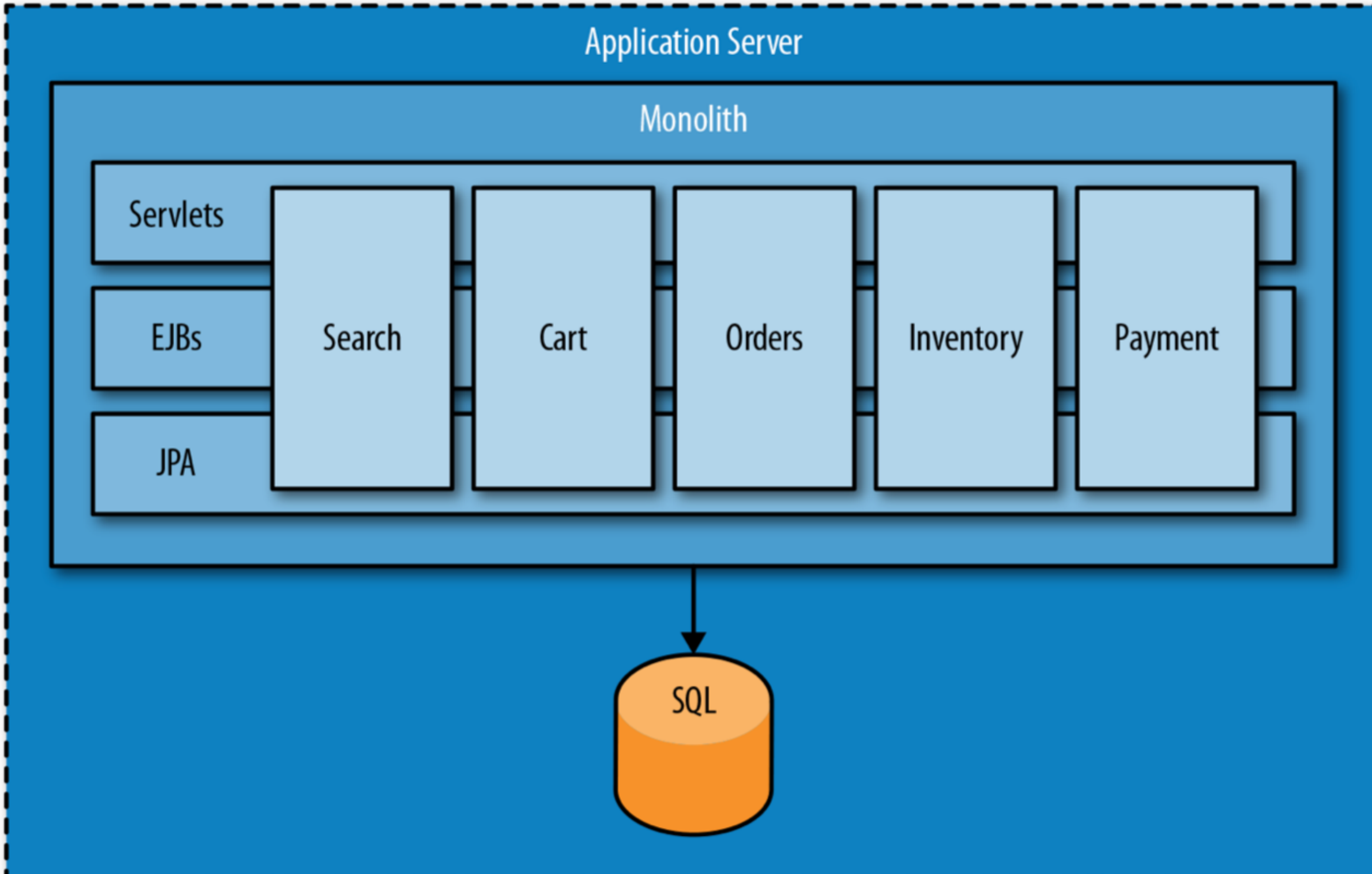
Internet of Things and Services

Service-oriented architectures

Reactive microservices and serverless computing

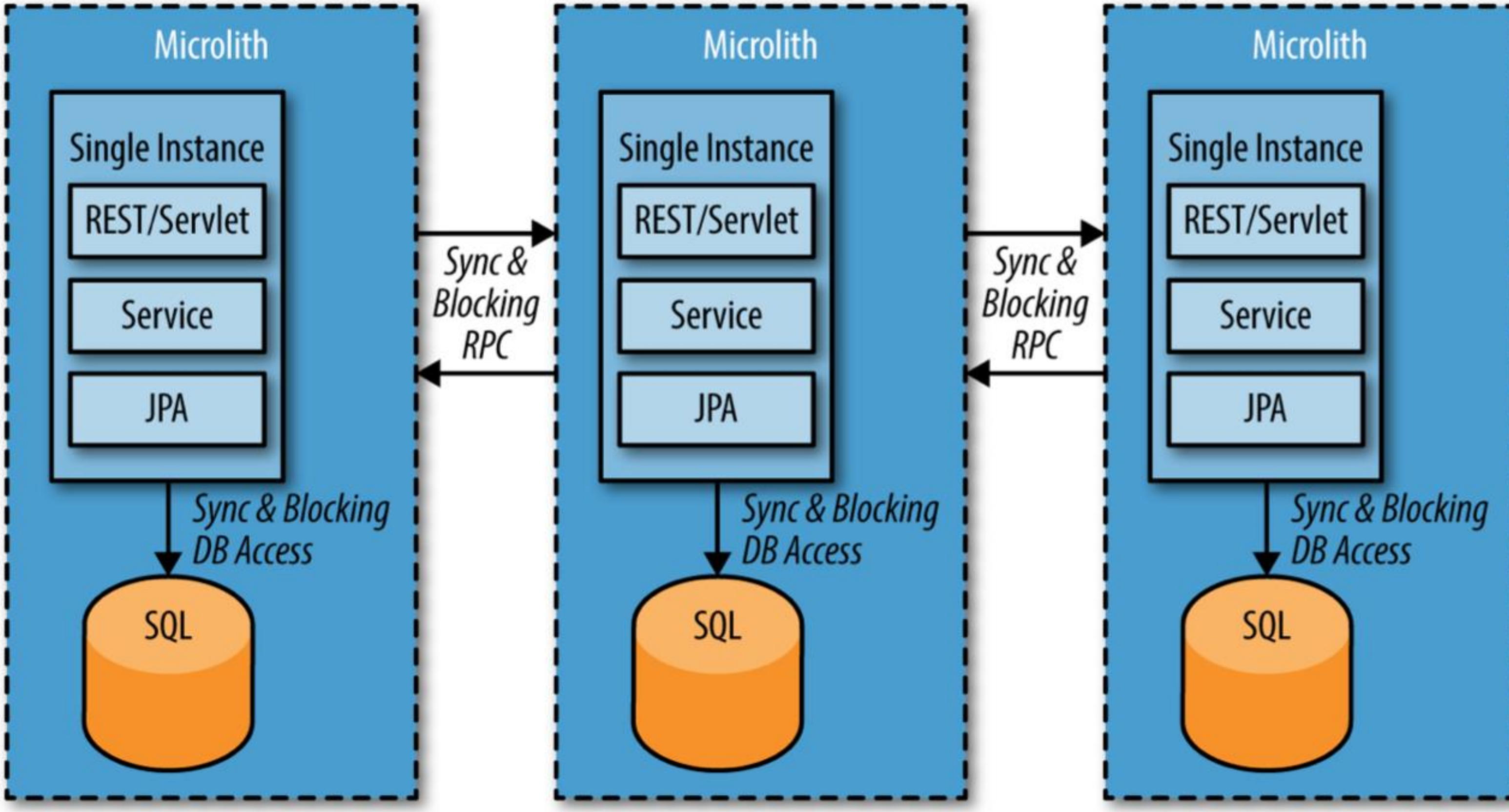
Department of Computer Science
Faculty of Electronic Engineering, University of Nis

