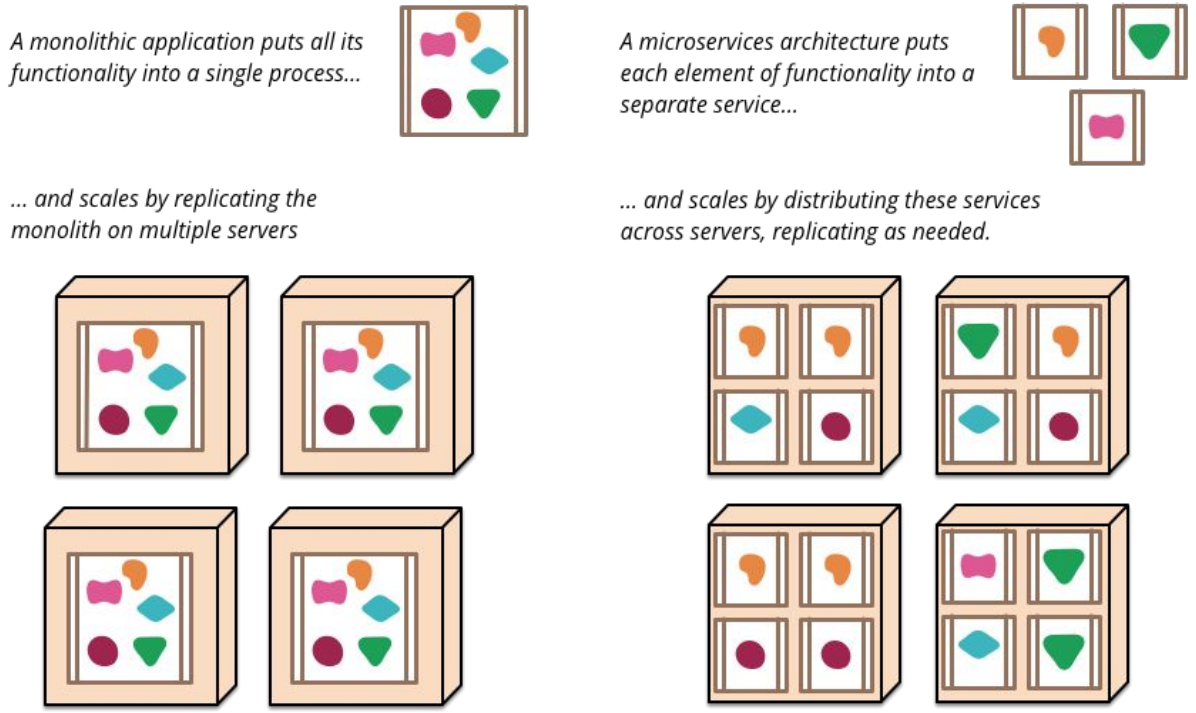
Microservices

Microservices

Microservices are an **application architecture style** where independent, self-contained programs with a single purpose each can communicate with each other over a network. Typically, these microservices are able to be deployed independently because they have strong separation of responsibilities via a well-defined specification with significant backwards compatibility to avoid sudden dependency breakage.



Example of a Sample Microservice

Let's take a scenario where you are creating an e-commerce store that uses Microservices pattern.

For a typical product say an iPhone on an e-com store, the details page displays:

- Basic Information about the product
- Your Purchase History
- People who bought iPhone, also bought cases
- Deals & Discounts associated with the iPhone
- Merchant Data
- Shipping Options
- Product Testimonials and so on and so-forth.

Additionally, a sample product details page will have multiple versions of the interface to cater to web, mobile and a REST API for use by 3rd party applications.

Example of a Sample Microservice

In a microservices pattern, the data is spread over multiple services. In this case, it can be

- Product Detail Service
- Merchant Service
- Payment Service
- Deals & Discount Service
- Inventory Service
- Pricing Service
- Review Service
- Recommendation Service How would these individual services be accessed?

The solution is to implement an API Gateway, which serves as the single entry point for all clients and calls specific Microservices spread across the entire infrastructure as per requirement.

