Cloud Native Approach

Lev Epshtein
lev@opsguru.io

About me

Lev Epshtein

Technology enthusiast with 15 years of industry experience in DevOps and IT. Industry experience with back-end architecture.

Solutions architect with experience in big scale systems hosted on AWS/GCP, end-to-end DevOps automation process.

Cloud DevOps, and Big Data instructor at NAYA and JB.

Certified GCP trainer,

Partner & Solution Architect Consultant at Opsguru.

lev@opsguru.io

Cloud Native

Cloud Native Definition from CNCF

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Cloud-Native characteristics

- Microservices
- Health reporting
- Telemetry data
- Resiliency
- Declarative, not reactive

Microservices

One of the best ways to fight **complexity** is to separate clearly defined functionality into **smaller services** and let **each service independently iterate**. This **increases the application's agility** by allowing portions of it to be changed more easily as needed. Each microservice **can be managed by separate teams**, **written in appropriate languages**, and be **independently scaled as needed.**

Health Reporting

To **increase the operability of cloud native applications**, applications should **expose a health check**. Developers can **implement this as a command or process signal** that the application can respond to after performing self-checks, or, more commonly, as a web endpoint provided by the application that returns **health status via an HTTP code.**

Moving health responsibilities into the application makes the application much easier to **manage and automate**.

Telemetry Data

Telemetry data is the information necessary for making decisions. It’s true that

telemetry data can overlap somewhat with health reporting, but they serve different purposes. **Health reporting informs us of application life cycle state, while telemetry data informs us of application business objectives.**

The metrics you measure are sometimes called *service-level indicators* (**SLIs**) or *key performance indicators* (**KPIs**). These are application-specific data that allow you to make sure the performance of applications is within a *service-level objective* (**SLO**).

Telemetry Data

Telemetry and metrics are used to solve questions such as:

- How many requests per minute does the application receive?
- Are there any errors?
- What is the application latency?
- How long does it take to place an order?

It is probably best to, at minimum, implement the **RED** method for metrics, which collects rate, errors, and duration from the application.

- Rate - How many requests received
- Errors - How many errors from the application
- Duration - How long to receive a response

Resiliency

There are two main aspects to **resiliency** we will consider with cloud native application: **design for failure**, and **graceful degradation.**

Design for Failure

The only systems that should never fail are those that keep you alive (e.g., heart implants, and brakes). If your services never go down, you are spending too much time engineering them to **resist failure** and not enough time **adding business** value. Your SLO determines how much uptime is needed for a service. Any resources you spend to engineer uptime that exceeds the SLO are wasted.

Two values you should measure for every service should be your your *mean time between failures* (MTBF) and *mean time to recovery* (MTTR). Monitoring and metrics allow you to detect if you are meeting your SLOs, but the platform where the applications run is key to keeping your MTBF high and your MTTR low.

Graceful degradation

The *Site Reliability Engineering* book describes graceful degradation in applications as offering “responses that are not as accurate as or that contain less data than normal responses, but that are easier to compute” when under excessive load.

The point of graceful degradation is to allow applications to always return an answer to a request.

Cloud Native Development

12 FACTOR APP & CONTAINER DESIGN

https://12factor.net/

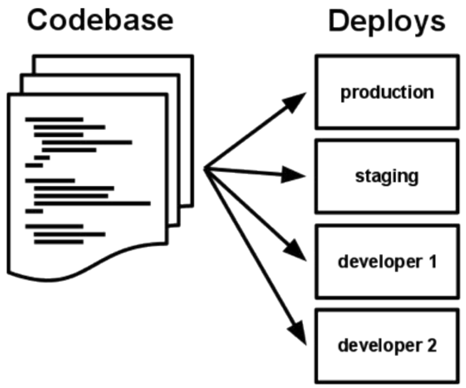
12-Factor Applications

Heroku was one of the early pioneers who offered a publicly consumable **PaaS**. Through many years of expanding its own platform, the company was able to identify patterns that helped applications run better in their environment. There are **12 main factors** that Heroku defines that a developer should try to implement.

The 12 factors are about **making developers efficient** by **separating code logic from data**; **automating as much as possible**; **having distinct build, ship, and run stages**; and **declaring all the application's dependencies**.

Factor 1: Codebase

Build on top of one codebase, fully tracked by a Version Control System (VCS). Deployments should be automatic, so everything can run in different environments without work. You should always have one repository for an individual application to ease CI/CD pipelines.



Factor 2: Dependencies

Do not copy any dependencies to the project codebase (Please don't...),

instead use a package manager. Always remember to use the correct versions of dependencies so that all environments are in sync and reproduce the same behavior.

For applications that are modularized and depend on other components, such as an HTTP service and a log fetcher, Kubernetes provides a way to combine all of these pieces into a single Pod, for an environment that encapsulates those pieces appropriately - SideCars [TBD]

Factor 3: Config

Store the config in Environment Variable (NOT FILES PER ENV) and Passwords in Vaults and KMS. **There should be a strict separation between config and code.** The code should remain the same irrespective of where the application is being deployed, but configurations can vary.

Factor 4: Backing Services - Critical

Treat backing services (database, caching, and so on) as attachable resources.

We should be able to swap **our database instance <u>without code changes.</u>**

One must be able to easily swap the backing service from one provider to another without code changes. This will ensure good portability and help maintain your system.

Factor 4: Continue - Backing Services - Critical

In a 12 Factor app, any services that are not part of the core application, such as **databases, external storage, or message queues, should be accessed as a service — via an HTTP or similar request** — and specified in the configuration, so that the source of the service can be changed without affecting the core code of the application.