Monolith architecture

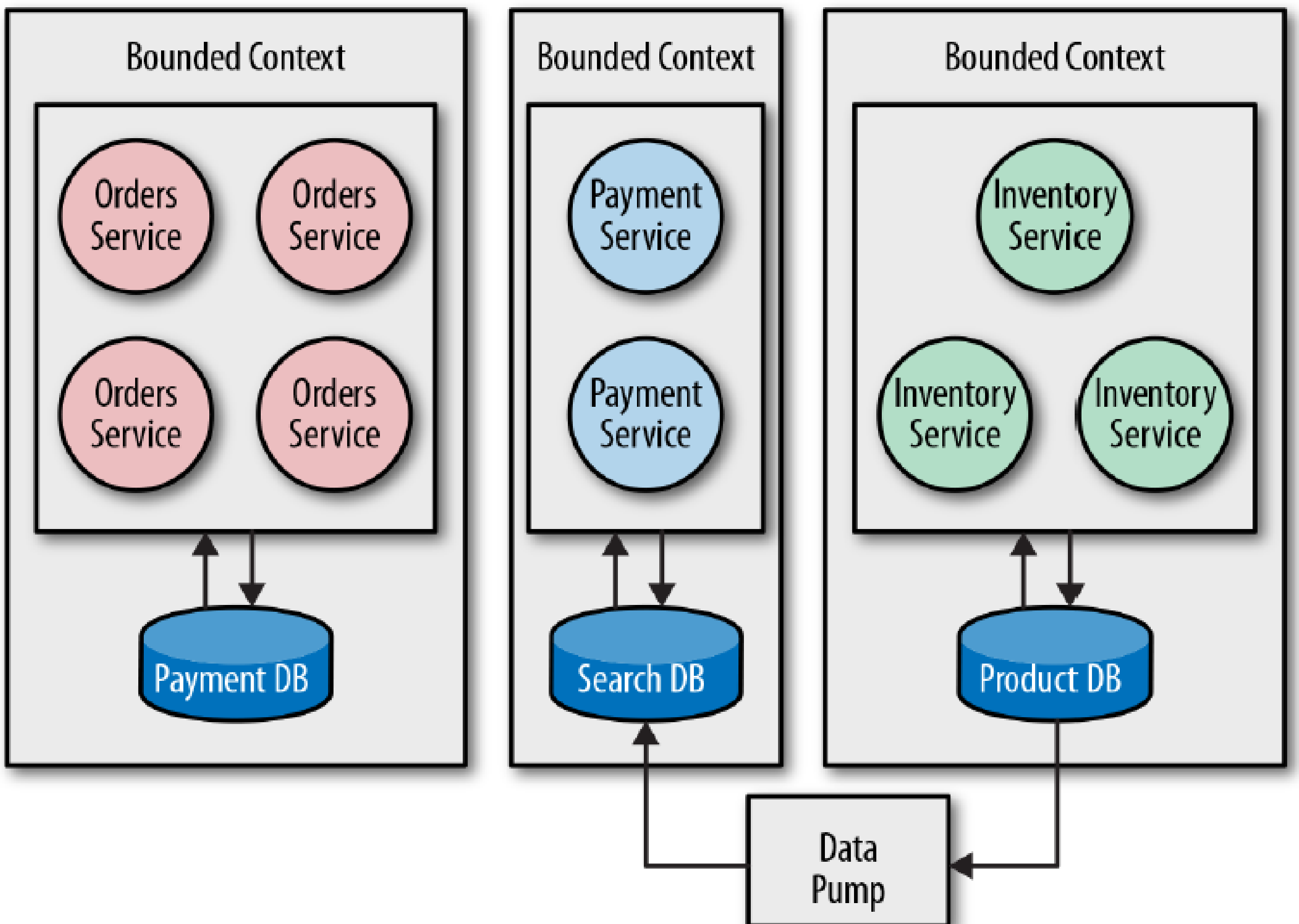


Microlith architecture

- Single instance microservices communicating over blocking protocols – Not resilient, not scalable



Microservice architecture



Essential traits of a microservice (1)

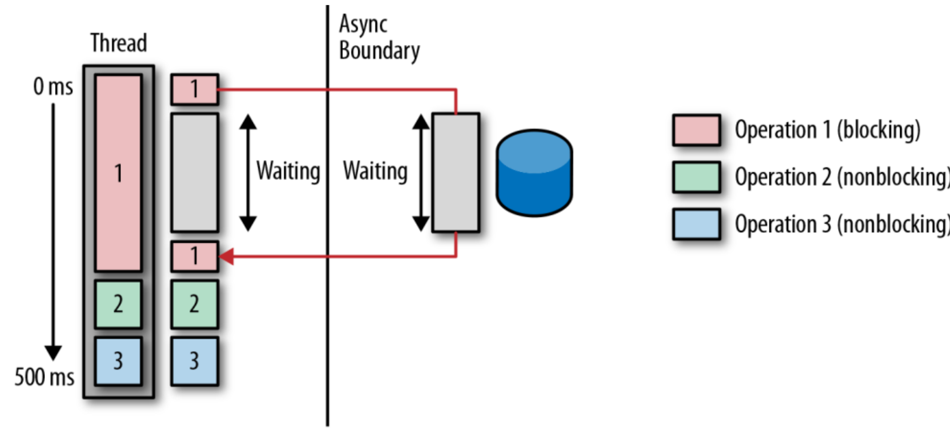
- Isolation
  - A prerequisite for resilience and elasticity and requires asynchronous communication boundaries between services to decouple them in: Time (allowing concurrency) and Space (allowing distribution and mobility—the ability to move services around)
  - Isolation also makes it easier to scale each service, as well as allowing them to be monitored, debugged and tested independently
- Autonomicity
  - Only when services are isolated can they be fully autonomous and make decisions independently, act independently, and cooperate and coordinate with others to solve problems.
- Single responsibility
  - Do One Thing, and Do It Well

Essential traits of a microservice (2)

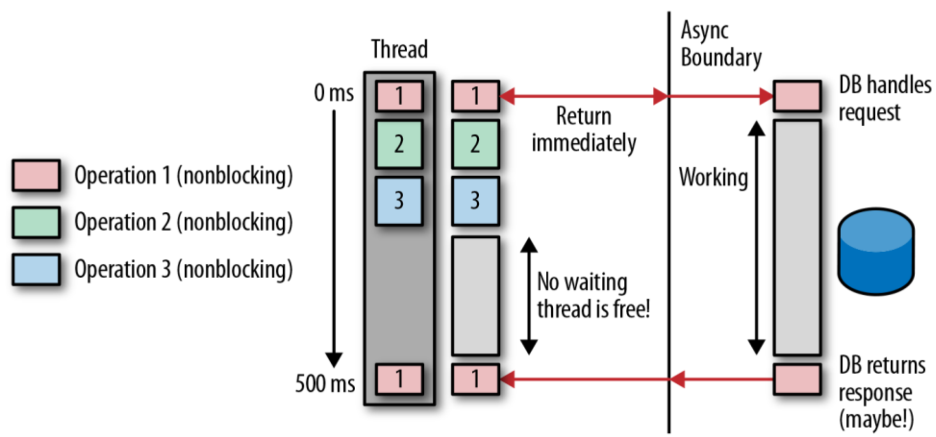
- Exclusive state (Own Your State, Exclusively)
  - Responsibility for their own state and the persistence thereof.
  - When communicating with another Microservice, across Bounded Contexts, each service responds to a request at its own will, with immutable data (facts) derived from its current state, and never exposes its mutable state directly.
  - If the Command to the service triggers a state change in the service then we can capture the state change as a new fact in an Event to be stored in the Event Log using Event Sourcing.
- Asynchronous Message-Passing
  - Asynchronous boundary between services is necessary in order to decouple them, and their communication flow, in *time* and *space*
- Mobility
  - Decoupling in space, *Location Transparency*, the ability to, at runtime, dynamically scale the Microservice, either on multiple cores or on multiple nodes, without changing the code

Reactive Programming

- Reactive Programming is essential to the design of microservices, allowing us to build highly efficient, responsive, and scalable services.
- Techniques for Reactive Programming include:
  - Asynchronous execution and I/O
  - Back-pressured streaming,
  - Circuit breakers.



Synchronous calls wait for a response and block the thread.
Blocking occurs because the thread is busy but not working.



Asynchronous calls issue a request and return immediately.
This one is nonblocking because the response is returned later!

Reactive Programming

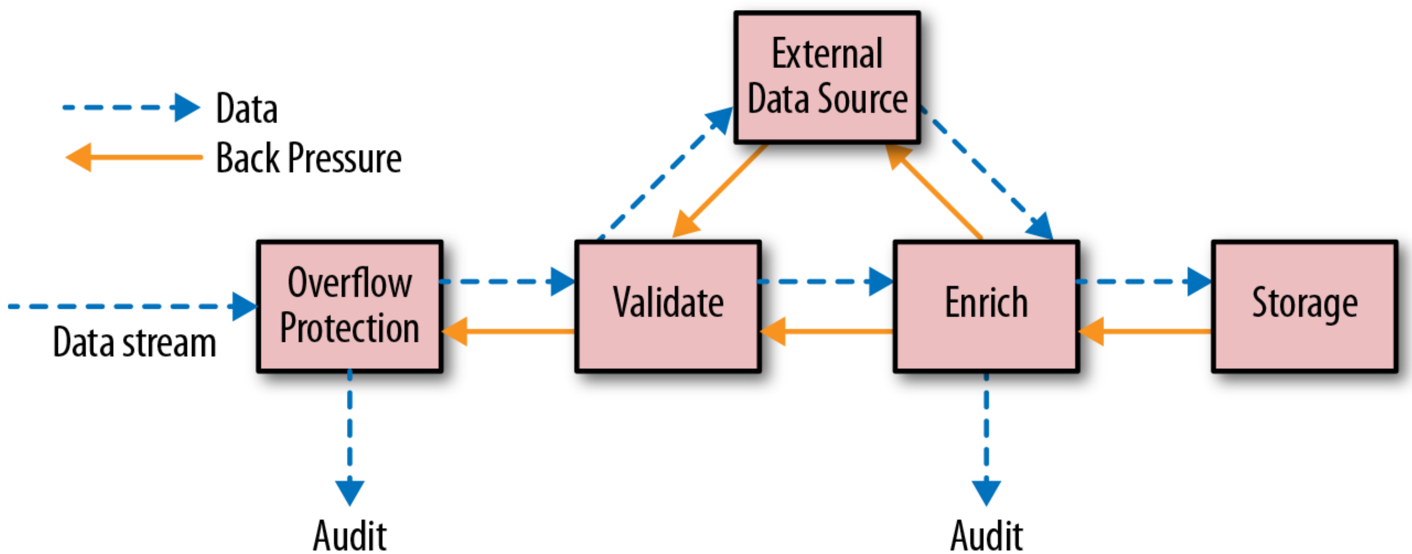
- Many of the ideas behind reactive are not new; plenty of them were described and implemented years ago, for example, Erlang's actor-based programming model (early 1980s), and Akka (JVM).
- The need for concurrent and distributed applications is growing stronger.
- A number of movements have contributed to the current rise of reactive programming, most notably:
  - IoT and mobile systems and applications
    - The server side has to handle millions of connected devices concurrently, a task best handled by asynchronous processing, due to its lightweight ability to represent resources such as "the device," or whatever it might be.
  - Cloud and containerization
    - While we've had cloud-based infrastructure for a number of years now, the rise of lightweight virtualization and containers, together with container-focused schedulers and PaaS solutions, has given us the freedom and speed to deploy much faster and with a finer-grained scope.