Microservices Case Study

Scenario: Drone delivery

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

Developing Microservices

Domain analysis. To avoid some common pitfalls when designing microservices, use domain analysis to define your microservice boundaries. Follow these steps:

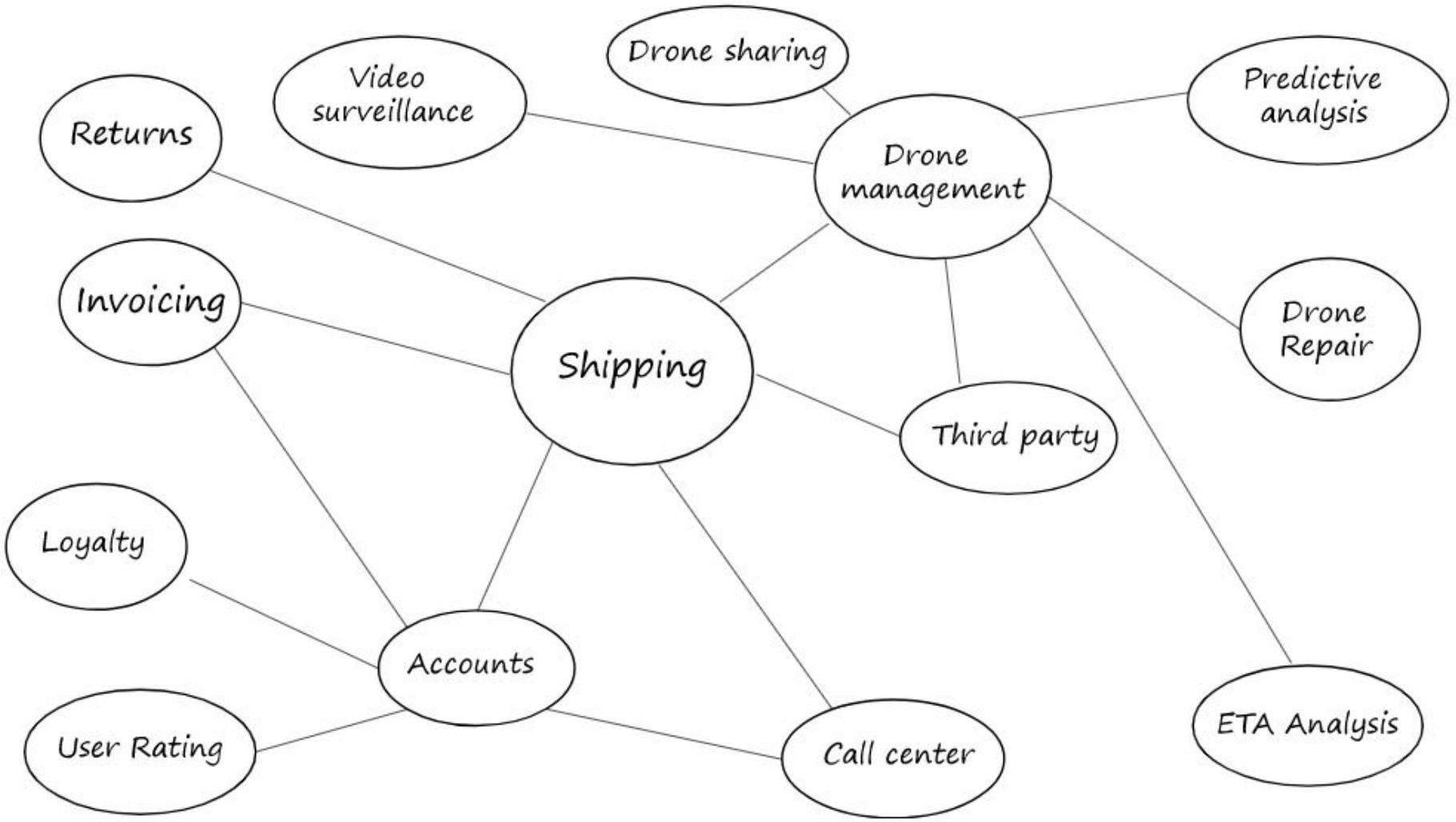
1. Use domain analysis to model microservices.
2. Use tactical DDD to design microservices. (*Domain-Driven Design*)
3. Identify microservice boundaries.