For example, if our application uses a message queuing system, we should be able to change from RabbitMQ to ZeroMQ (or ActiveMQ or even Kafka) without having to change anything but configuration information.

Factor 5: Build, Run, Release

A twelve-factor application requires a strict separation between Build, Release and Run stages. Every release should always have a unique release ID and **releases should allow rollback natively.** Automation and maintaining the system should be as easy as possible.

Factor 6: Stateless Processes

You should not be introducing state into your services, applications should execute as a single, stateless process. The Twelve-factor processes are stateless and share-nothing. This factor lies at the **core of microservices architecture.**

Factor 7: Port Binding

The twelve-factor app is completely self-contained that runs on its own process and not using proxies such apache or tomcats.

Factor 8: Concurrency

When one writing a twelve-factor app, make sure that you’re designing it to be scaled out, rather than scaled up. That means that in order to add more capacity, you should be able to add more instances rather than more memory or CPU to the machine on which the app is running. Note that this specifically means being able to start additional processes on additional machines, which is, fortunately, a key capability of Kubernetes.

Factor 9: Disposability

It seems like this principle was tailor made for containers and Kubernetes-based applications. The idea that processes **should be disposable means that at any time, an application can die and <u>the user won’t be affected</u>**, either because there are others to take its place, because it’ll start right up again, or both.

Processes should be less time-consuming. Make sure you can run and stop fast. And that you can handle failure. Without this, automatic scaling and ease of deployment, development- are being diminished.

Support Signals

Factor 9: Disposability - Continue

A 12-factor app’s processes can be started or stopped (with a **SIGTERM**) anytime. Thus, minimizing startup time and gracefully shutting down is very important.

For example, when a web service receives a **SIGTERM**, it should stop listening on the HTTP port, allow in-flight requests to finish, and then exit. Similar, processes should be robust against sudden death; for example, worker processes should use a robust queuing backend.

Supporting this should be fairly easy by implementing something like:

```
import signal
signal.signal(signal.SIGTERM, lambda *args: server.stop(timeout=60))
```

Factor 10: Dev-Prod Parity

Keep development, staging, and production as similar as possible so anyone can understand it and release it. Continuous deployment needs continuous integration based on matching environments to limit deviation and errors. This also implicitly encourages a DevOps culture where Software Development and Operations are unified. Containerization is a huge help here and K8S for making every environment from Dev to production the same.

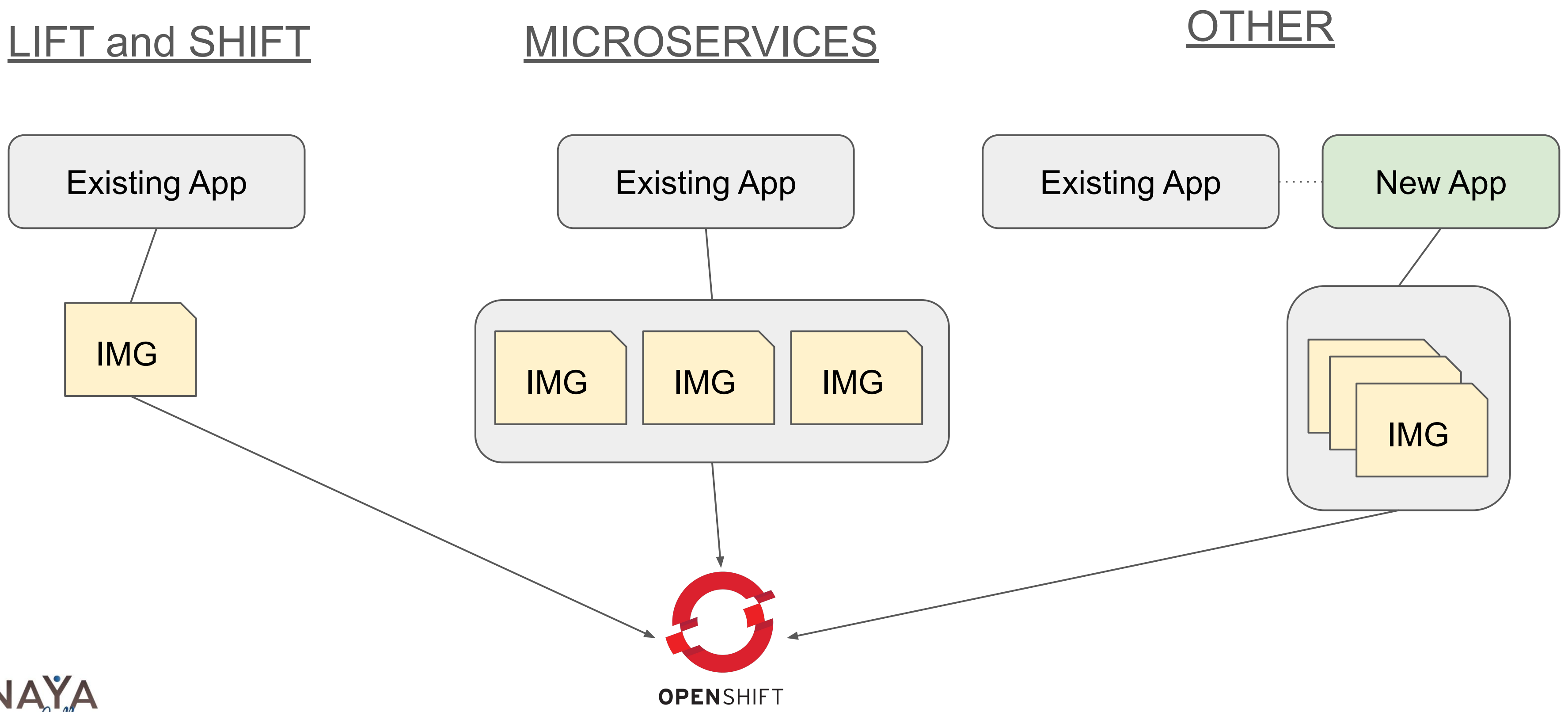
Factor 11: Logs

While most traditional applications store log information in a file, the Twelve-Factor app directs it, instead, to stdout as a stream of events in **JSON and not clear text**; it’s the execution environment that’s responsible for collecting those events. That might be as simple as redirecting stdout to a file, but in most cases it involves using a log router such as Fluentd and saving the logs to ES.