Reactive Programming

- ReactiveX extensions (API) for many programming languages - http://reactivex.io/
  - Java (RxJava), C# (Rx.NET), C++ (RxCpp), Python (RxPY), JavaScript (RxJS), Go (RxGo),…
- Reactive frameworks on top of the JDK (JVM based) and implement the Reactive Streams specification
  - Reactor, Vert.x, Akka, Ratpack,…
- Programming languages that have native reactive capabilities
  - Java 9 and Spring Framework 5, and Clojure, Scala, Go,...
- JavaScript libraries
  - Angular.js, React.js, Ractive.js, and Node.js that can be used to build front-end reactive applications.

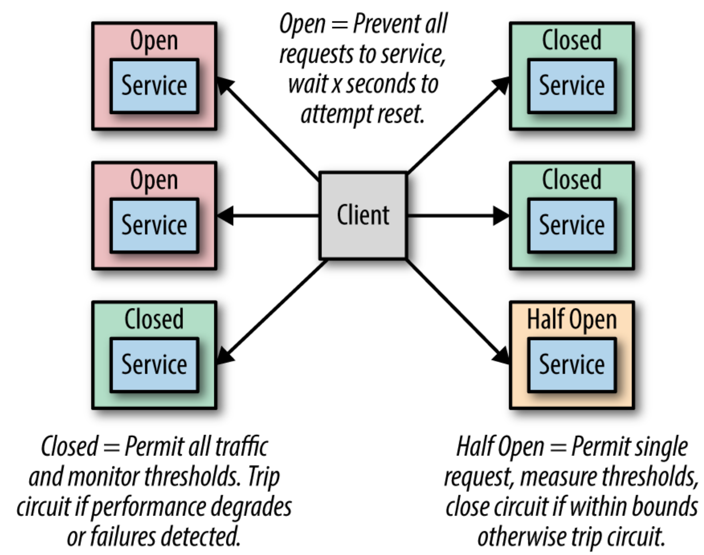
Toward Reactive Microsystems (1)

- Embrace Reactive Programming
- Asynchronous and nonblocking I/O (Pub-Sub messaging)
- Applying backpressure
  - Backpressure is all about flow control, ensuring that a fast producer should not be able to overwhelm a slower consumer by being allowed to send it more data than it can handle

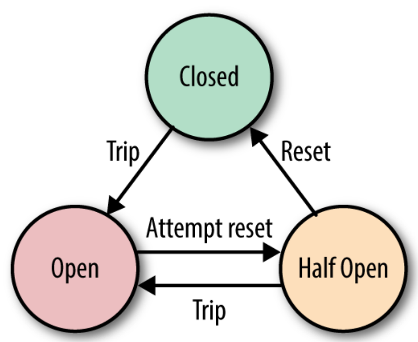


- Circuit breaker
  - A finite-state machine (FSM), which means that it has a finite set of states: *Closed*, *Open*, and *Half-Open*. The default state is *Closed*, which allows all requests to go through
  - When a failure (or a specific number of failures) have been detected, the circuit breaker "trips" and moves to an *Open* state. In this state, it does not let any requests through, and instead fails fast to shield the component from the failed service.
  - After a timeout has occurred, the service is back up again, so it attempts to "reset" itself and move to a *Half*-Open state

Circuit breakers act as a wrapper around external services



When an external service is too slow or begins to fail, a circuit breaker trips and prevents requests to that service...



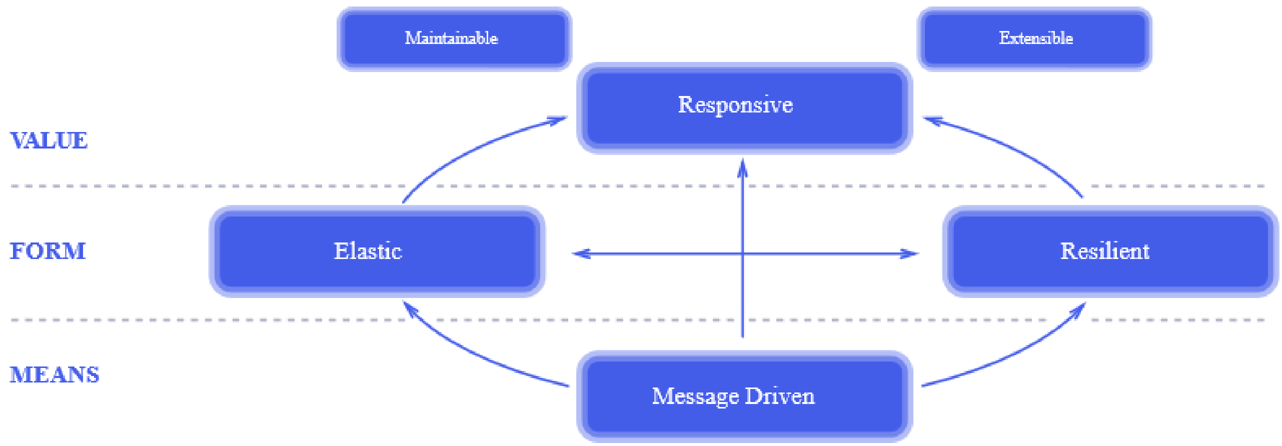
...until the service heals. This prevents cascading failures in a distributed system.

Reactive Systems (1)

- Reactive Manifesto - https://www.reactivemanifesto.org/
- Reactive Systems are:
  - **Responsive:** The system responds in a timely manner if at all possible. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service.
  - **Resilient:** The system stays responsive in the face of failure. This applies not only to highly-available, mission-critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole.

Reactive Systems (2)

- **Elastic**: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.
- **Message Driven**: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary.



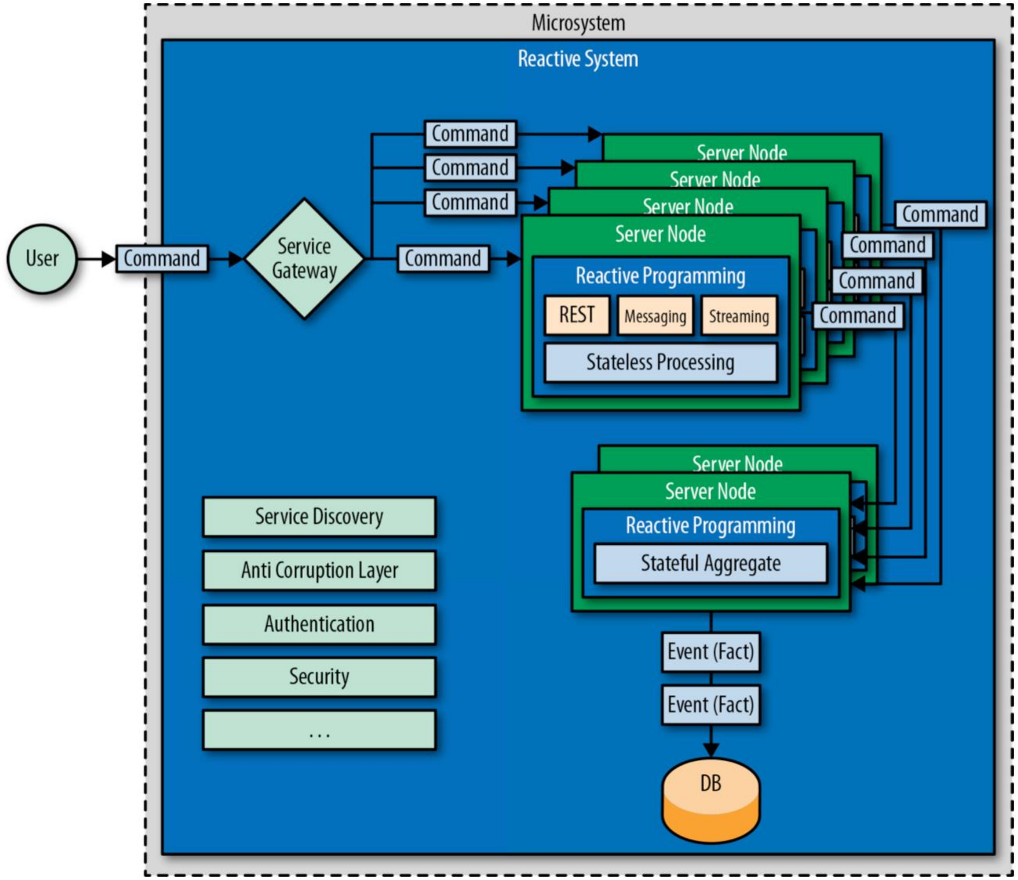
Reactive programming vs. Reactive systems

