serverless and event-driven-architectures from the ground up. If you want to learn more you can visit <https://serverlessland.com/reinvent2022/30dayssls>

30 days of Serverless

30 days of Serverless

Learn how to build websites and event-driven applications using the different AWS serverless services.

According to [ChatGPT](#) these are the 3 tips to get started with Serverless in AWS today.

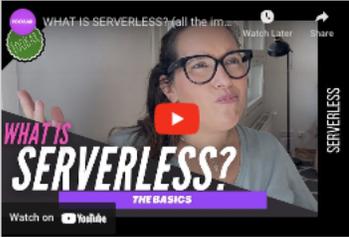
1. Research the different serverless services available in AWS. This will help you decide which service is best for your specific needs.
2. Familiarize yourself with the serverless architecture and how it differs from more traditional server-based approaches.
3. Monitor your serverless applications to ensure they are performing as expected and to quickly identify any potential issues.

This learning plan will help you with those!

Day 1 - What is Serverless?

Objective 1 - Learn the basic concepts

What is serverless? What does it mean? What benefits it has? Why use it? How to get started with serverless?



Day 2 - AWS Lambda 101

Objective 1 - Learn the basic concepts

This video gives a high-level overview of AWS Lambda. It talks about what a Lambda function is and when you should use it.



Day 3 - Amazon API Gateway 101

Objective 1 - Learn the basic concepts

A high-level overview of Amazon API Gateway, what an API is and how API Gateway handles many of the API tasks we often manage in code.



Day 4 - Amazon DynamoDB 101

Objective 1 - Learn the basic concepts

This video gives a high-level overview of Amazon DynamoDB and why it is purpose-built for EDA applications built on serverless.



Figure 44: Learn event-driven architecture on Serverlessland.com

Summary

EDA visuals are small bite sized visuals about event-driven architectures. You can use the visuals to help you get a high level overview of areas of event-driven architectures, and use the resources to dive deeper.

The visuals in this document are from <https://serverlessland.com/event-driven-architecture/visuals> and this document will be generated again with every new visual added.

Hopefully you find this content useful, and feel free to connect with me if you want to learn more.

- Twitter: <https://twitter.com/boyney123>
- Mastodon: [@boyney123@hachyderm.io](https://hachyderm.io/@boyney123)
- LinkedIn: <https://www.linkedin.com/in/david-boyne/>
- GitHub: <https://github.com/boyney123>