Factor 12: Admin Processes

Run admin/management tasks as one-off processes—tasks like database migration or executing one-off scripts in the environment or as an INIT Containers in K8S [TBD].

Cloud Native Development Strategy


LIFT and SHIFT
MICROSERVICES
OTHER
Existing App
IMG
Existing App
IMG
IMG
IMG
Existing App
New App
IMG
OPENSHIFT

Monolith to MSA

Refactoring a monolith to microservices

Truly greenfield development of microservices-based applications is relatively rare. Many organizations that want to adopt microservices already have a monolithic application. The recommended approach is to use the **Strangler application pattern** and incrementally migrate function from the monolith into services.

Refactoring strategies

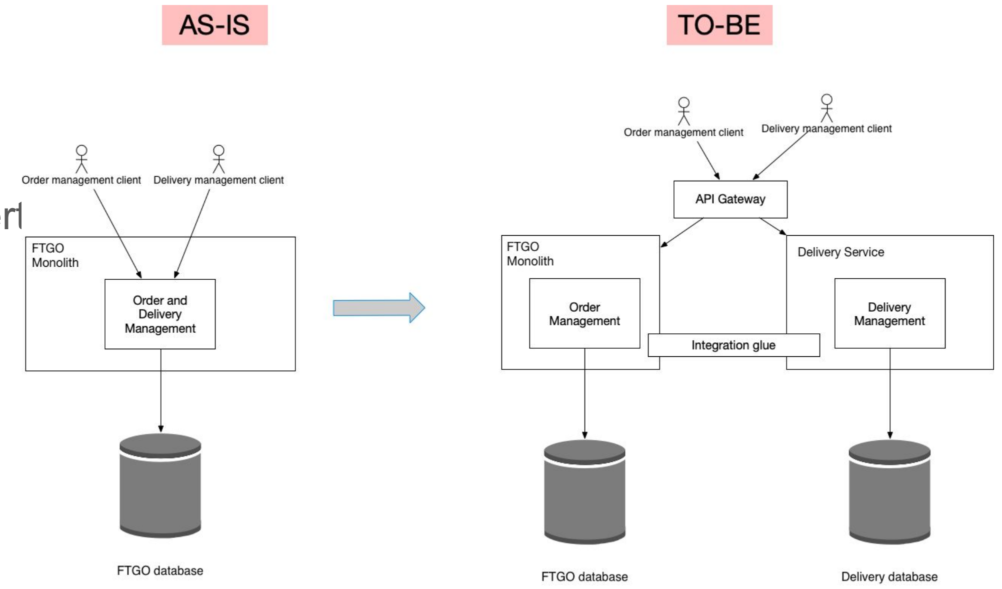
- Implement new functionality as services
- Extract services from the monolith

Implement new functionality as services

A good way to begin the migration to microservices is to implement significant new functionality as services. This is sometimes easier than breaking apart of the monolith. It also demonstrates to the business that using microservices significantly accelerates software delivery.

Extract services from the monolith

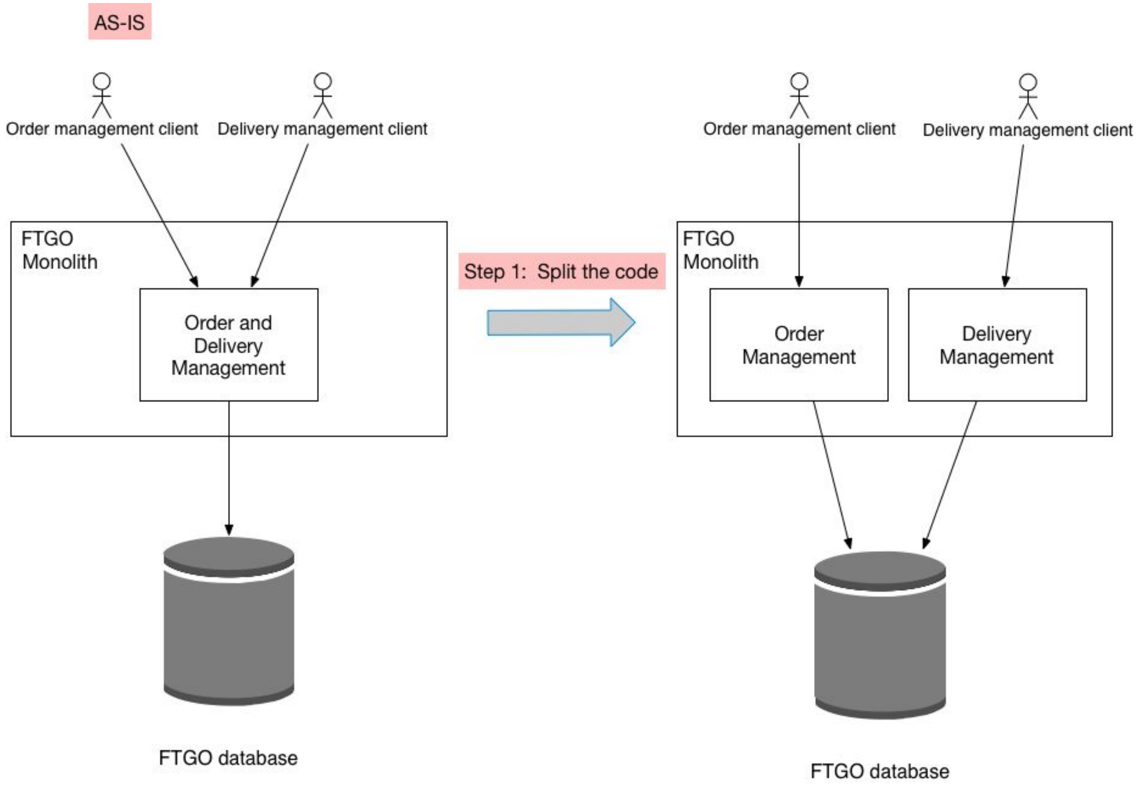
While implementing new functionality as services is extremely useful, the only way of eliminating the monolith is to incrementally extract modules out of the monolith and convert them into services.