- ***Reactive Programming*** is a great technique for making individual components performant and efficient through asynchronous and non-blocking execution, most often together with a mechanism for backpressure.
  - It has a *local focus* and is *event-driven*, publishing facts to 0–*N* anonymous subscribers.
- ***Reactive Systems*** takes a holistic view on system design, focusing on keeping *distributed systems* responsive by making them *resilient and elastic*.
  - It is *message-driven*, based upon asynchronous message passing, which makes distributed communication to addressable recipients first class—allowing for elasticity, location transparency, isolation, supervision, and self-healing.

https://www.oreilly.com/radar/reactive-programming-vs-reactive-systems/

Microservices as Reactive Systems

- Each microservice needs to be designed as a component of a distributed system - a Microsystem

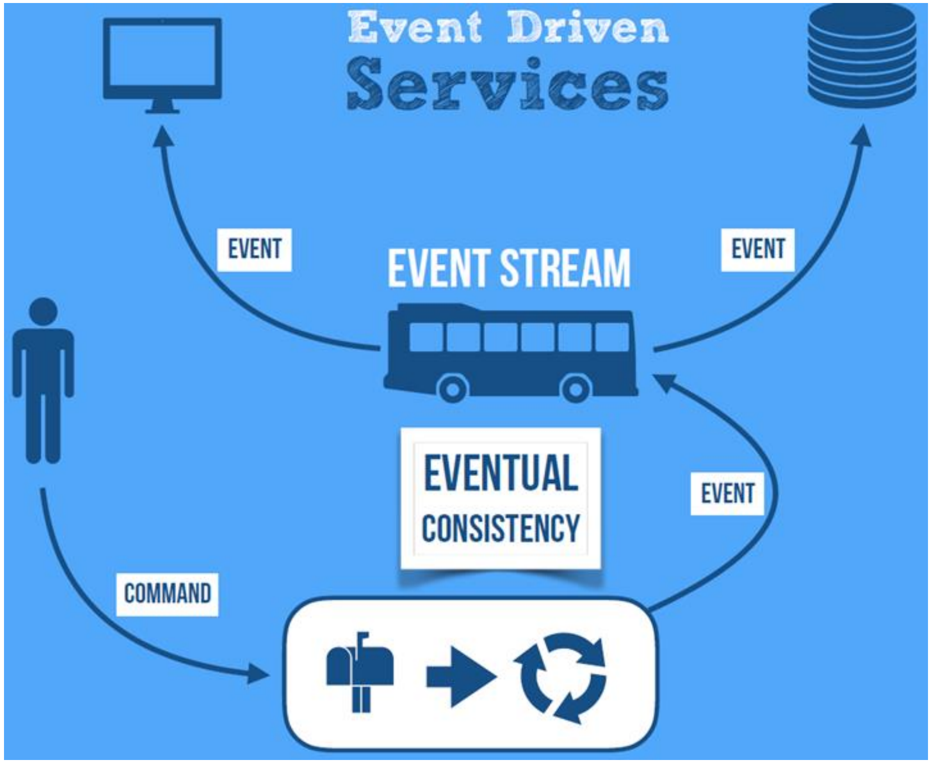


Event Driven Architecture

- Event-Driven Architecture (EDA) is a design paradigm in which a software component executes in response to receiving one or more event notifications
- **Events** represent Facts of information
  - Facts are immutable
  - Facts accrue - Knowledge can only grow
- Events/Facts can be Disregarded/Ignored
- Events/Facts can not be Retracted (once accepted)
- Events/Facts can not be Deleted (once accepted)
- Events/Facts (new) can Invalidate existing Facts

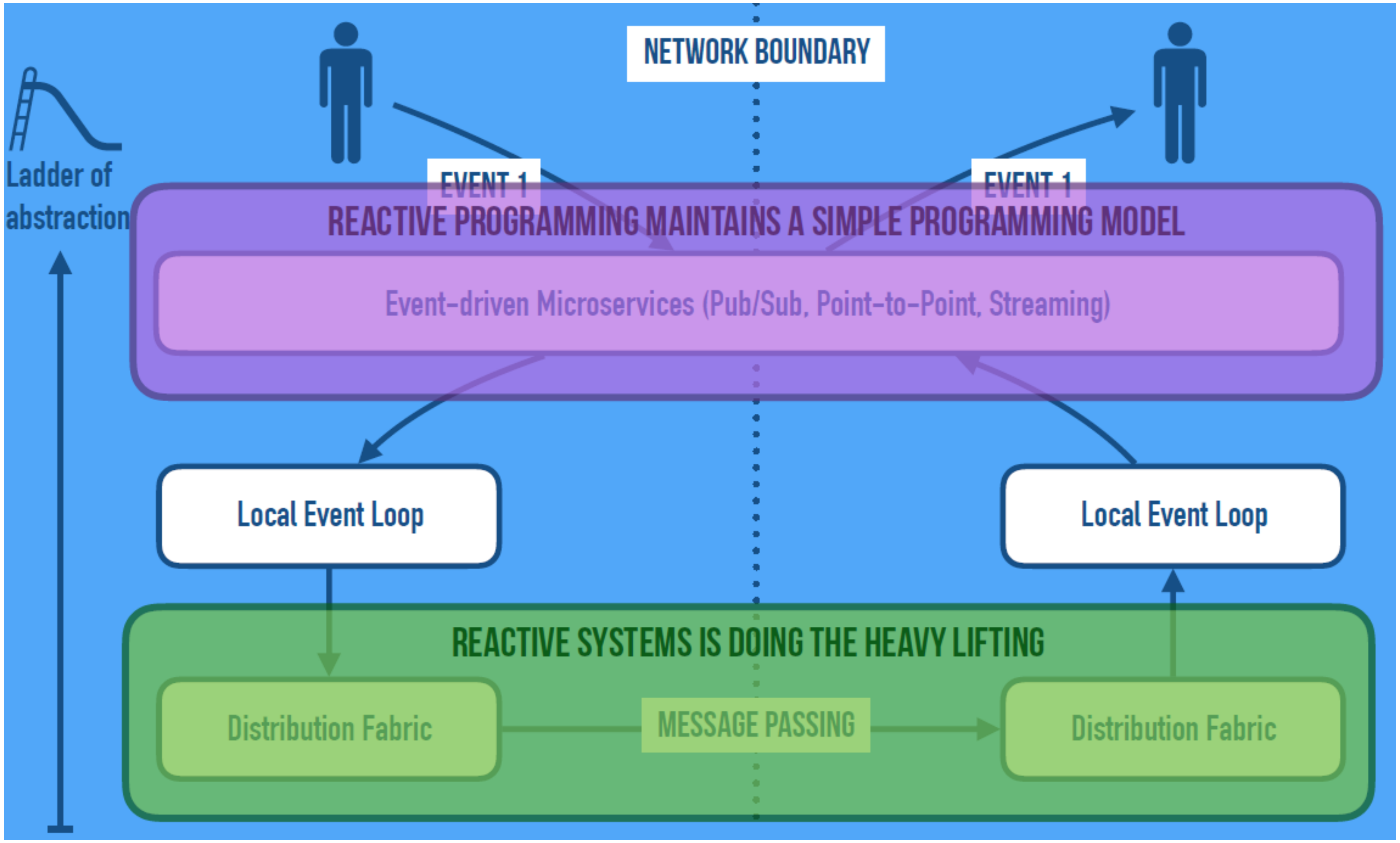
Event Driven Services

1. Receive and react (or not) to facts, that are coming its way
2. Publish new facts (as events) to the rest of the world
3. Invert the control flow to minimize coupling and increase autonomy



Event Driven Microservices

- Event Driven Microservices Powered By Reactive Systems Fabric



Events First Domain Driven Design

- “When you start modeling events, it forces you to think about the behavior of the system. As opposed to thinking about the structure of the system.”
  - Greg Young, A Decade of DDD, CQRS, Event Sourcing
- Don’t focus on the things (Nouns, Domain Objects)
- Focus on what happens (Verbs, Events)