Design the services. Microservices require a different approach to designing and building applications. For more information, see Designing a microservices architecture.

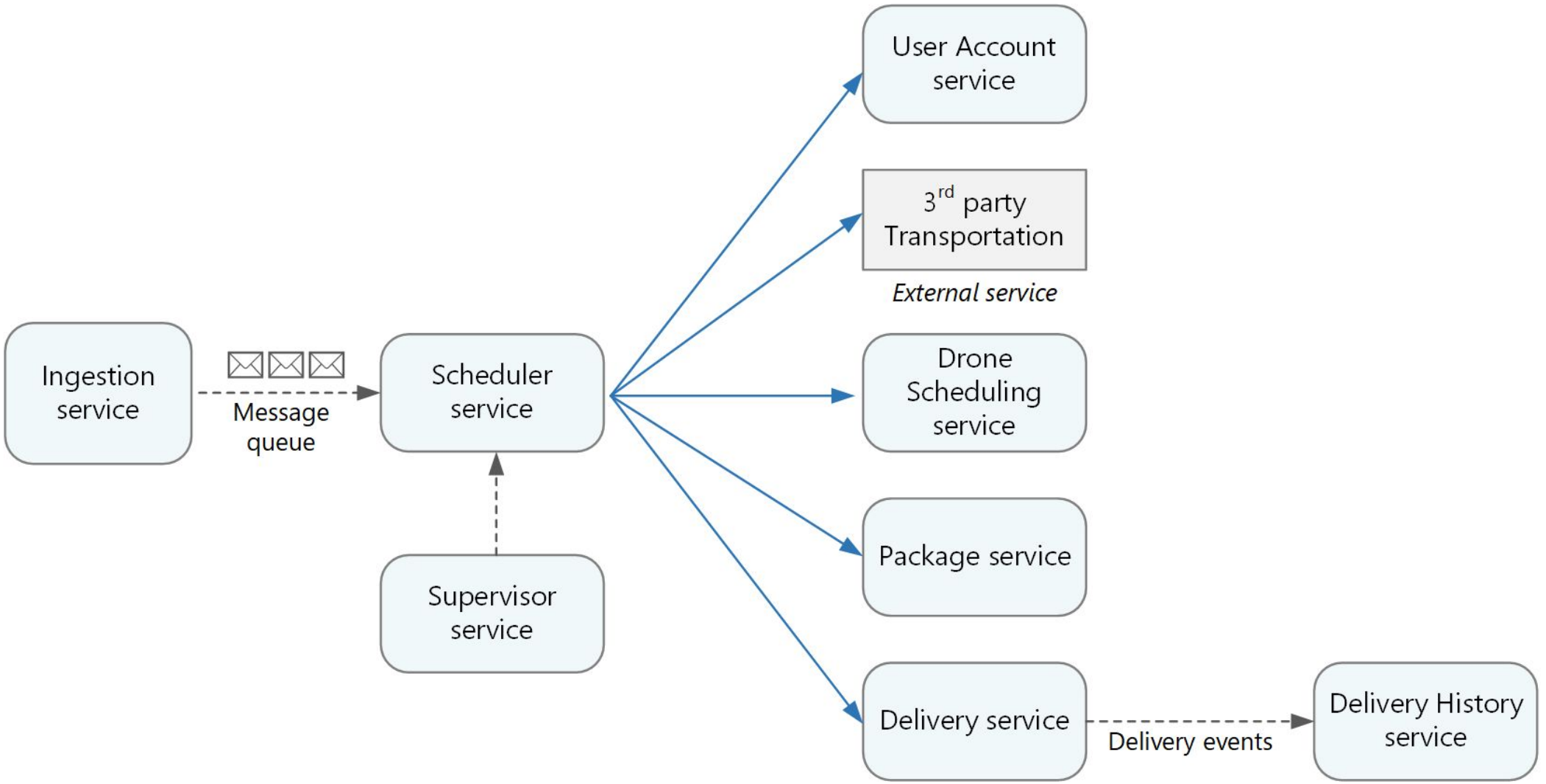
Operate in production. Because microservices architectures are distributed, you must have robust operations for deployment and monitoring.

- CI/CD for microservices architectures
- Build a CI/CD pipeline for microservices on Kubernetes
- Monitor microservices running on Azure Kubernetes Service (AKS)

Domain Analysis



Design Service



What are microservices?

- Microservices are small, independent, and **loosely coupled**. A single small team of developers can write and maintain a service.
- Each service is a **separate codebase,** which can be managed by a small development team.
- Services can be **deployed independently.** A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Services don't need to share the same technology stack, libraries, or frameworks.

Other Components in Microservices

- **Management/orchestration**. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

- **API Gateway**. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

    - API Gateway decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.

References

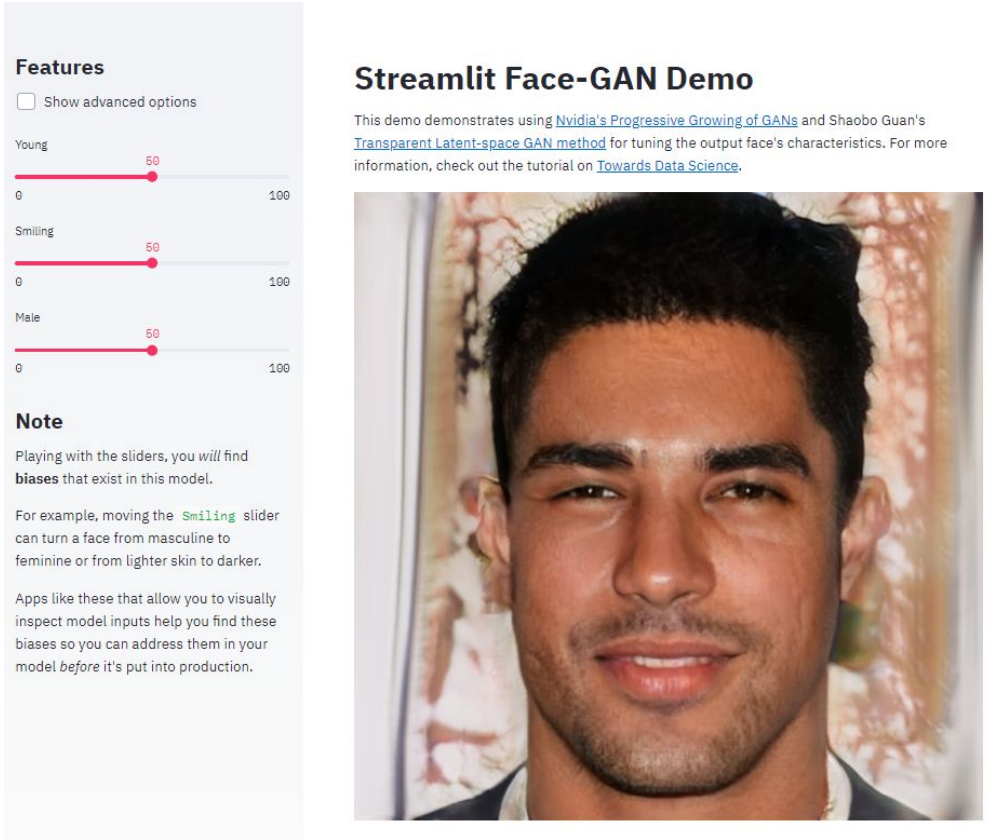
- https://www.fullstackpython.com/microservices.html
- https://martinfowler.com/articles/microservices.html
- https://docs.microsoft.com/en-in/azure/architecture/microservices/

Streamlit

The fastest way to build Apps

Streamlit

Streamlit turns data scripts into shareable web apps in minutes.
All in Python. No front-end experience required.



Live Demos

- Streamlit Components
- Taxi Dataset
- Streamlit Demo: The Udacity Self-driving Car Image Browser

Streamlit Cheatsheet

- https://share.streamlit.io/daniellewisdl/streamlit-cheat-sheet/app.py