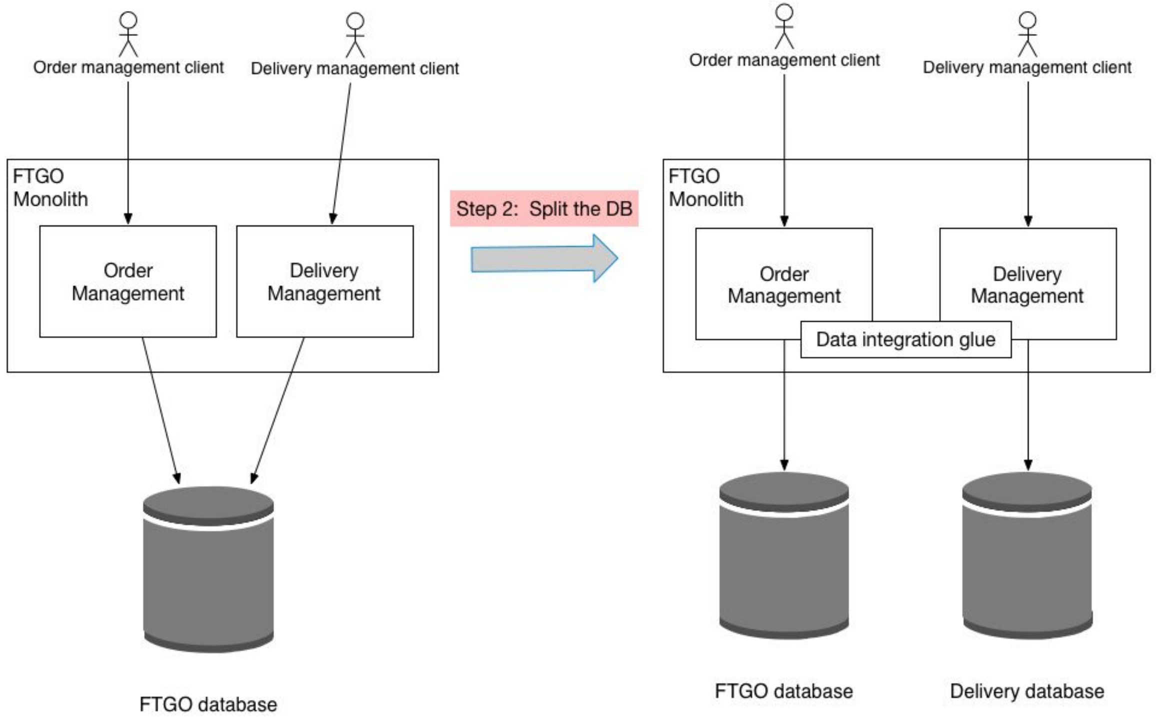
Extracting the Delivery Service consists of the following the steps:

- Split the code and convert delivery management into a separate, loosely coupled module within the monolith
- Split the database and define a separate schema for delivery management.
- Define a standalone **Delivery Service**
- Use the standalone **Delivery Service**
- Remove the old and now unused delivery management functionality from the FTGO monolith

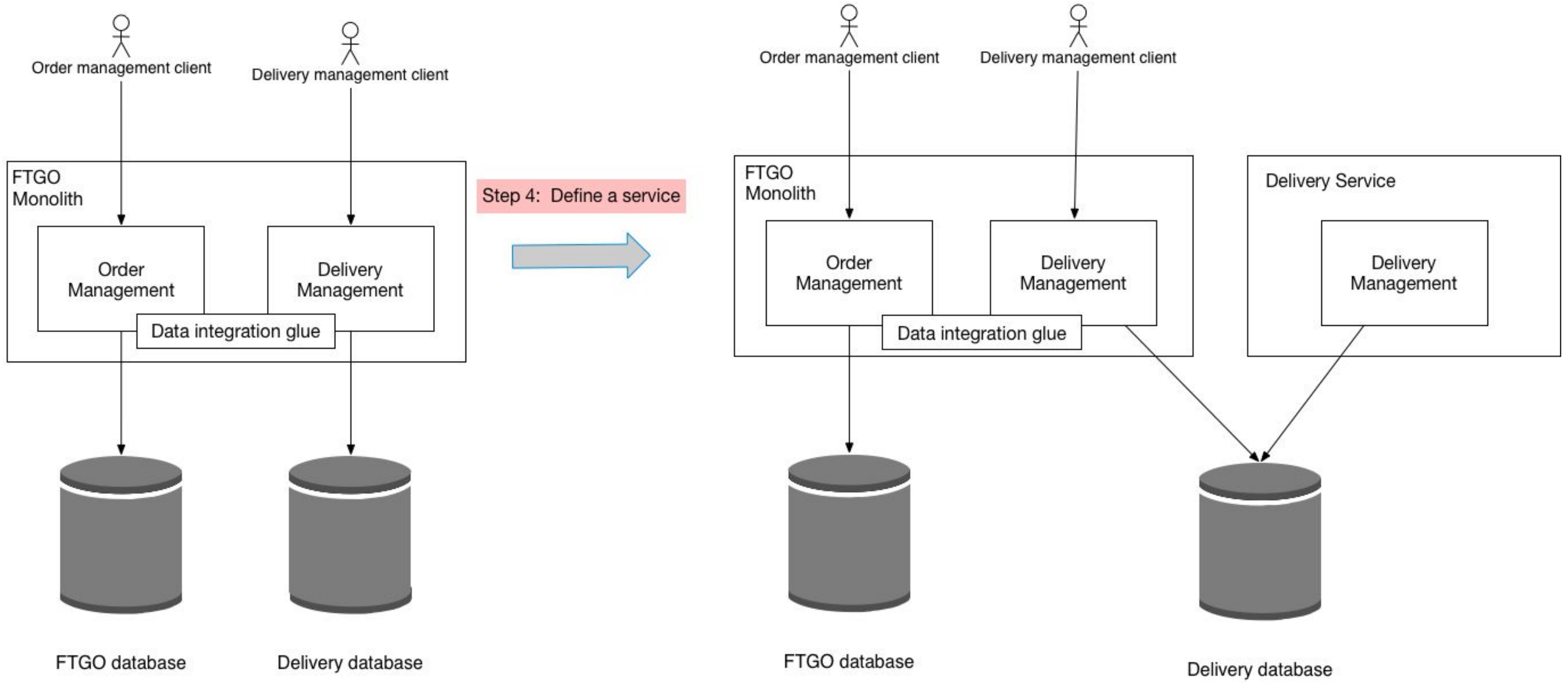
Step 1: Split the code



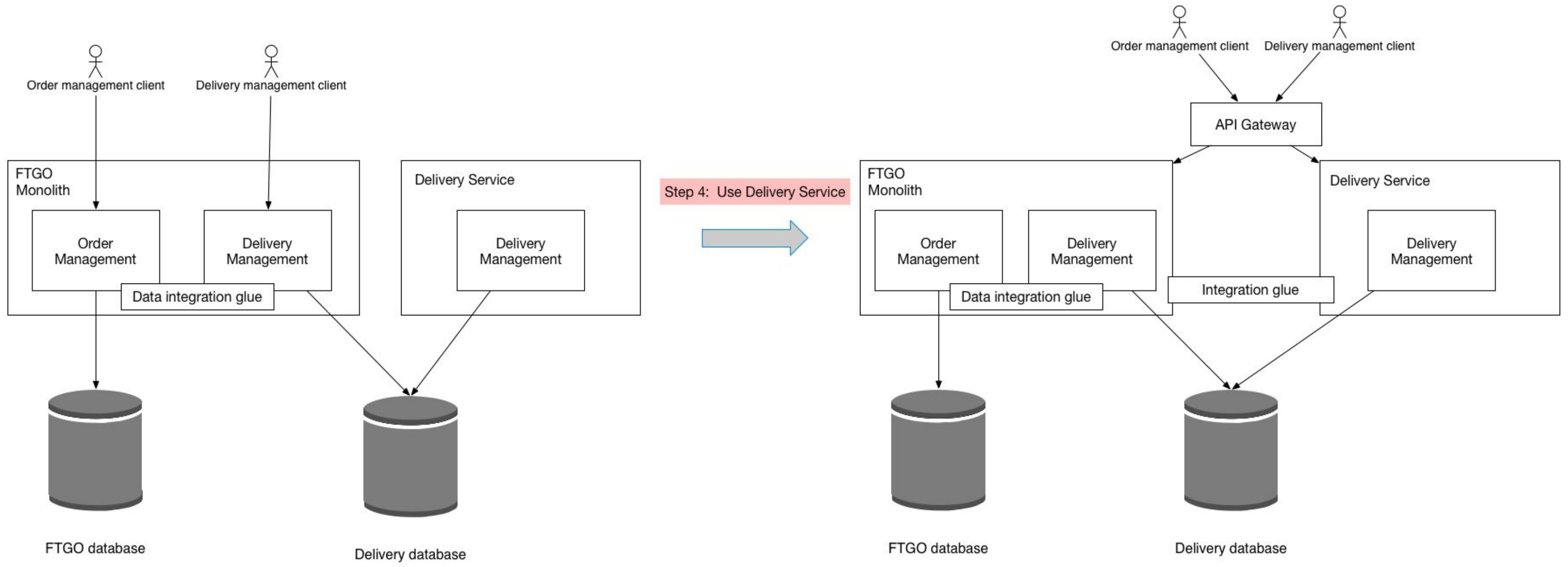
Step 2: Split the database



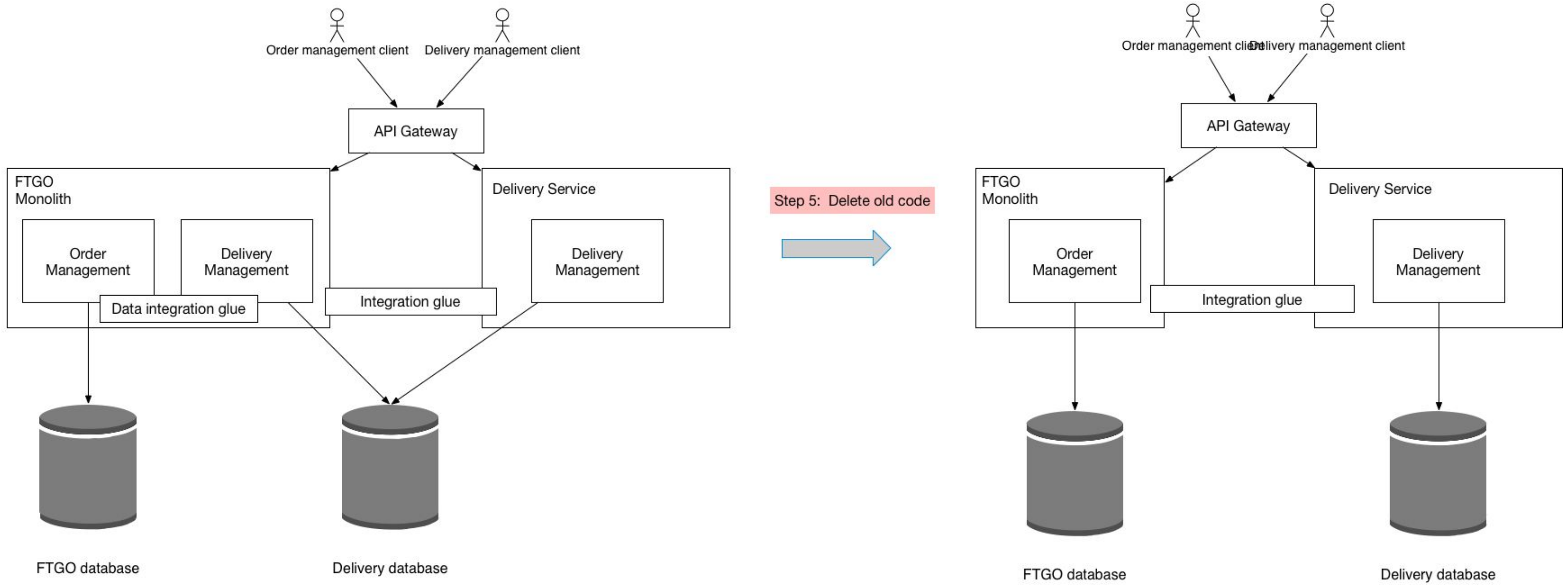
Step 3: Define a standalone Delivery Service



Step 4: Use the standalone Delivery Service



Step 5: Remove the delivery management functionality



Architecting Cloud-Native Serverless Solutions

Serverless and FaaS

When we say serverless, what we are usually referring to is an application that's built on top of a serverless platform. Serverless started as a new cloud service delivery model where everything except the code is abstracted away from the application developer.