Event Driven Design

INTENTS

- Communication
- Conversations
- Expectations
- Contracts
- Control Transfer

COMMANDS

FACTS

- State
- History
- Causality
- Notifications
- State Transfer

EVENTS

Event Driven Design

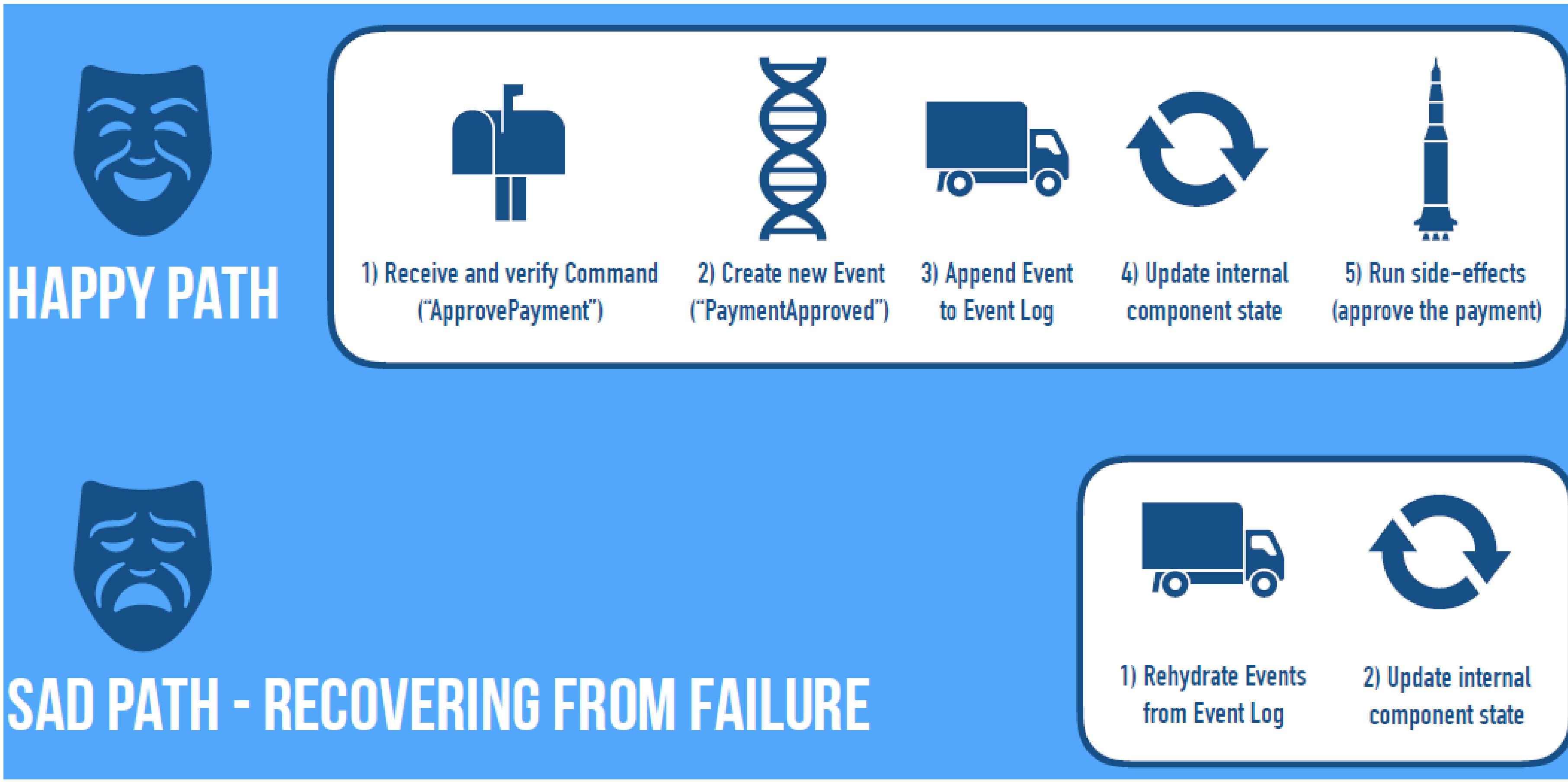
- Commands
  - **Object form of** method/Action request
  - Imperative: CreateOrder, ShipProduct
- Reactions
  - **Represents** side-effects
- Events
  - **Represents something that** has happened
  - Past-tense: OrderCreated, ProductShipped

COMMANDS vs	EVENTS
1. All about intent	1. Intentless
2. Directed	2. Anonymous
3. Single addressable destination	3. Just happens – for others (0-N) to observe
4. Models personal communication	4. Models broadcast (speakers corner)
5. Distributed focus	5. Local focus
6. Command & Control	6. Autonomy

Event Sourcing

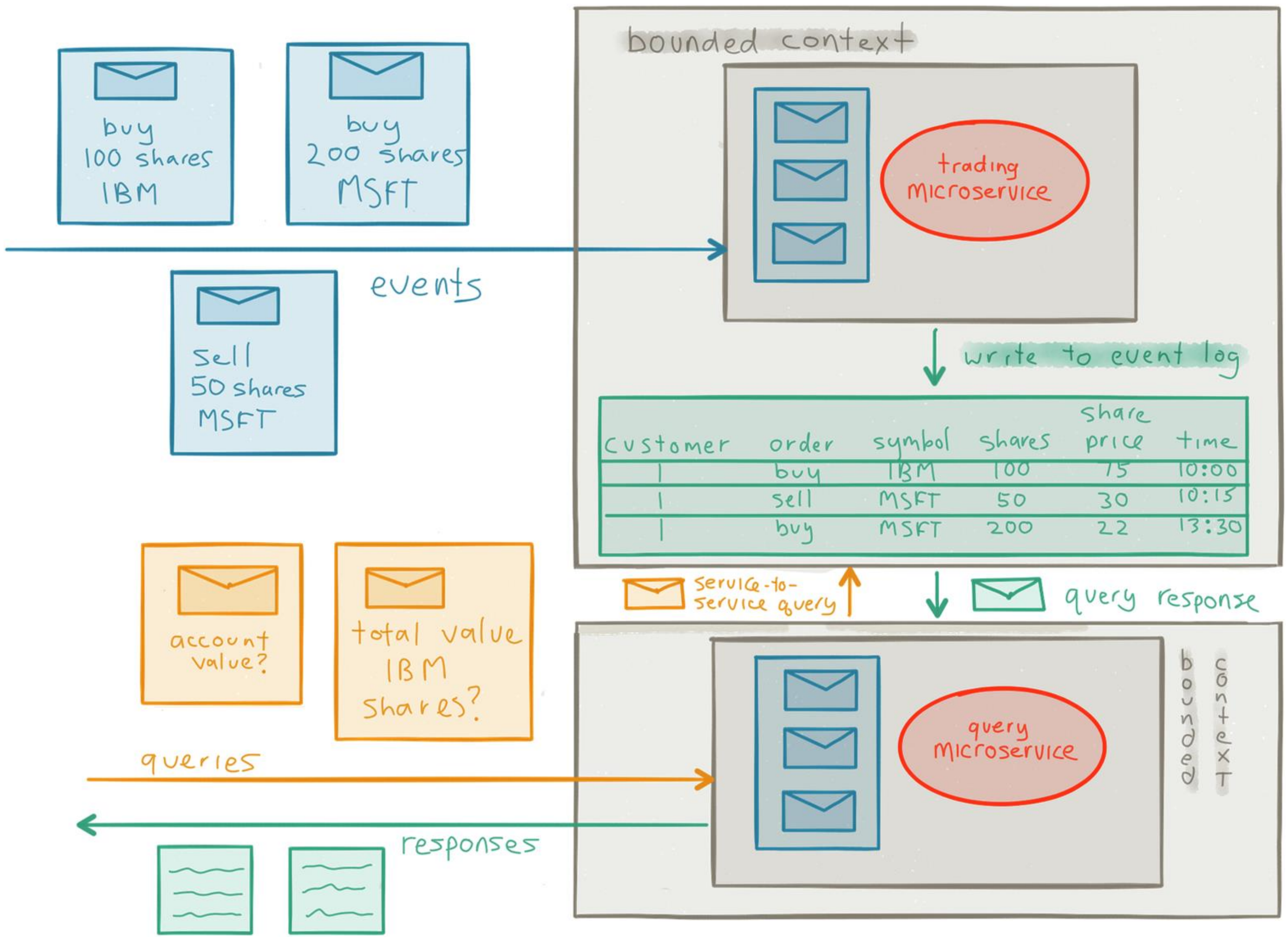
- Benefits
  - One single Source of Truth with All history
  - Allows for Memory Image - durable in-memory state
  - Avoids the Object-relational Impedance mismatch
  - Allows others to Subscribe to state changes
  - Mechanical sympathy (Single Writer Principle etc.)
- Disadvantages
  - Unfamiliar model
  - Versioning of events
  - Deletion of events (legal or moral reasons)

Event Sourced Microservices



Event Logging and CQRS

- Event-based persistence through Event Logging and CQRS



Reactive Microservice Systems

- Key design principles

1. Don’t build Microliths
2. Microservices come in (distributed) systems
3. Microservices come as (micro)systems
4. Embrace the Reactive Principles
5. Embrace Events-first Domain Driven Design
6. Embrace Event-driven Architecture
7. Embrace Event-driven Persistence

SERVERLESS COMPUTING

Serverless computing

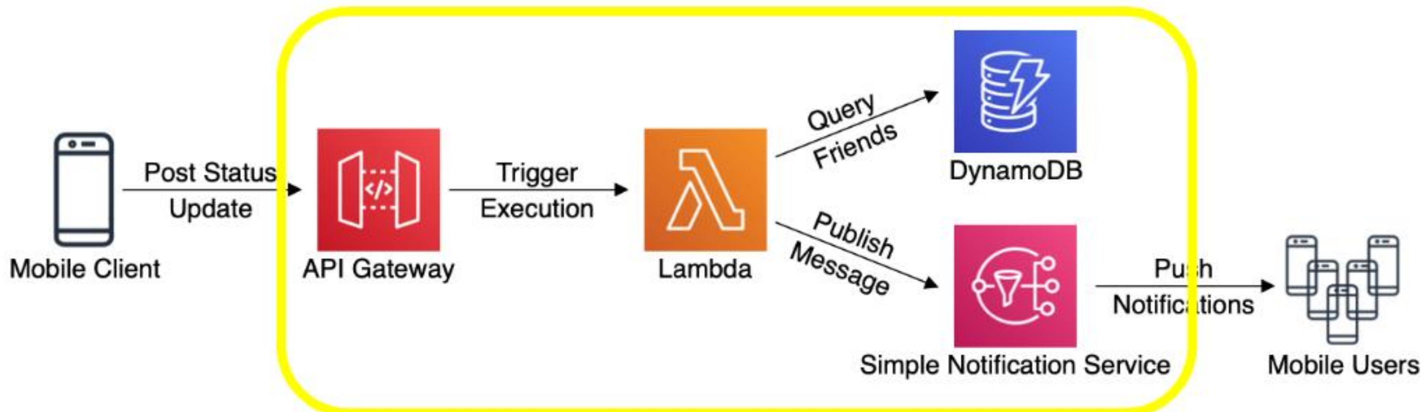
- Cloud computing model which aims to abstract server management and low-level infrastructure decisions away from users
  - Users can develop, run and manage application code (i.e., functions), without no worry about provisioning, managing and scaling computing resources
  - Runtime environment is fully managed by Cloud provider
  - Serverless: functions still run on “servers” somewhere but we don’t care
- Function as a Service (FaaS) often as synonym

Serverless computing: characteristics

- Ephemeral compute resources
  - May only last for one function invocation
  - Automated (i.e., zero configuration) elasticity
- True pay-per-use
  - Pay only for consumed compute time, rather than on pre-purchased units of capacity
- Event-driven
  - When an event is triggered, a piece of infrastructure is allocated dynamically to execute the function code
- Can simplify the process of deploying code into production
  - Scaling, capacity planning and maintenance operations are hidden from developers or operators

Example of serverless application

- Mobile backend for a social media app
  1. The user composes the status update using the mobile clients of the social media platform
  2. The users ends the status update using the mobile client
  3. The platform orchestrates the operations needed to propagate the update inside the social media platform and to the user's friends using serverless technology
  4. Each friend receives the update on their social media clients



Serverless cloud services

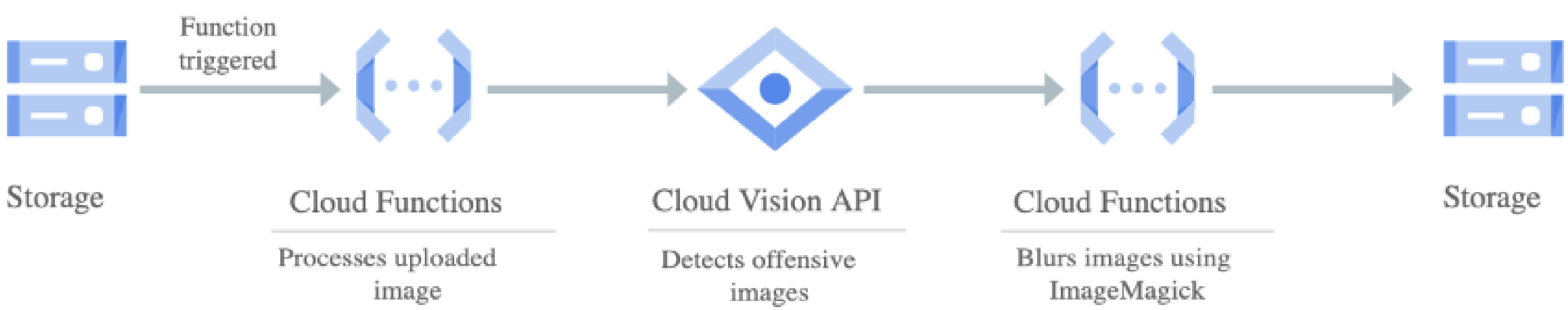
- Several Cloud service providers offer serverless computing on their public clouds as fully managed service
  - AWS Lambda
    - Includes lambda functions at the edge
  - Azure Functions
  - Google Cloud Functions
  - IBM Cloud Functions
- These cloud platforms also offer other supporting services (e.g., event notification, storage, database services) that are necessary for operating an overall serverless ecosystem

Example: Google Cloud Functions

- The “Hello World” FaaS example from Google
  - HTTP request written in Node.js that displays “Hello World” or “Hello (name)” if you pass in a parameter

```
/**
   * HTTP Cloud Function.
   *
   * @param {Object} req Cloud Function request context.
   * @param {Object} res Cloud Function response context.
   */
   exports.helloHttp = function helloHttp (req, res) {
   res.send(`Hello ${req.body.name || 'World'}!`);
   };
```

- A more sophisticated example



Serverless computing: challenges and limitations

- Performance
  - Startup latency and cold starts
  - “The first time you deploy a function it may take several minutes as we need to provision the underlying infrastructure to support your functions. Subsequent deployments will be much faster.” (Google Cloud Functions)
- Programming language support
- Resource limits
- Lack of standards and vendor lock-in

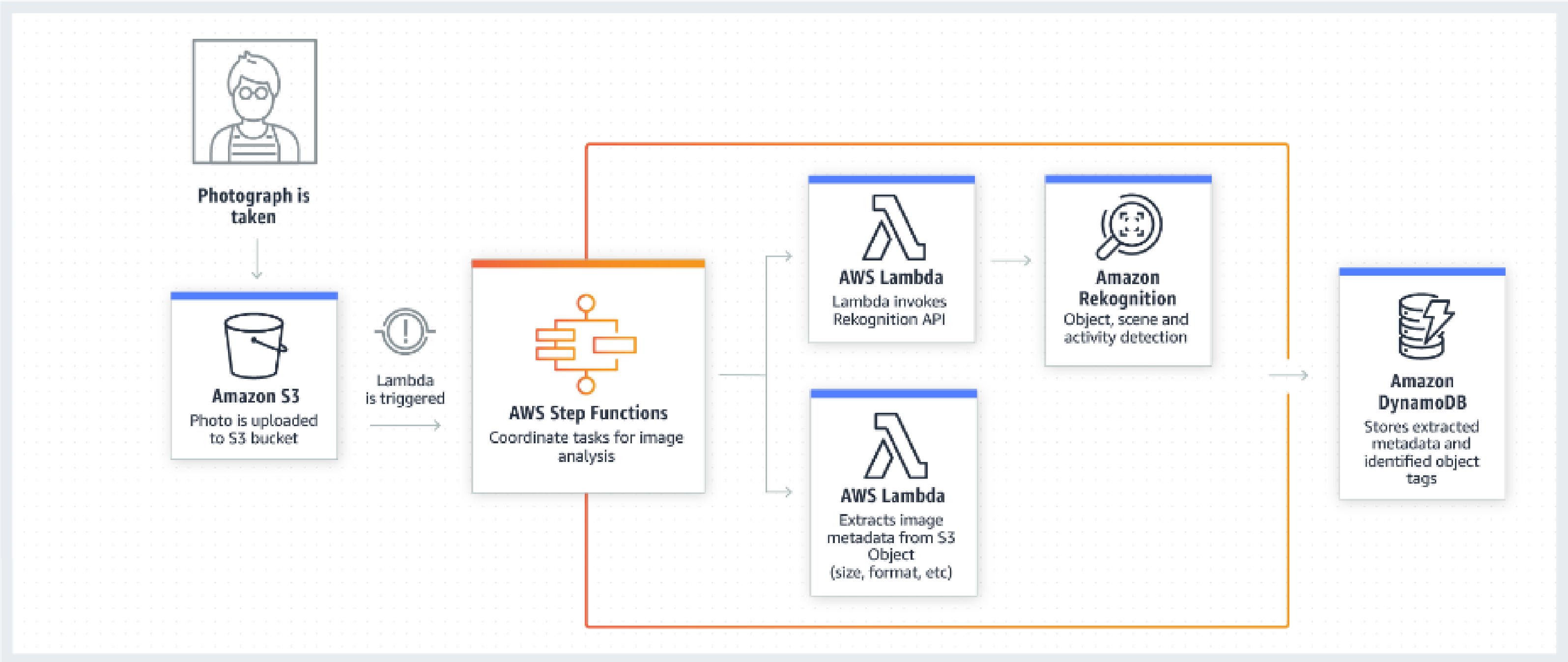
Reduced flexibility

Composition of serverless functions

- Write small, simple, stateless functions
  - Complex functions are hard to understand, debug, and maintain
  - Separate code from data structures
- Then compose them in a workflow

Example: AWS Step Functions

- AWS Step Functions is a serverless orchestration service that allows the developer to coordinate multiple Lambda functions into workflows
- Example: process photo immediately after its upload in S3