Key Components

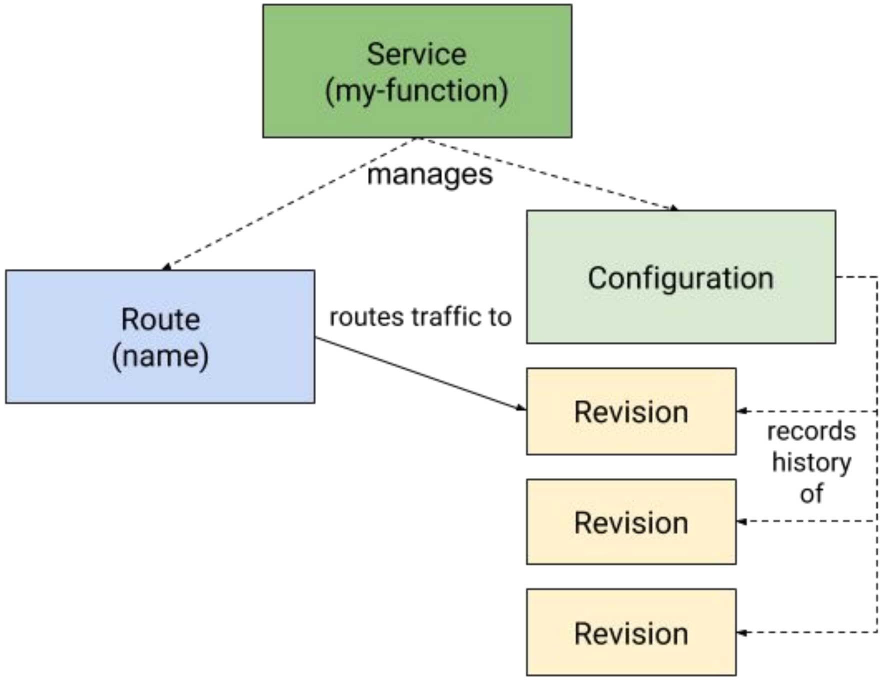
Functions as a Service (FaaS): Break down applications into smaller, independent functions. Platforms like AWS Lambda, Google Cloud Functions, Azure Functions or **Knative on Openshift** open enable execution without server provisioning.

Knative

Knative is a platform-agnostic solution for running serverless deployments.

Knative Serving

Knative Serving defines a set of objects as Kubernetes Custom Resource Definitions (CRDs). These resources are used to define and control how your serverless workload behaves on the cluster.



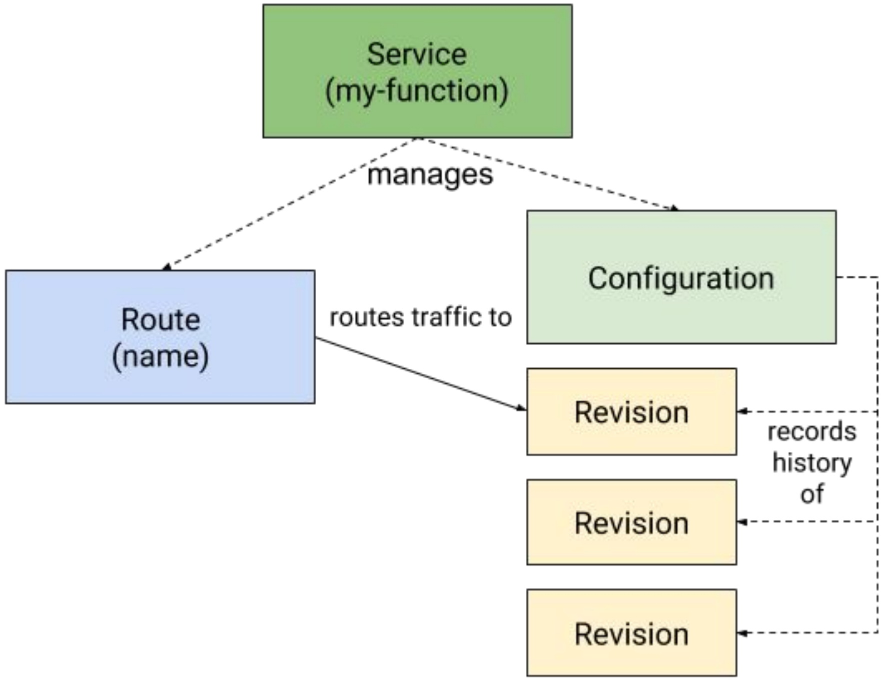
Knative

Services:

The service.serving.knative.dev resource automatically manages the whole lifecycle of your workload. It controls the creation of other objects to ensure that your app has a route, a configuration, and a new revision for each update of the service. Service can be defined to always route traffic to the latest revision or to a pinned revision.

Routes:

The route.serving.knative.dev resource maps a network endpoint to one or more revisions. You can manage the traffic in several ways, including fractional traffic and named routes.



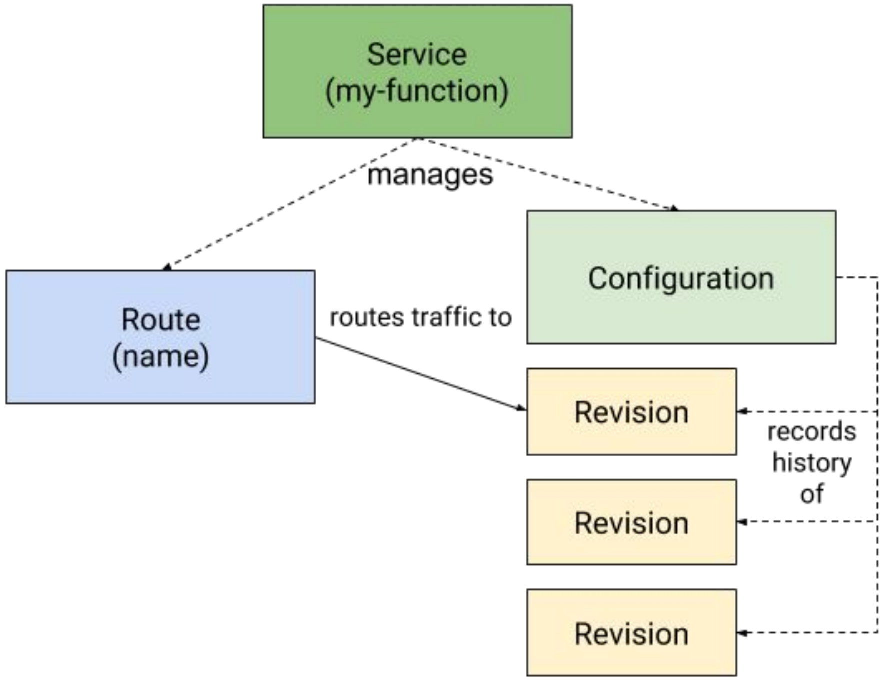
Knative

Configurations:

The configuration.serving.knative.dev resource maintains the desired state for your deployment. It provides a clean separation between code and configuration and follows the Twelve-Factor App methodology. Modifying a configuration creates a new revision.

Revisions:

The revision.serving.knative.dev resource is a point-in-time snapshot of the code and configuration for each modification made to the workload. Revisions are immutable objects and can be retained for as long as useful. Knative Serving Revisions can be automatically scaled up and down according to incoming traffic.



Knative Eventing

Knative Eventing is a collection of APIs that enable you to use an **event-driven architecture** with your applications. You can use these APIs to create components that route events from event producers (known as sources) to event consumers (known as sinks) that receive events. Sinks can also be configured to respond to HTTP requests by sending a response event.

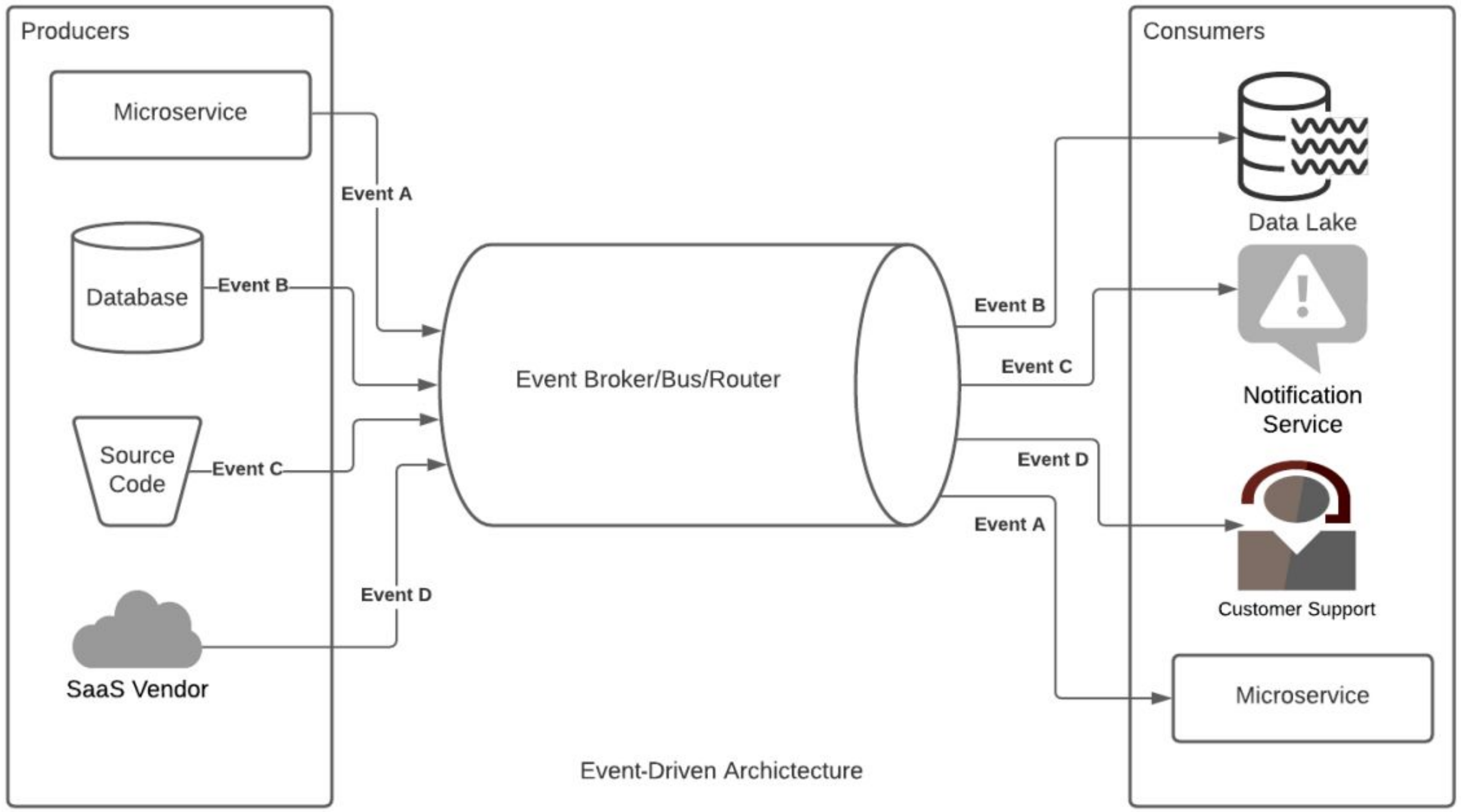
Event-driven architecture

EDA is an architectural pattern where capturing, processing, and storing events is the central theme. This allows a bunch of microservices to exchange and process information asynchronously.

Events

An event is the record of a significant occurrence or change that's been made to the state of a system. The source of the event could be a change in the hardware or software system. An event could also be a change to the content of a data item or a change in the state of a business transaction. Some examples of events are as follows:

- Customer requests
- Change of balance in a bank account
- A food delivery order being placed
- A user being added to a server
- Sensor reading from a hardware or IoT device
- A security breach in a system


Producers
Microservice
Database
Source Code
SaaS Vendor
Event A
Event B
Event C
Event D
Event Broker/Bus/Router
Event B
Event C
Event D
Event A
Consumers
Data Lake
Notification Service
Customer Support
Microservice
Event-Driven Archictecture

Benefits of EDA

- Improved scalability and fault tolerance due to a producer or consumer failing doesn't impact the rest of the systems.
- Real-time data processing for better decisions and customer experience – businesses can respond in real time to changes in customer behavior and make decisions or share data that improves the quality of the service.
- Operational stability and agility.
- Cost efficiency compared to batch processing. With batch processing, large volumes of data had to be stored and processed in batches.
- Better interoperability between independent services.
- High throughput and low latency
- Easy to filter and transform events.
- The rate of production and consumption doesn't have to match.
- Works with small as well as complex applications.

Use cases

EDA has a very varied set of use cases; some examples are as follows:

- Real-time monitoring and alerting based on the events in a software system
- Website activity tracking
- Real-time trend analysis and decision making
- Fraud detection
- Data replication between similar and different applications
- Integration with external vendors and services

Disadvantages

- The decoupled nature of events can also make it difficult to debug or trace back the issues with events.
- The reliability of the system depends on the reliability of the broker. Ideally, the broker should be either a cloud service or a self-hosted distributed system with a high degree of reliability.
- Consumer patterns can make it difficult to do efficient capacity planning. If many of the consumers are services that wake up only at a defined interval and process the events, this could create an imbalance in the capacity for that period.
- There is no single standard in implementing brokers – knowing the guarantees that are provided by the broker is important. Architectural choices such as whether it provides a strong guarantee of ordering or the promise of no duplicate events should be figured out early in the design, and the producers and consumers should be designed accordingly.