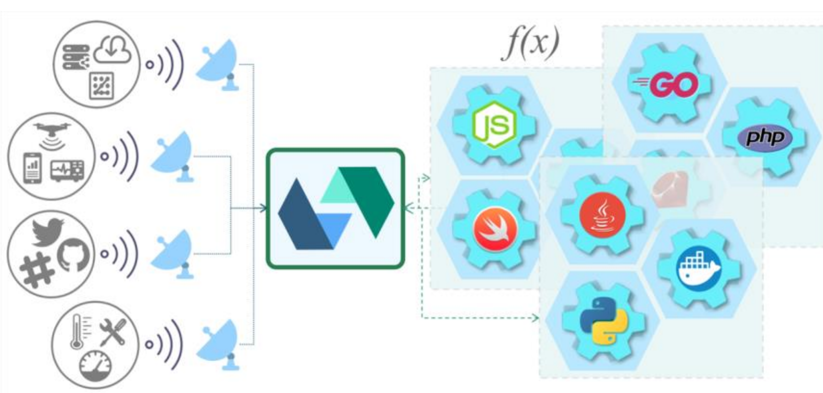


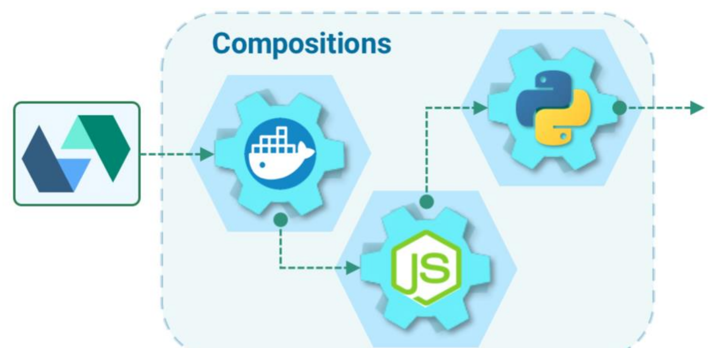
Open-source FaaS platforms

- Can run on commodity hardware
- Most platforms rely on Kubernetes for orchestration and management of serverless functions
  - Configuration management of containers
  - Container scheduling and service discovery
  - Elasticity management
- Prominent platforms
  - Apache OpenWhisk - https://openwhisk.apache.org/
  - OpenFaaS - https://www.openfaas.com/
  - Nuclio - Serverless Platform for Automated Data Science - https://nuclio.io/
  - Fission - https://fission.io/
  - Knative - https://knative.dev/

OpenWhisk

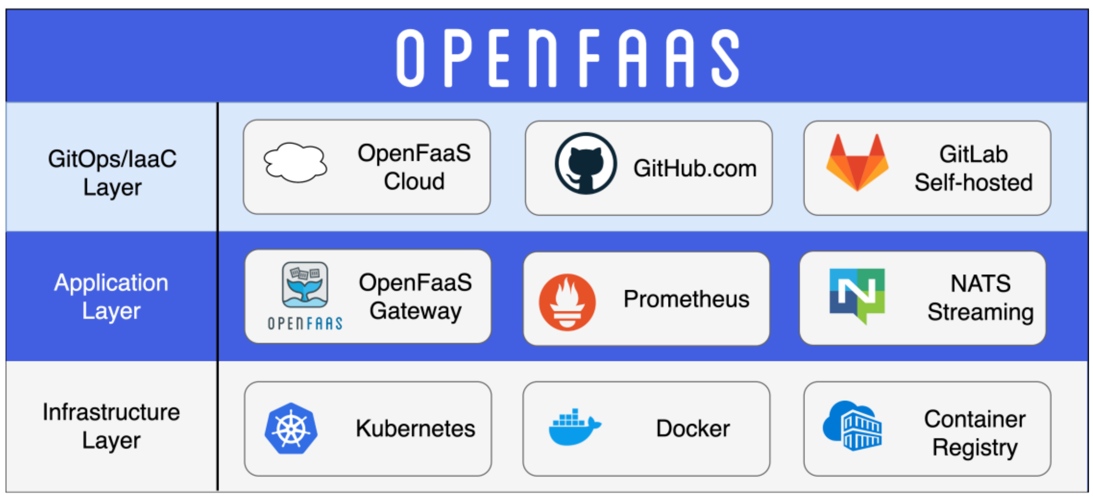
- Open-source, distribute serverless platform that executes functions in response to events at any scale
- Based on Docker containers
- Deployed on Kubernetes, OpenShift, Mesos, Docker Compose
- Developers write functional logic (called Actions)
  - In any supported programming language
  - Can be dynamically scheduled and run in response to associated events (via Triggers) from external sources (Feeds) or from HTTP requests
- Functions can be combined into compositions





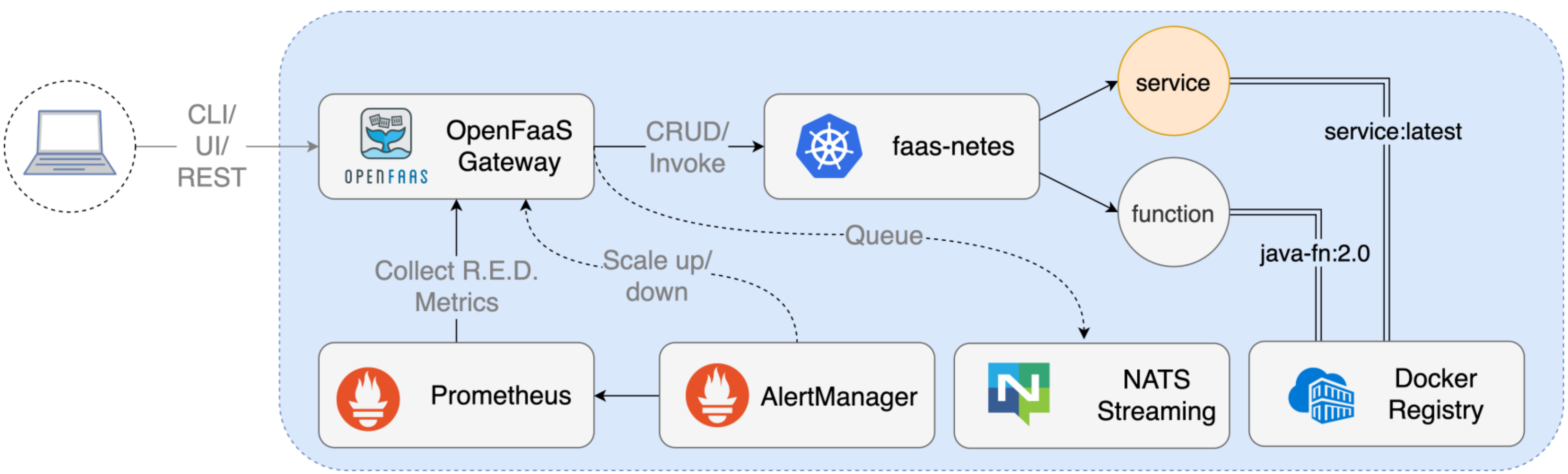
OpenFaaS

- Open-source FaaS framework for building functions on top of Docker and Kubernetes
- OpenFaaS stack
  - Gateway provides an external route into the functions, collects metrics and scale functions
  - Prometheus provides metrics and enables auto-scaling
  - NATS provides asynchronous execution and queuing



OpenFaaS

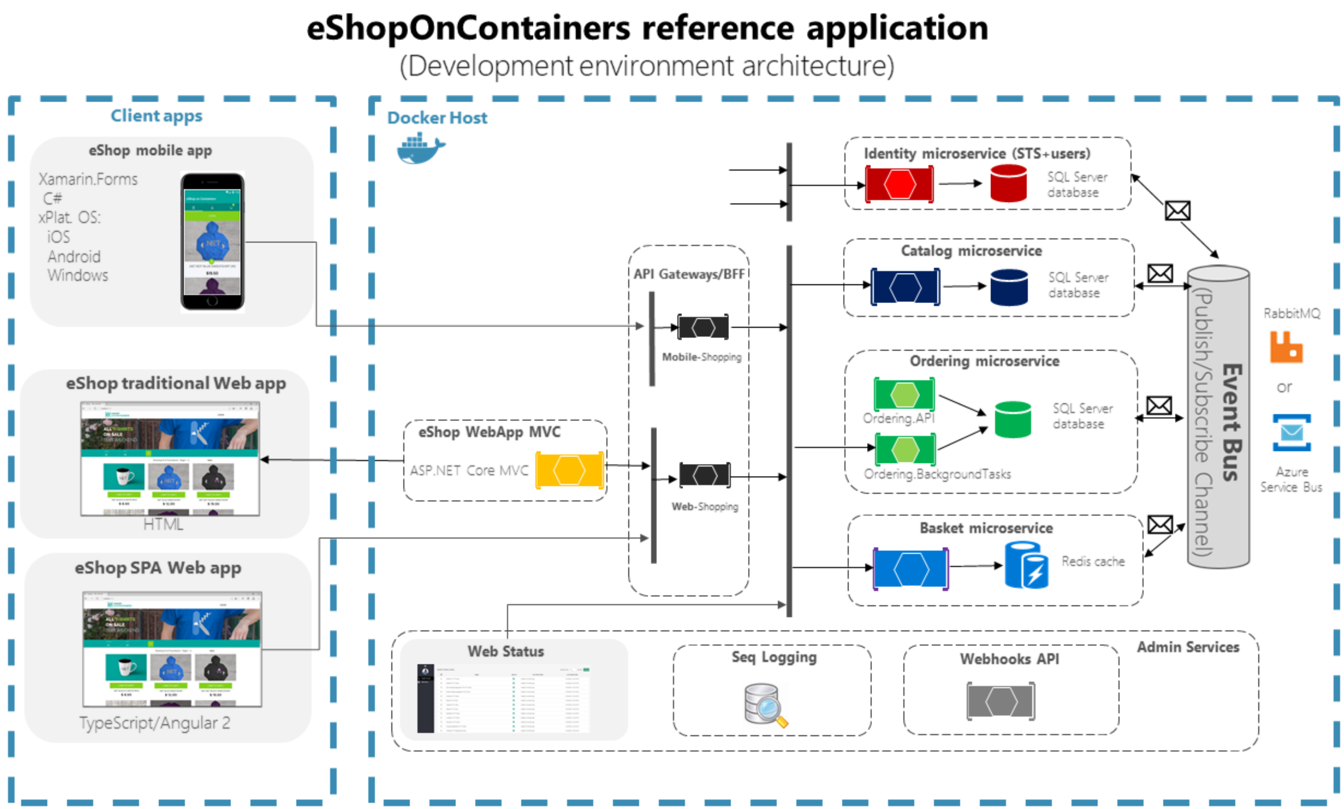
- Conceptual workflow
  - OpenFaaS Gateway can be accessed through its REST API, via CLI or through UI
  - Prometheus collects metrics which are available via Gateway's API and are used for auto-scaling
  - NATS Streaming enables long-running tasks or function invocations to run in the background



MICROSERVICE EXAMPLES

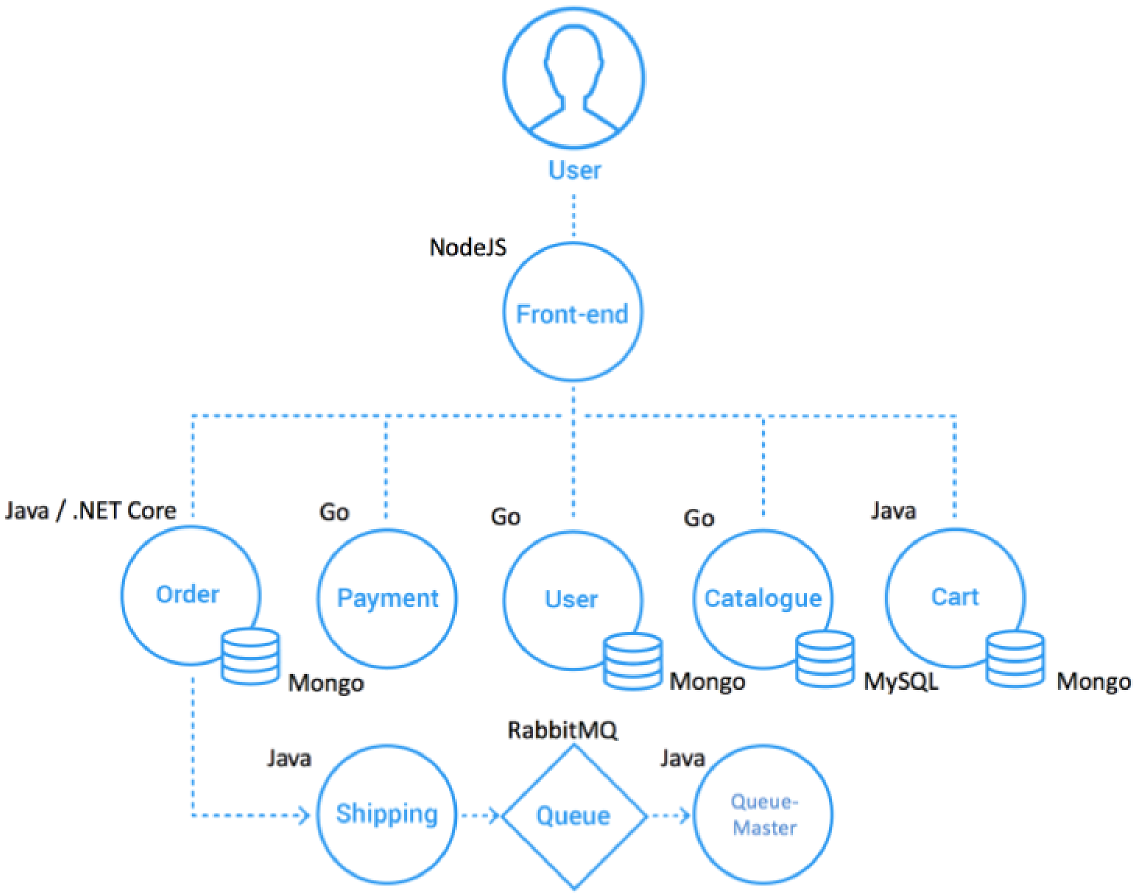
Example 1: eShopOnContainers

- eShopOnContainers
  - https://github.com/dotnet-architecture/eShopOnContainers
  - https://dotnet.microsoft.com/learn/aspnet/microservices-architecture



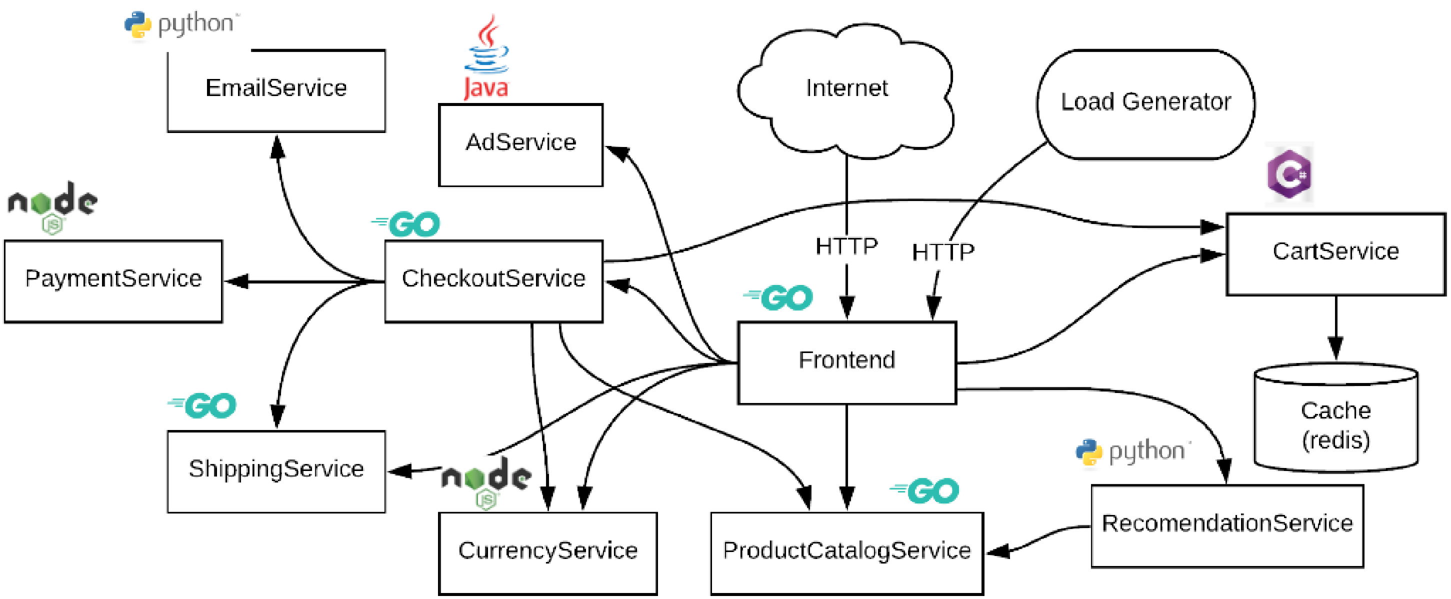
Example 2: Sock shop

- An online shop https://microservices-demo.github.io/



Example 3: Online boutique

- Another online store https://github.com/GoogleCloudPlatform/microservices-demo
  - Composed of many microservices written in different languages that talk to each other over gRPC
  - Used by Google to demonstrate use of Kubernetes/GKE, Istio, Stackdriver, gRPC and OpenCensus



References

- Jonas Boner, *Reactive microservice architecture: Design Principles for Distributed Systems*, O'Reilly Media Inc. 2016.
  - https://www.lightbend.com/ebooks/reactive-microservices-architecture-design-principles-for-distributed-systems-oreilly
- Jonas Boner, *Reactive Microsystems: The Evolution of Microservices at Scale*, O'Reilly Media Inc. 2017.
  - https://www.lightbend.com/ebooks/reactive-microsystems-evolution-of-microservices-scalability-oreilly
- Reactive Architecture: Foundations (CognitiveClass course) https://cognitiveclass.ai/learn/reactive-architecture-foundations
  - Reactive Architecture: Introduction to Reactive Systems
  - Reactive Architecture: Domain Driven Design
  - Reactive Architecture: Reactive Microservices

References - Serverless

- M. Roberts, *Serverless Architectures*,
  - https://martinfowler.com/articles/serverless.html
- P. Castro et al., *The Rise of Serverless Computing*, ACM Comm., Dec. 2019.
  - https://cacm.acm.org/magazines/2019/12/241054-the-rise-of-serverless-computing/fulltext
- E. Jonas et al., *Cloud Programming Simplified: A Berkeley View on Serverless Computing*, Technical Report No. UCB/EECS-2019-3, 2019.
  - https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf