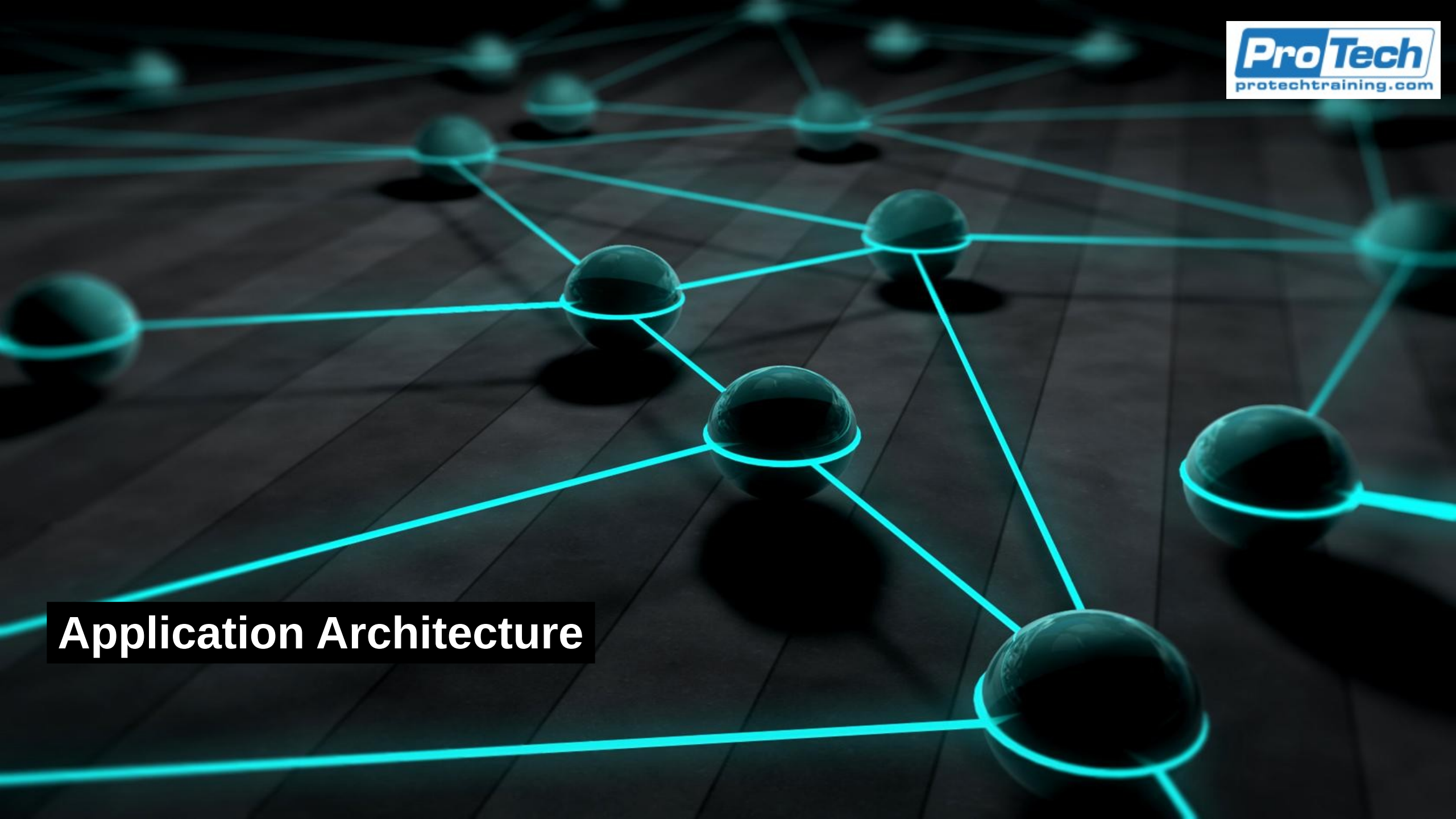
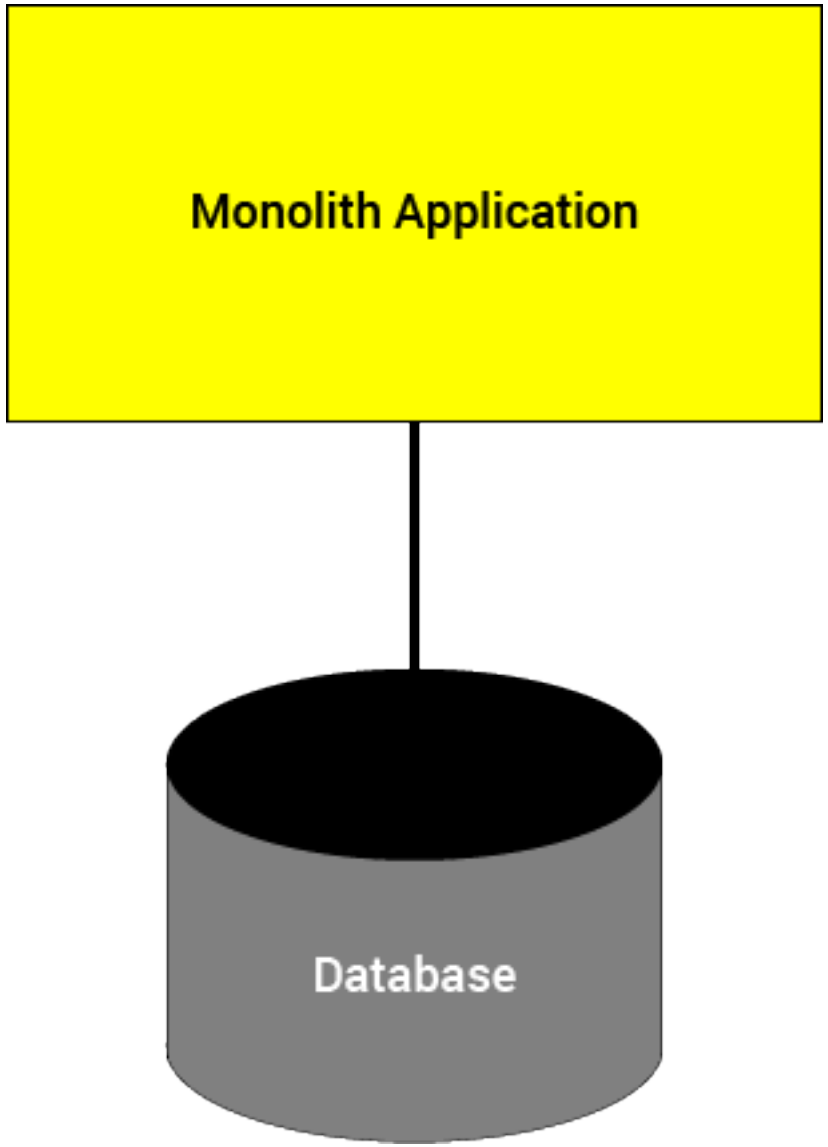


Application Architecture

Monolith Application

- Characterized by a single code base
  - May be modular at the programming language level
- Integrated with a single database
  - All code uses a common schema
- “Monolith” means
  - If a change to the code base or to the data schema
  - Then the entire application needs to be redeployed

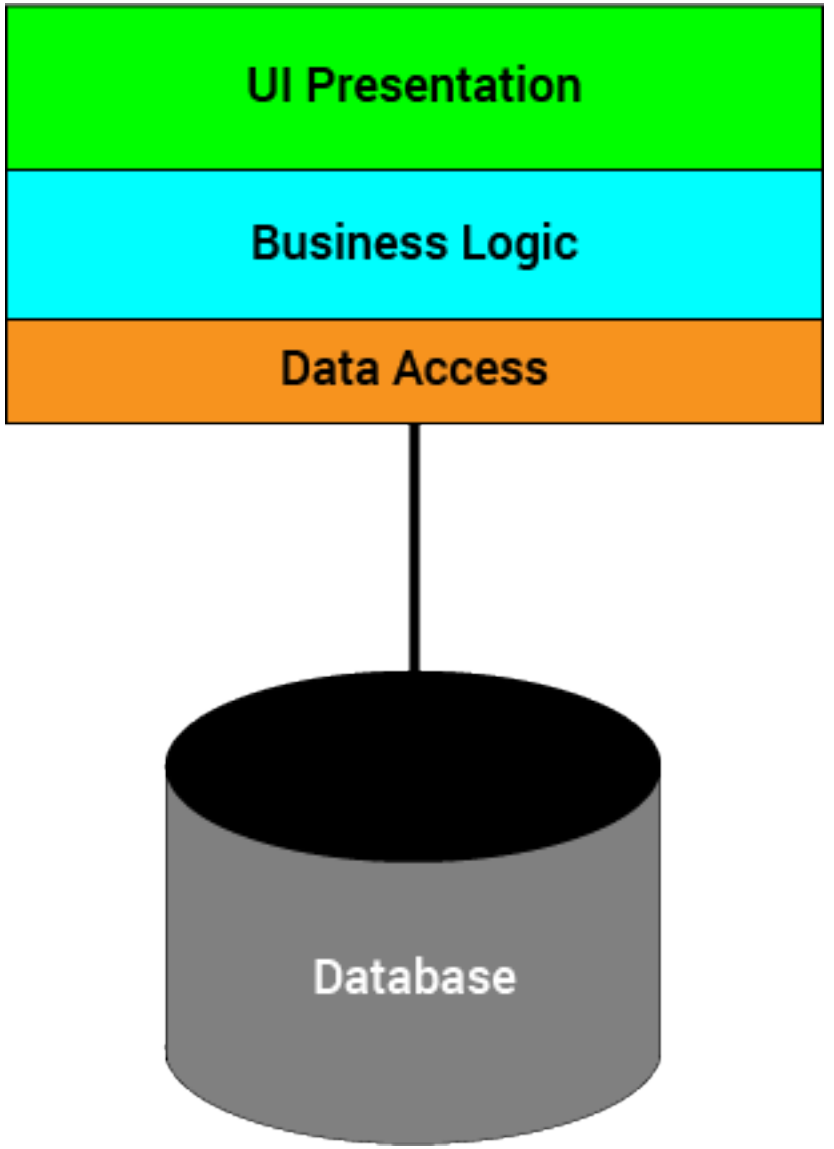


Business Analogy

- Start-up businesses are monoliths
- There are a few people who do everything
- The enterprise is small enough that this model works
  - It's actually counterproductive to have a highly structured departmental organization with just a few employees
- Early versions of applications are similar
  - Simple enough that all of the code is manageable
  - Single code base for the whole app
  - Flat or minimal architectural structure

Modular Monolith Application

- Code is modularized
- Organized along job descriptions
  - Front end dev has their modules
  - Programmers have their modules
  - Data engineers have their modules
- Shows up historically as a n-tier architecture

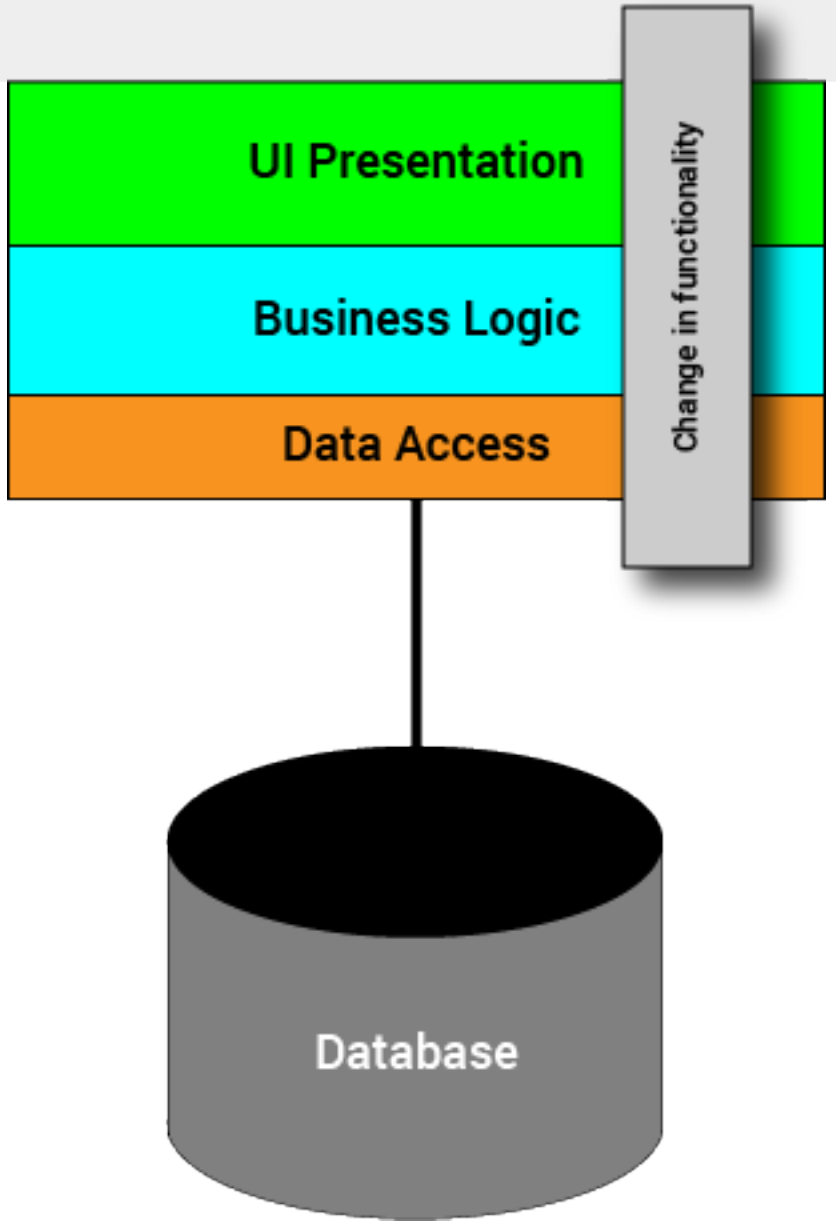


Business Analogy

- As the start-up grows, it adds more people
- The original organization starts to become ineffective
  - The entrepreneurial “all hands-on deck” model doesn’t scale well
  - Processes become chaotic
  - Difficult to manage
  - Productivity stalls
- At some point the business must modularize
  - Usually by creating specialized departments
  - Accounting, sales, HR, etc.

Modular Monolith Application

- The application is still a monolith
- A change in functionality
  - Affects each layer because related changes have to be made in each layer
- Interacts with the data model
  - The data model may constrain what changes can be made
  - Changing the data model might break other parts of the app
- The modules and the database often show high coupling, due to how the modules are defined



Pros and Cons of Monoliths

Pros

- Simple to develop
- Simple to test
- Simple to deploy
- Simple to scale horizontally
  - Run multiple copies behind a load balancer
- Although persistence is problematic when using horizontal scaling

Cons

- The size of the application can slow down the start-up time
- The entire application must be redeployed on each update
- Monolithic applications can also be challenging to scale vertically
- Reliability – easy to crash the whole application
- Difficult to migrate to new technologies

Legacy Monolithic Issues

- Codebase is enormous
  - Little or no documentation
  - Coupled to legacy technology
- E.g. Internal Revenue Service
  - Running 1960s vintage code
  - Application highly complex
  - Attempts to replace it have failed
  - Over $15 billion so far
- The IRS is not unique
  - Migrating legacy monolith to a modern monolith is not an option

Scaling in Systems

- Scaling is an increase in size or quantity along some dimension
- Can take place in the development or operations space
- Development scaling
  - Increase in complexity, functionality or volume of code
  - These dimensions are often related
  - Business analogy is a company increasing the range of services and products they offer or expanding into different markets (like a Canadian company expanding into Europe)
- Operational scaling
  - Increase in the amount of activity of a system
  - Throughput, load, transaction time, simultaneous users, etc
- Traditional monoliths tend to be scalable only to a limited degree
  - There is a certain level of complexity after which they become unmaintainable

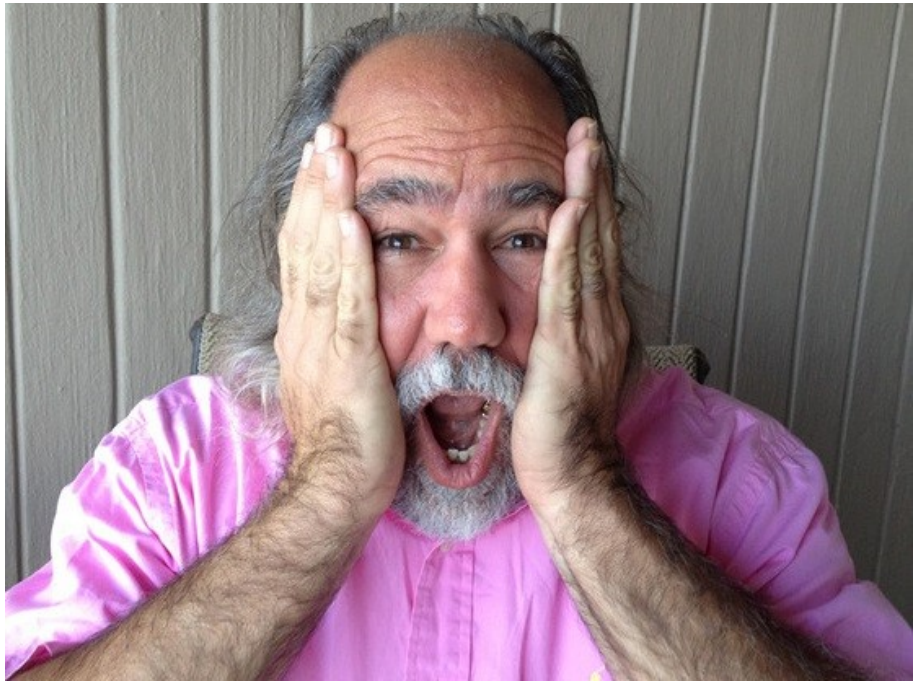
Mission Critical Industrial Strength Software

Mission critical software tends to have a long lifespan, and over time, many users come to depend on their proper functioning. In fact, the organization becomes so dependent on the software that it can no longer function in its absence. At this point, we can say the software has become industrial-strength.

The distinguishing characteristic of industrial strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all of the subtleties of its design.

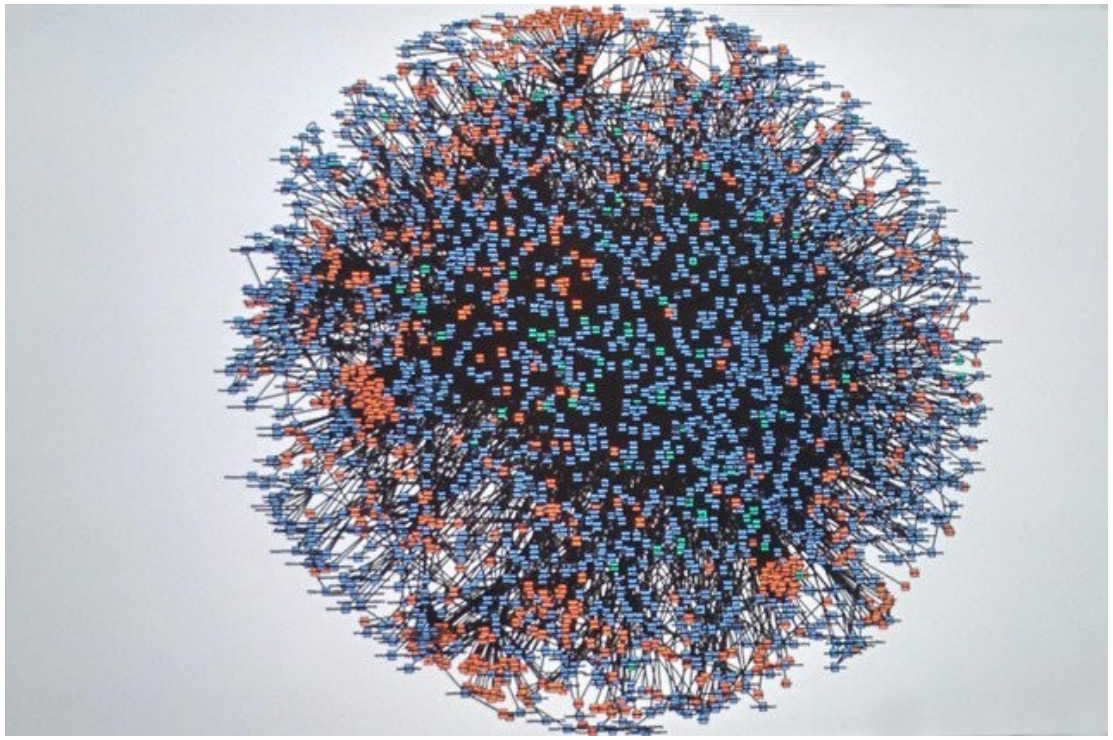
Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software sys-- tems. By 'essential' we mean that we may master this complex- ity, but we can never make it go away.

Grady Booch



Operational Complexity

- Modern applications have to deal with
  - Petabytes of streaming data
  - Billions of transactions
  - Mission critical fault tolerance
- Non-functional requirements
  - Throughput, response time, loading, stress
  - Disaster recovery, transaction time
- Results in a “death star” architecture
  - Image: Amazon in 2008
  - The complexity of the operational architecture is now industrial strength



IT Failures and Complexity

The United States is losing almost as much money per year to IT failure as it did to the [2008] financial meltdown. However the financial meltdown was presumably a onetime affair. The cost of IT failure is paid year after year, with no end in sight. These numbers are bad enough, but the news gets worse. According to the 2009 US Budget [02], the failure rate is increasing at the rate of around 15% per year.

Is there a primary cause of these IT failures? If so, what is it?.... The almost certain culprit is complexity.... Complexity seems to track nicely to system failure.

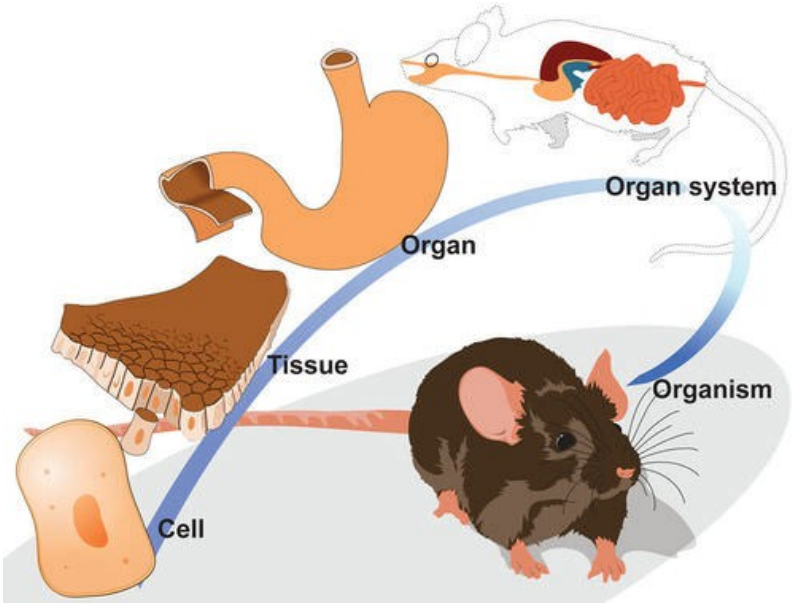
Once we understand how complex some of our systems are, we understand why they have such high failure rates.

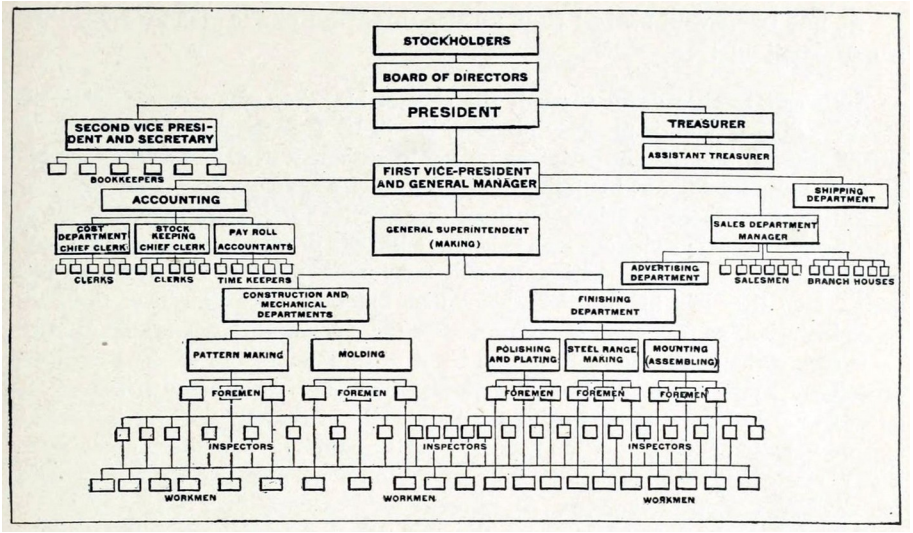
We are not good at designing highly complex systems. That is the bad news. But we are very good at architecting simple systems. So all we need is a process for making the systems simple in the first place.

Roger Sessions

Natural Systems Organization

- Naturally occurring systems scale successfully
  - Biological processes, social organization etc.
  - They all show similarities in organization regardless of domain
- Natural occurring systems tend to be
  - Recursive in structure
  - Hierarchical or layered
  - Modular at each layer
  - Loosely coupled and highly cohesive





Operational Complexity in Practice

- In production, systems have to scale operationally
- Consider a diner
  - During the lunch and dinner hour rush, the number of servers and cooks have to scale up
  - There has to be clear boundaries on what each role does
  - During non-peak hours, the amount of staff can be scaled down

Operational Complexity in Practice

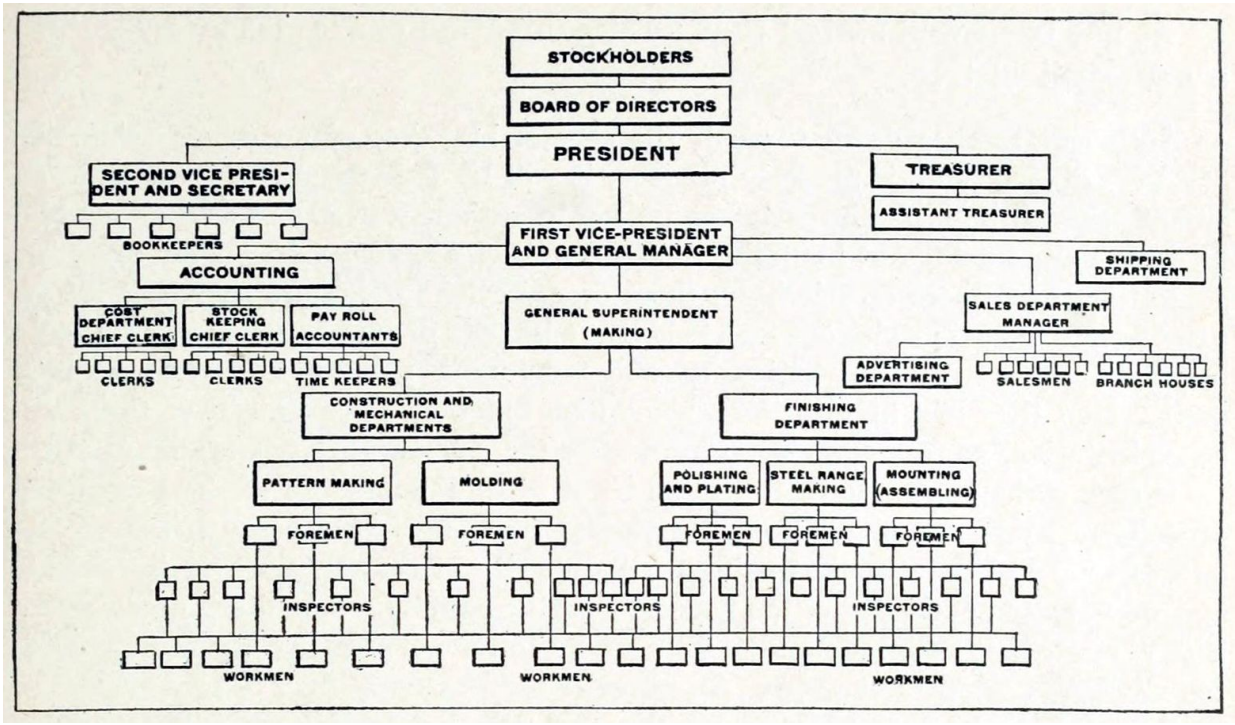
- Operations are specialized
- Tasks are broken down into sub-tasks, and sub-tasks broken down into sub-sub-tasks, and so on
- At some point
  - Specialized systems or agents perform an individual sub-task
  - These are all coordinated
- For example, the specialized positions on a sports team

Complex Systems

Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached

Courtois

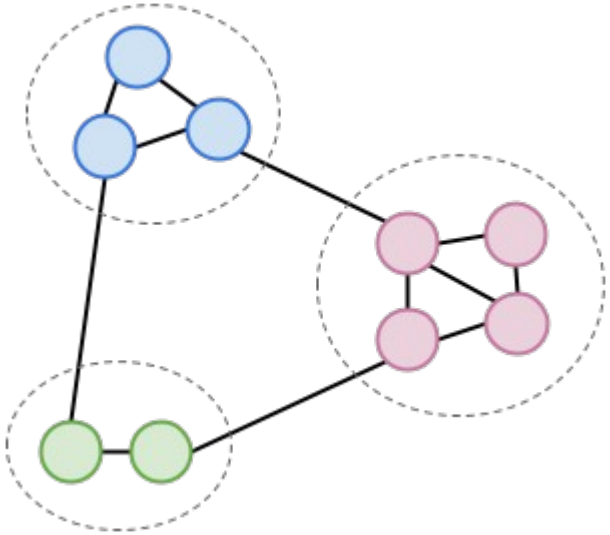
On Time and Space Decomposition of Complex Structures
Communications of the ACM, 1985, 28(6)



Complex Systems

Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high frequency dynamics of the components – involving the the internal structure of the components – from the low frequency dynamics – involving the interaction among components

Simeon



Complex Systems

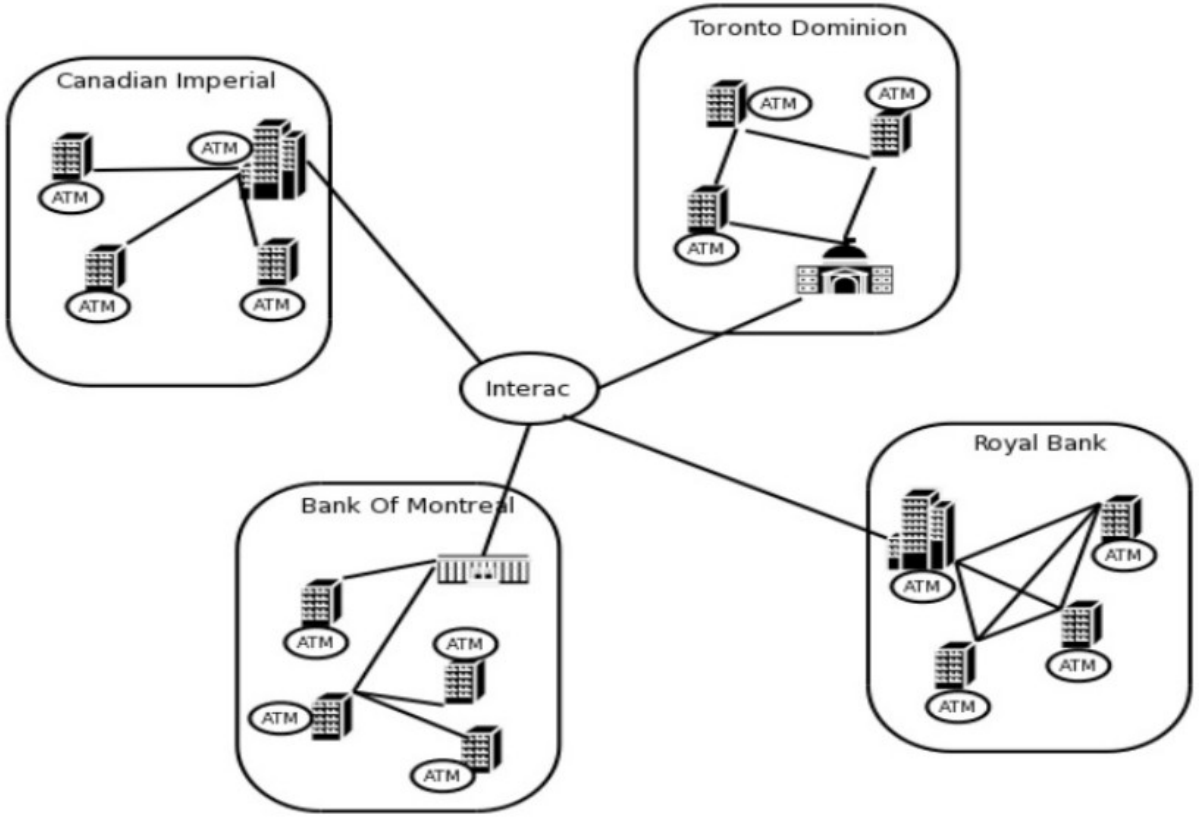
Hierarchical systems are usually composed of only a few different kinds of sub-systems in various combinations and arrangements

Simeon

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and can never be patched up to make it work. You have to start over, beginning with a simple working system.

John Gall

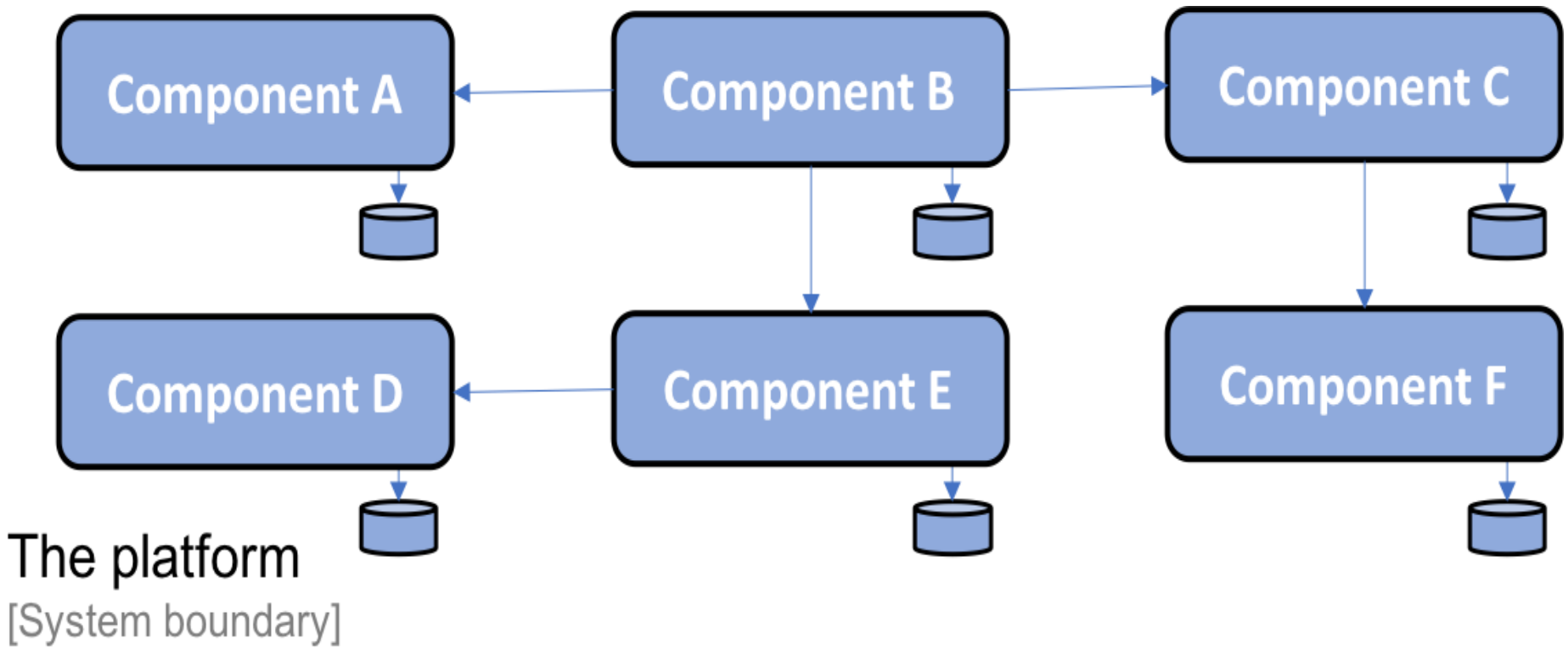
Systemantics: How Systems Really Work and How They Fail
1975



Microservices

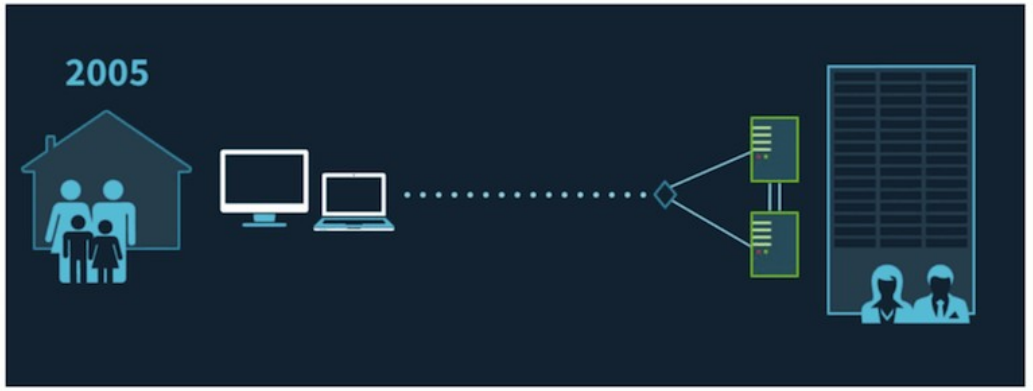
"The Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

Microservices

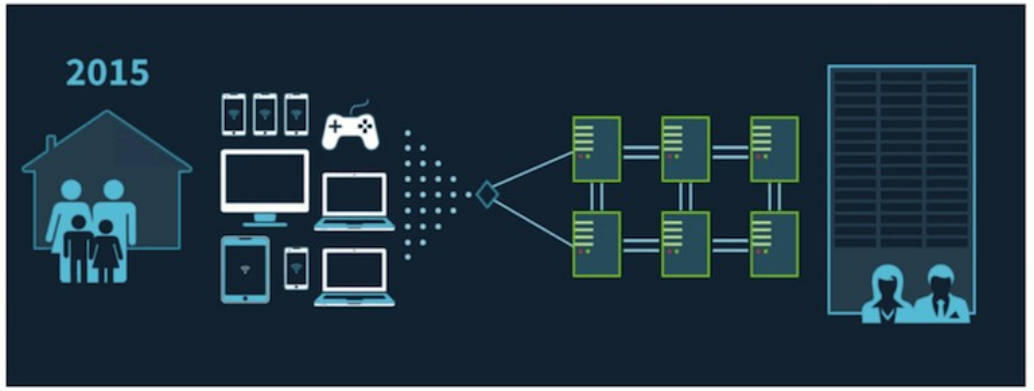


Microservices

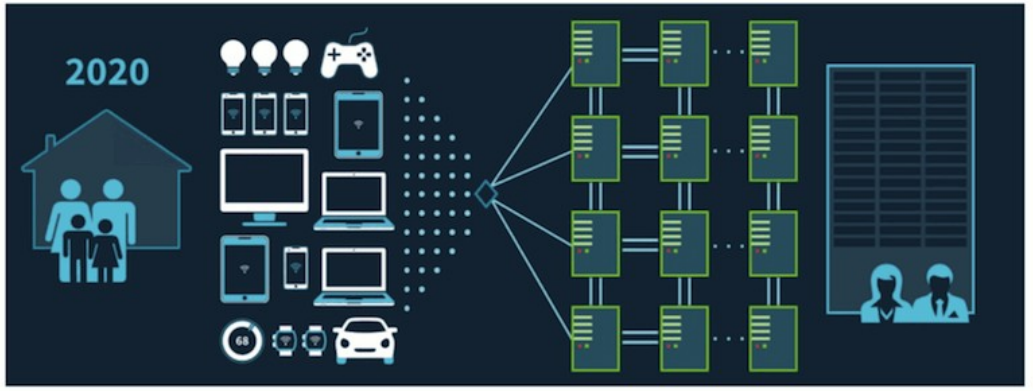
2005 ARCHITECTURE



2015 ARCHITECTURE



2020 ARCHITECTURE



THE WORLD BY 2020

» 4 billion connected people

» 25+ million apps

» 25+ billion embedded systems

» 40 zettabytes (40 trillion gigabytes)

» 5,200 GB of data for every person on Earth

Microservices Drivers

Modeling Microservices

- A core principle of microservices is that they model a business domain or a domain of activity
- These domains are often complex and wide ranging in scope
- Very often, they have evolved a domain structure that is
  - A hierarchy of sub-domain layers
  - The structure is recursive
  - Each layer is made up of modular components
  - The components work together by sending well-defined messages
  - Messages are sent through interfaces
  - Components are cohesive and loosely coupled
- This is a sort of idealized version of services “in the wild”
  - There are many factors that can make this organization dysfunctional

In Real Life

- A hospital provides a health care service
- Made up of individual microservices
  - Laboratory
  - X-Ray
  - Pharmacy
  - Medical Staffing
- Each microservice:
  - Specializes in a specific domain activity
  - Only that microservice performs that domain activity (the pharmacy doesn't do x-rays for example)
  - Each microservice operates autonomously

In Real Life

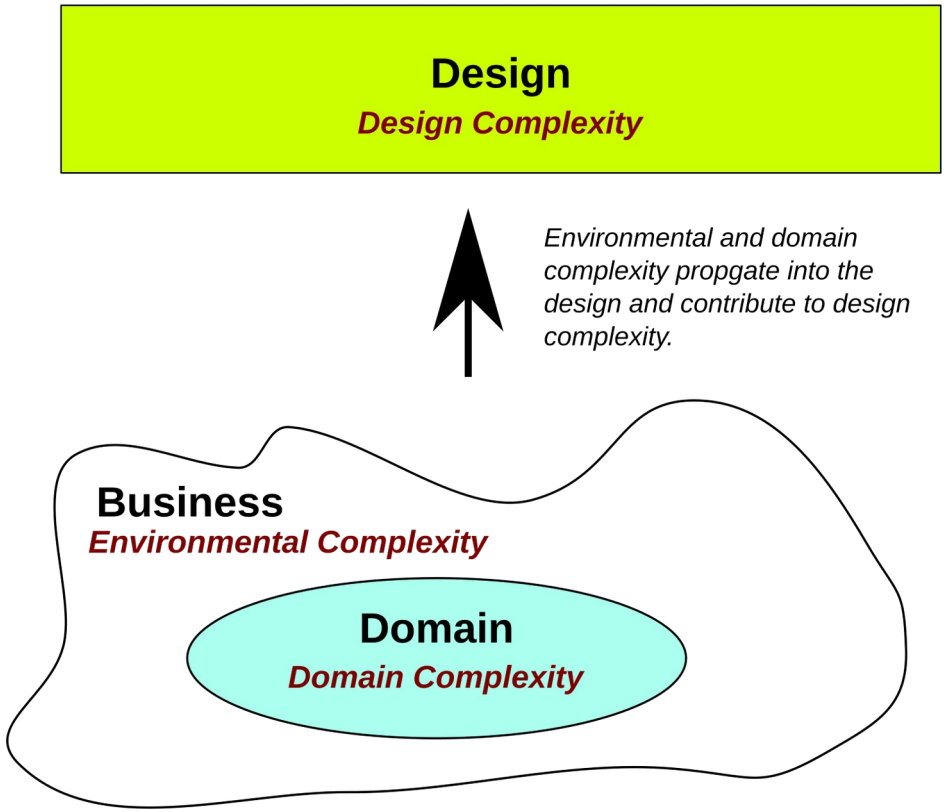
- Microservices request services from each other
  - They do not need to know the internal workings of the other microservice
- Requests are made through interfaces
  - Called APIs in software engineering
  - These are often paper based in the real world
  - Requisitions and official forms and paperwork used to make requests of a service
  - Response to a request is often a report of some kind
- Microservices make sense intuitively

COVID-19 VIRUS LABORATORY TEST REQUEST FORM[1]

Submitter information			
NAME OF SUBMITTING HOSPITAL, LABORATORY, or OTHER FACILITY*			
Physician			
Address			
Phone number			
Case definition:[2]	☐ Suspected case ☐ Probable case		
Patient info			
First name		Last name	
Patient ID number		Date of Birth	Age:
Address		Sex	☐ Male ☐ Female ☐ Unknown
Phone number			
Specimen information			
Type	☐ Nasopharyngeal and oropharyngeal swab ☐ Bronchoalveolar lavage ☐ Endotracheal aspirate ☐ Nasopharyngeal aspirate ☐ Nasal wash ☐ Sputum ☐ Lung tissue ☐ Serum ☐ Whole blood ☐ Urine ☐ Stool ☐ Other:		
All specimens collected should be regarded as potentially infectious and you must contact the reference laboratory before sending samples. All samples must be sent in accordance with category B transport requirements.			
Please tick the box if your clinical sample is post mortem ☐			
Date of collection		Time of collection	
Priority status			
Clinical details			
Date of symptom onset:			
Has the patient had a recent history of travelling to an affected area?	☐ Yes ☐ No	Country Return date	
Has the patient had contact with a confirmed case?		☐ Yes ☐ No ☐ Unknown ☐ Other exposure:	
Additional Comments			

The Complexity Problem

- There are three kinds of complexity we have always been concerned with
  - **Domain complexity** - trying to automate a domain that is inherently complex
  - **Design complexity** - Designs that are not elegant (simple and effective)
  - **Environmental complexity** – Resulting from a disorganized and poorly functioning business
- We can eliminate a lot of design complexity with good design and engineering practices
- We cannot make domain complexity go away, we can only manage it



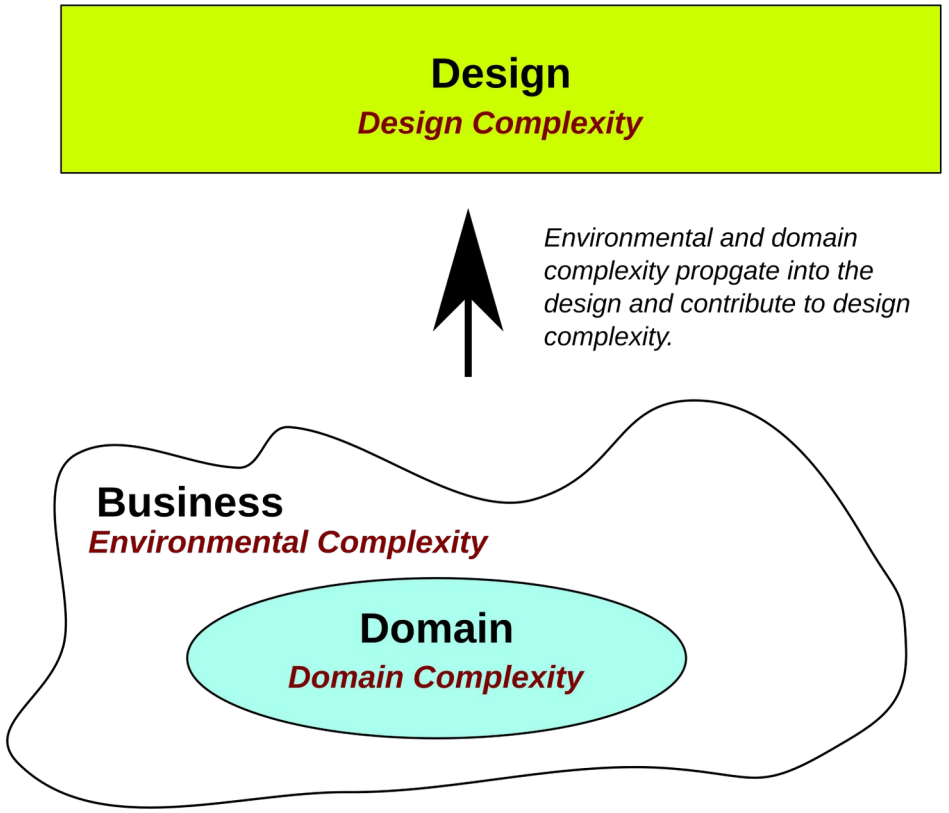
Controlling complexity is the essence of computer programming. Brian Kernighan

Propagation of Complexity

- Not properly managed, environmental and domain complexity directly propagate into design complexity
- We should not confuse environmental complexity with domain complexity
  - We cannot deal with it, so we have to ignore it in understanding the domain

The most important single aspect of software development is to be clear about what you are trying to build.
Edsger Dijkstra

First, solve the problem. Then, write the code
Donald Knuth



Microservices Essential Principles

- Single Responsibility
- Discrete
- Carries its own data
- Transportable
- Ephemeral

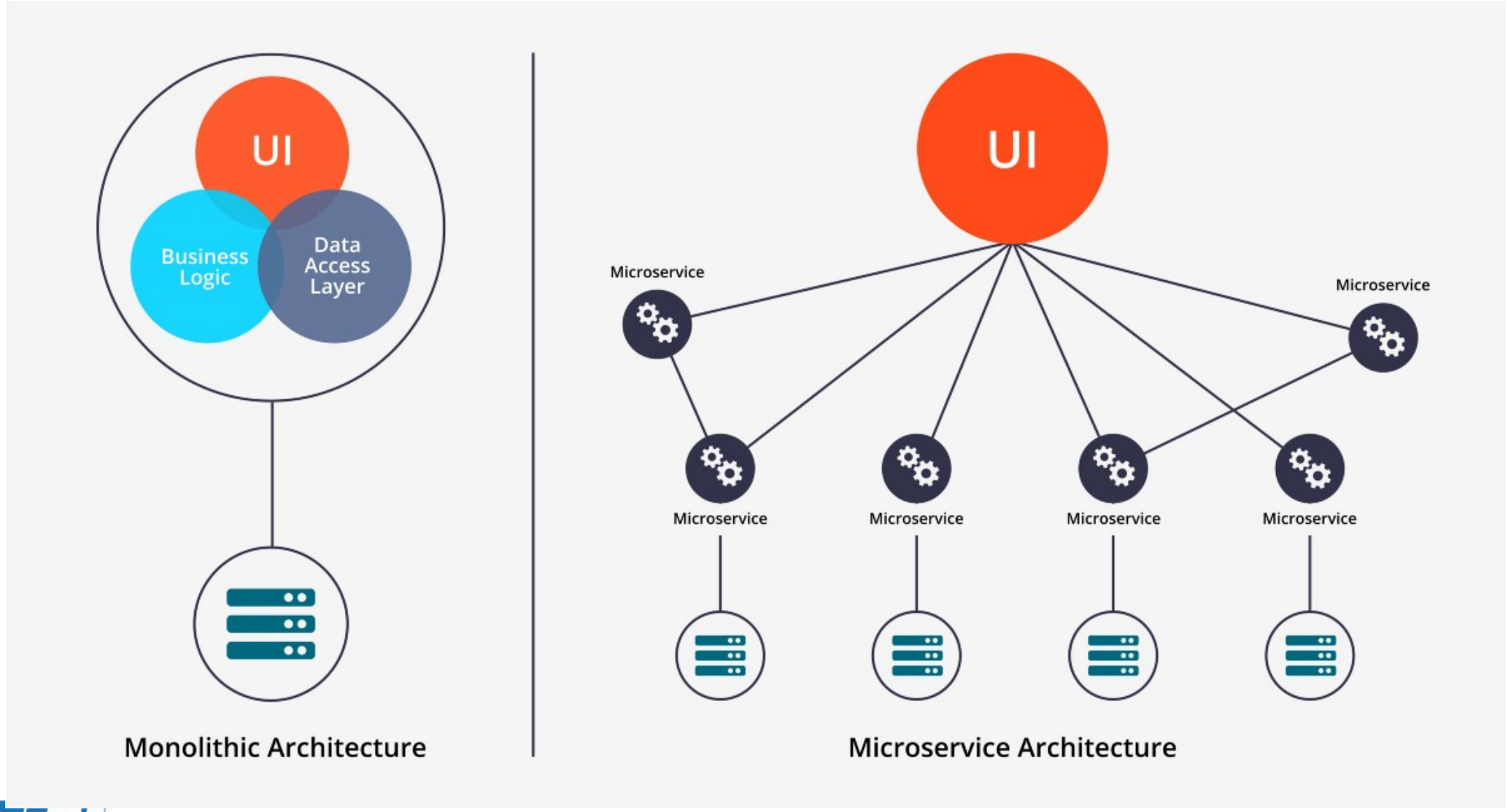
Microservices Core Concepts

- Microservices are small, independent, composable services
- Each service can be accessed by way of a well-known API format
  - Like REST, GraphQL, gRPC or in response to some event notifications
- Breaks large business processes into basic cohesive components
  - These basic process actions are implemented as microservices
  - The business process is executed by making calls to these microservices
- Each microservice provides one cohesive service
  - Microservices do not exist in isolation
  - They are part of a larger organization framework

Microservices Core Concepts

- Microservices coordinate with other microservices to accomplish the tasks normally handled by a monolithic application
- Microservices communicate with each other synchronously or asynchronously
- Microservices make application components easier to develop and maintain
- BUT A LOT HARDER TO MANAGE

Monolith to Microservice



Characteristics of Microservice Deployments

- Light-weight, independent, and loosely-coupled
- Each has its own codebase
- Responsible for a single unit of business functionality
- Uses the best technology stack for its use cases
- Has its own DevOps plan for test, release, deploy, scale, integrate, and maintain independent of other services
- Deployed in a self-contained environment
- Communicates with other services by using well-defined APIs and simple protocols like REST over HTTP
- Responsible for persisting its own data and keeping external state

Business Drivers

- Business Agility
  - Speed of change is faster with a more modular architecture
  - React quicker to feature and enhancement requirements
  - Easy integration with new technology, processes (Mobile, Cloud, Continuous DevOps)
- Composability
  - Allows for reuse of capability and functionality
  - Allows for integration with other internal and external services
  - Reduces technical debt and replication
- Robustness and Migration
  - A single microservice failure does not bring down an application
  - The business functionality is always available
  - Updated functionality can be rolled out in a controlled and safe manner
  - Smaller components reduces risk exposure

Business Drivers

- Scalability
  - Services can scale up to handle peak loads, or down to save costs
- Enable Polyglot Development
  - Different technologies can be used for different services
  - No lock-in to a single technology across the whole business

Pros and Cons of Microservices

Pros

- Easy to develop individually
- Easy to understand individually
- Easy to deploy individually
- Easy to monitor each service
- Flexible Release Schedule
- Use standard APIs (JSON, XML)

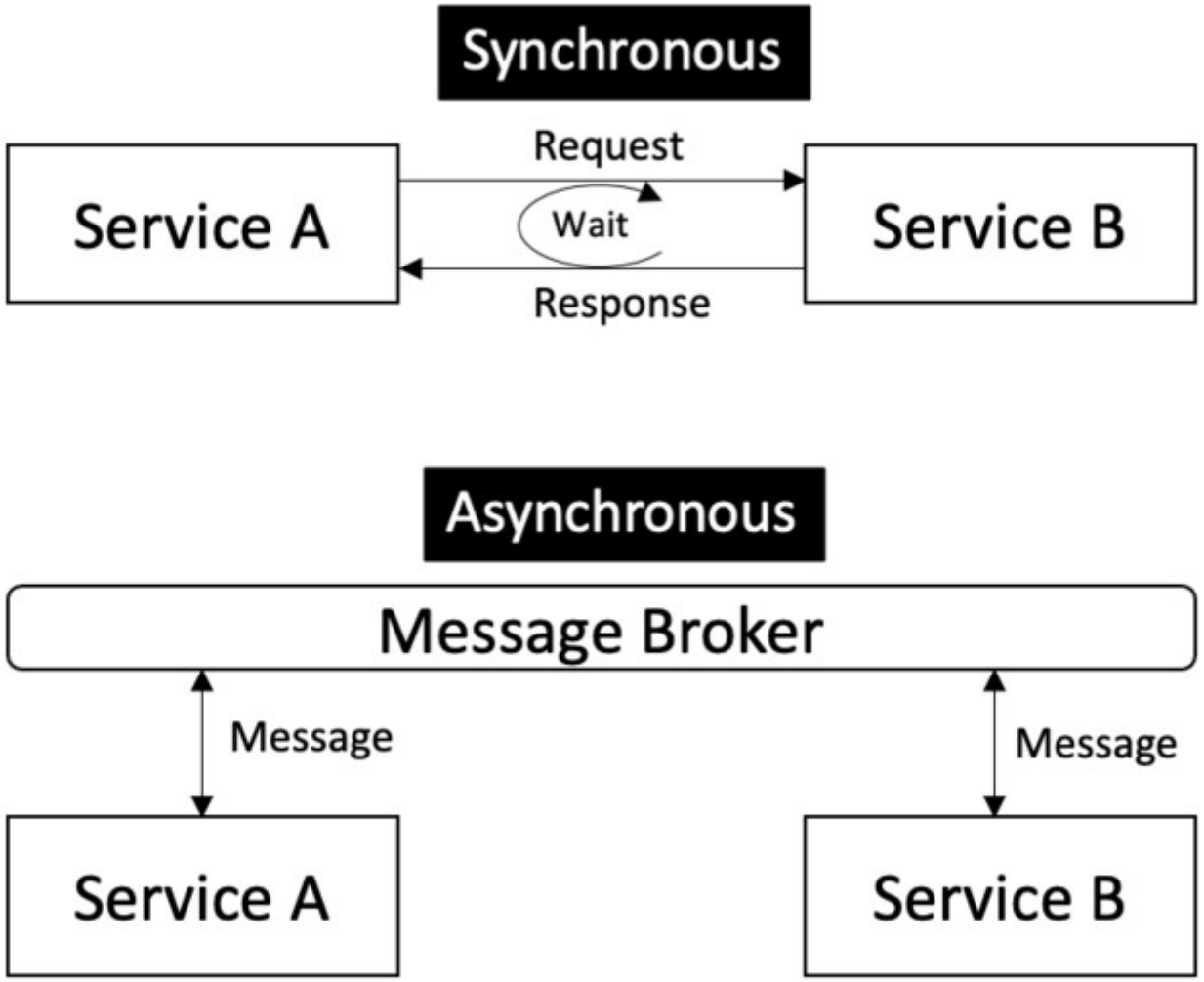
Cons

- Requires retooling
- Requires more deployments
- Requires translation (JSON, XML)
- Requires more monitoring
- Operations configuration can be very complex
- System can be very complex

Types of Microservices

- Primary distinction Request versus Event based
- Request based
  - A request is made of a service from a specific client
  - Some sort of response is expected by the client
- Event based
  - The service is responding to a series of events that are occurring somewhere
  - Events don’t expect a response
- These are often referred to as synchronous versus asynchronous
  - This is proving to be an inadequate classification

Basic Architectural Approaches



Request Based Microservice

- Generally part of a logic flow
- Implementation of some sort of scenario or process
- For example: Online purchase
- There is a clear workflow
  - User logs in and is taken to home screen
  - User searches for item and selects item
  - User places order
  - Payment is approved
  - Request for shipment is generated and sent to fulfillment
- Each step may be handled by a different microservice
  - Account services, payment service, catalog service, etc.

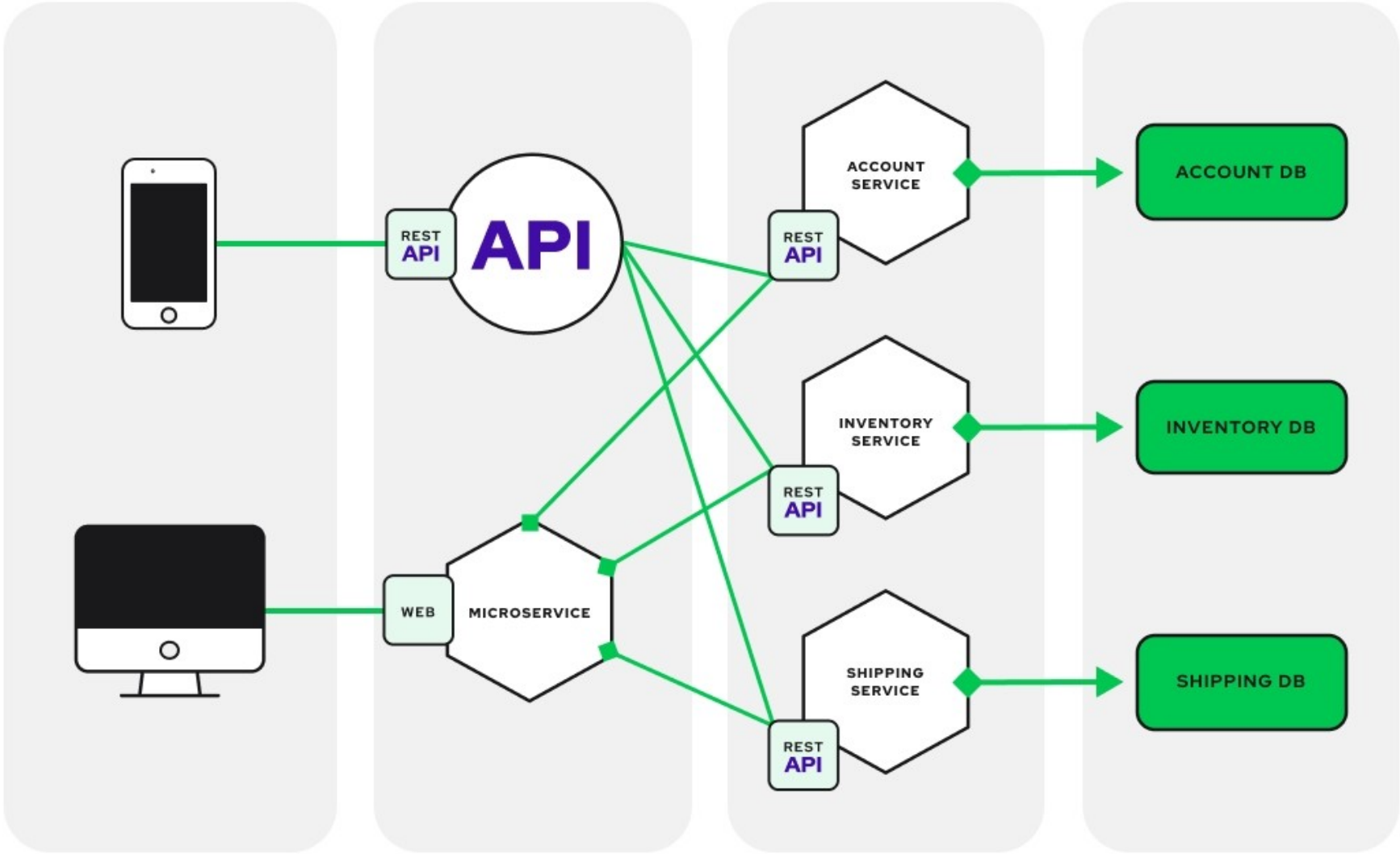
Request Based Microservice

- Often referred to as a synchronous service
- Each request requires a reply
- The service may “block” waiting for a reply
  - This is the more common use of the term synchronous
- However, request based microservices may be asynchronous
  - Often referred to as “fire and forget”
  - The process can continue while waiting for a reply
- In the sales example
  - The request for shipment can be made
  - But the sale ends without getting a confirmation of the shipment details
  - The confirmation can be sent later (maybe email) once the order is processed

A Subtle Distinction

- What do we mean by a reply?
- It could mean that we have the answer to our request
  - “Your payment has been successfully processed, here is your receipt”
  - “You have been authenticated to the system, access token enclosed”
- Or we just have the receipt of our request acknowledged
  - “Your order has been received, you will receive a confirmation email later”
  - “Your request has been received and you will receive a ticket number once it has been entered into the system”
- Request based microservices
  - Can be synchronous or asynchronous
  - But tend to be the better option when dealing with synchronous types of processing

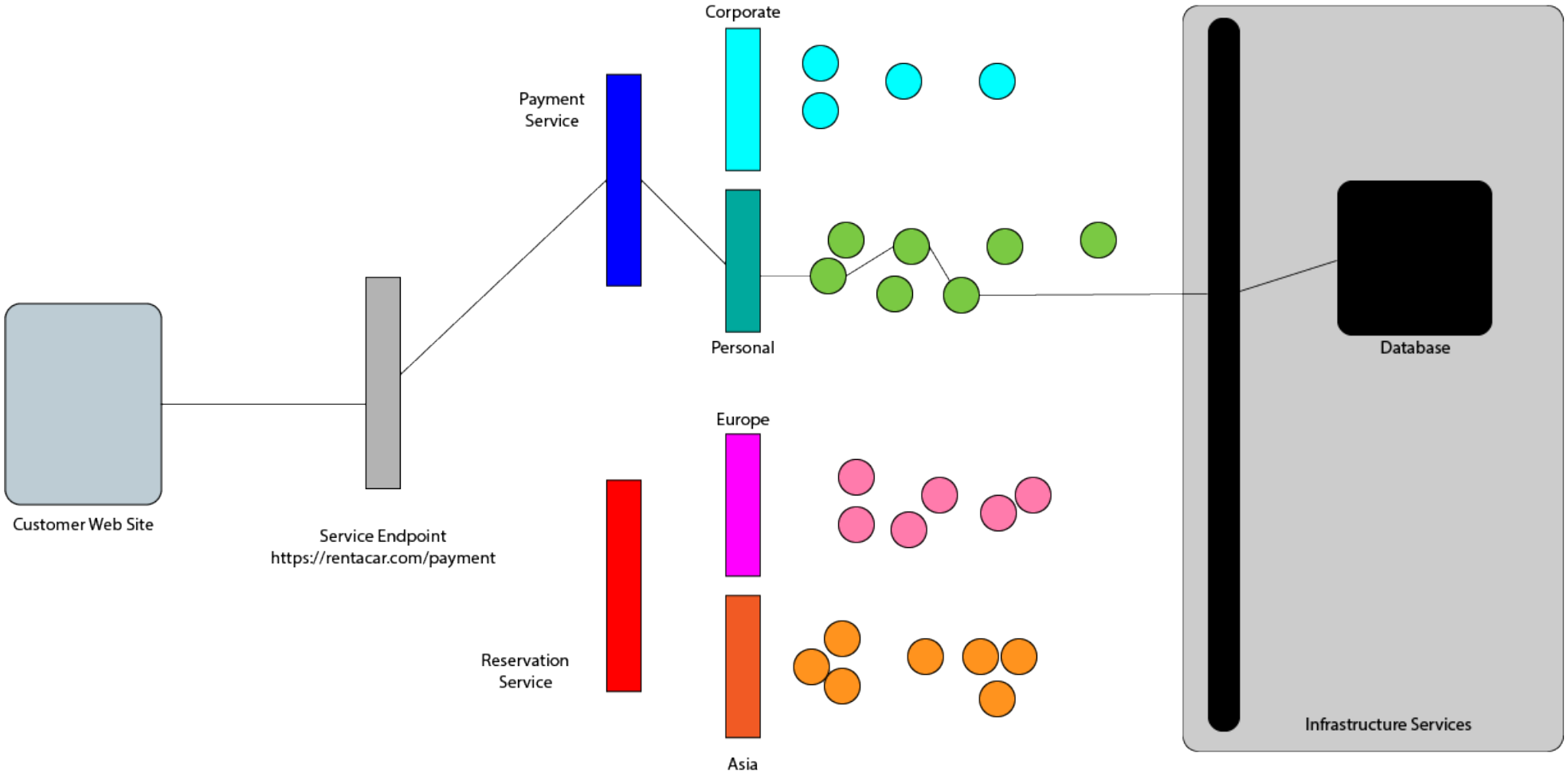
Request Based



Processing a Request

- Request arrives at an endpoint
- Depending on the service requested
  - The message is routed to an internal service that handles that type of request
  - That service routes the message to a running process to handle the process
  - The process retrieves any state data it needs from the data service
  - Request is processed
  - State data store is updated
  - Response returned to service that returns to the endpoint that received the request
- More than one process type may be involved
  - Processes are stateless and horizontally scale-able
  - The main challenge: how do you orchestrate the flow of messages?

Processing a Request

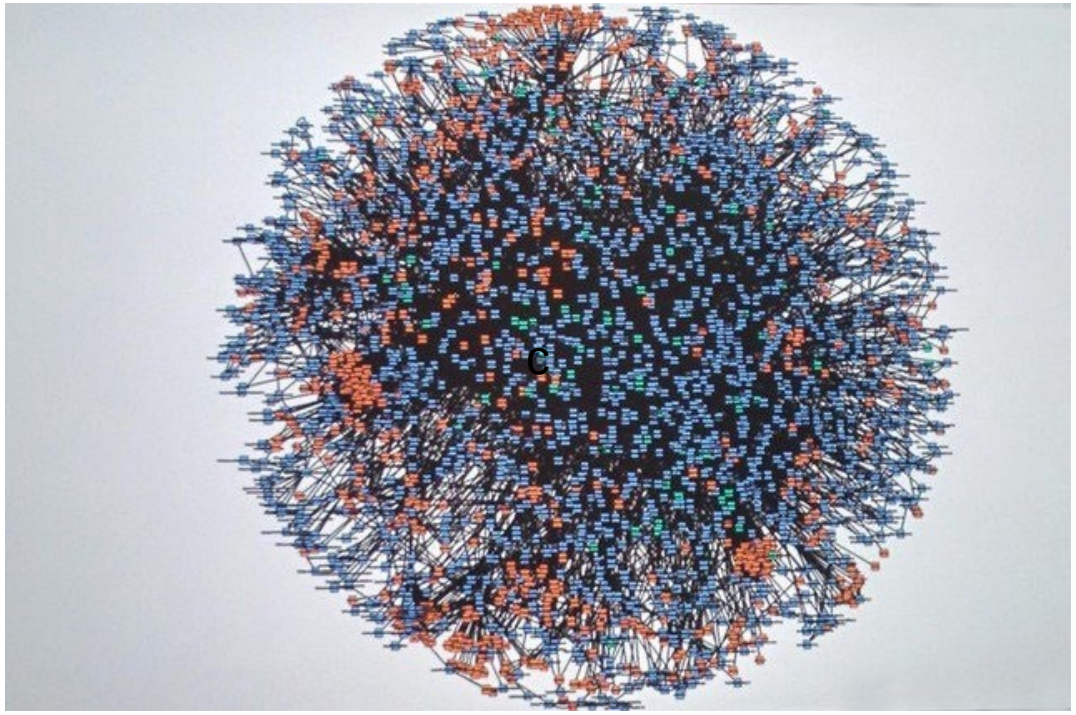


The Operations Challenge

- Processes are usually deployed in Docker or similar containers
- But we have to solve:
  - Coordinating the activity of possibly thousands of containers that need to work together
  - Creating and maintaining connections between containers
  - Ensure the whole system operates well enough to meet Service Level Agreements (SLAs)
- We need to deal with non-functional requirements
  - Loading, throughput, stress, response times
  - Disaster recovery
  - Security
- The lack of an effective way to do this was a major impediment to the deployment of microservice based applications

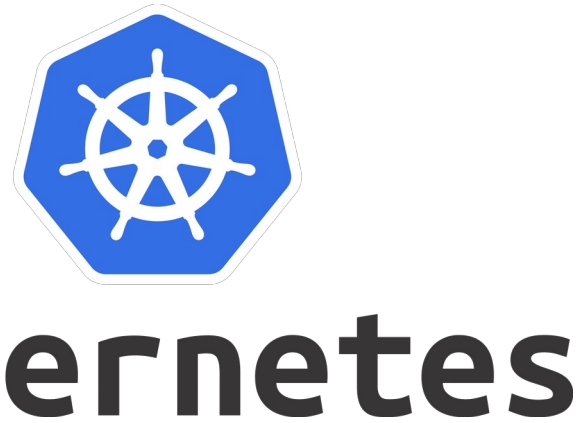
Site Reliability Engineering

- Practices designed to ensure large systems are operational
- Continuously checking for potential problems
- Manages a set of mitigation responses to react to problems
- Recent examples
  - Rogers Canada 2022 network failure
  - Facebook October 2021 upgrade failure
  - Check out risks.org
- As applications scale, this becomes increasingly difficult



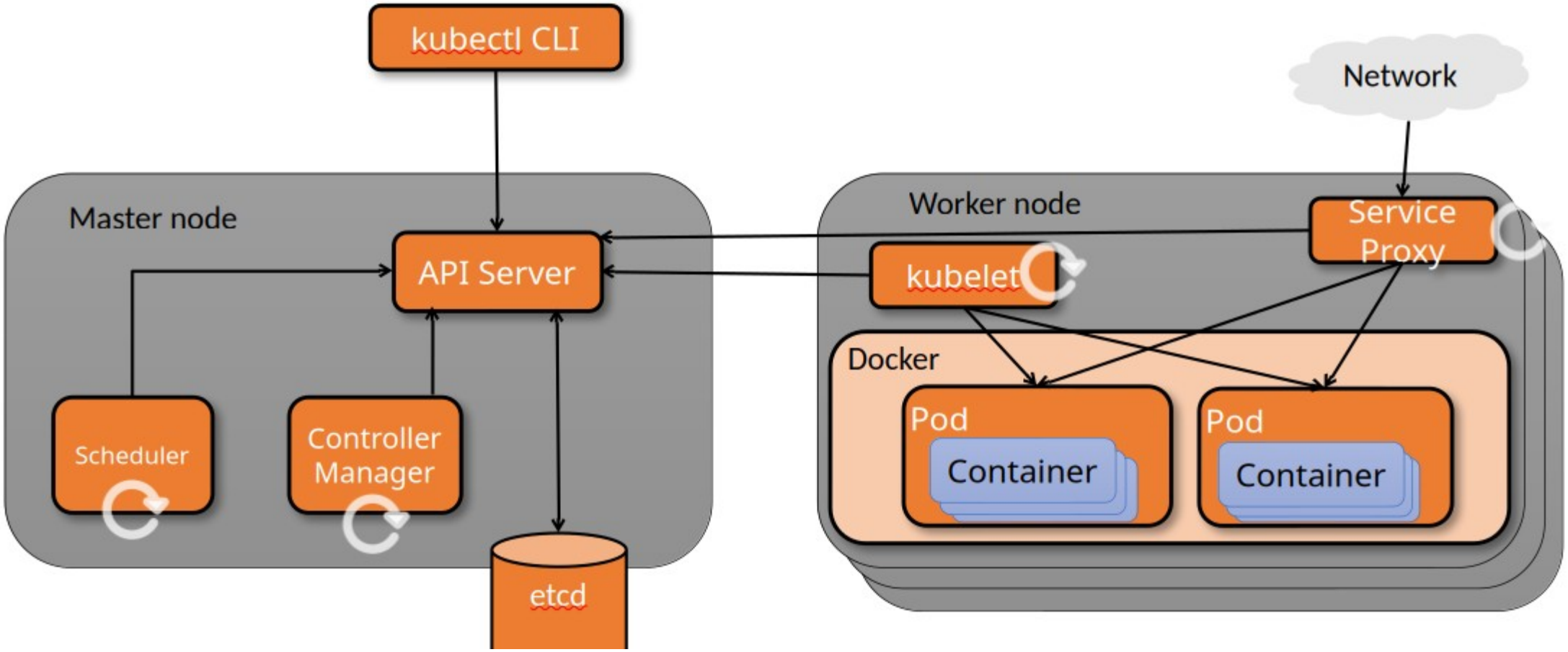
Kubernetes

- Kubernetes is a container orchestration manager
  - Not the only manager
  - Docker Swarm does the same
  - Kubernetes is “industrial strength”
- Orchestration:
  - Manages “clusters” of containers
  - Provides service discovery
  - Manages scaling and failover
  - Works at the Ops level
- Infrastructure as Code

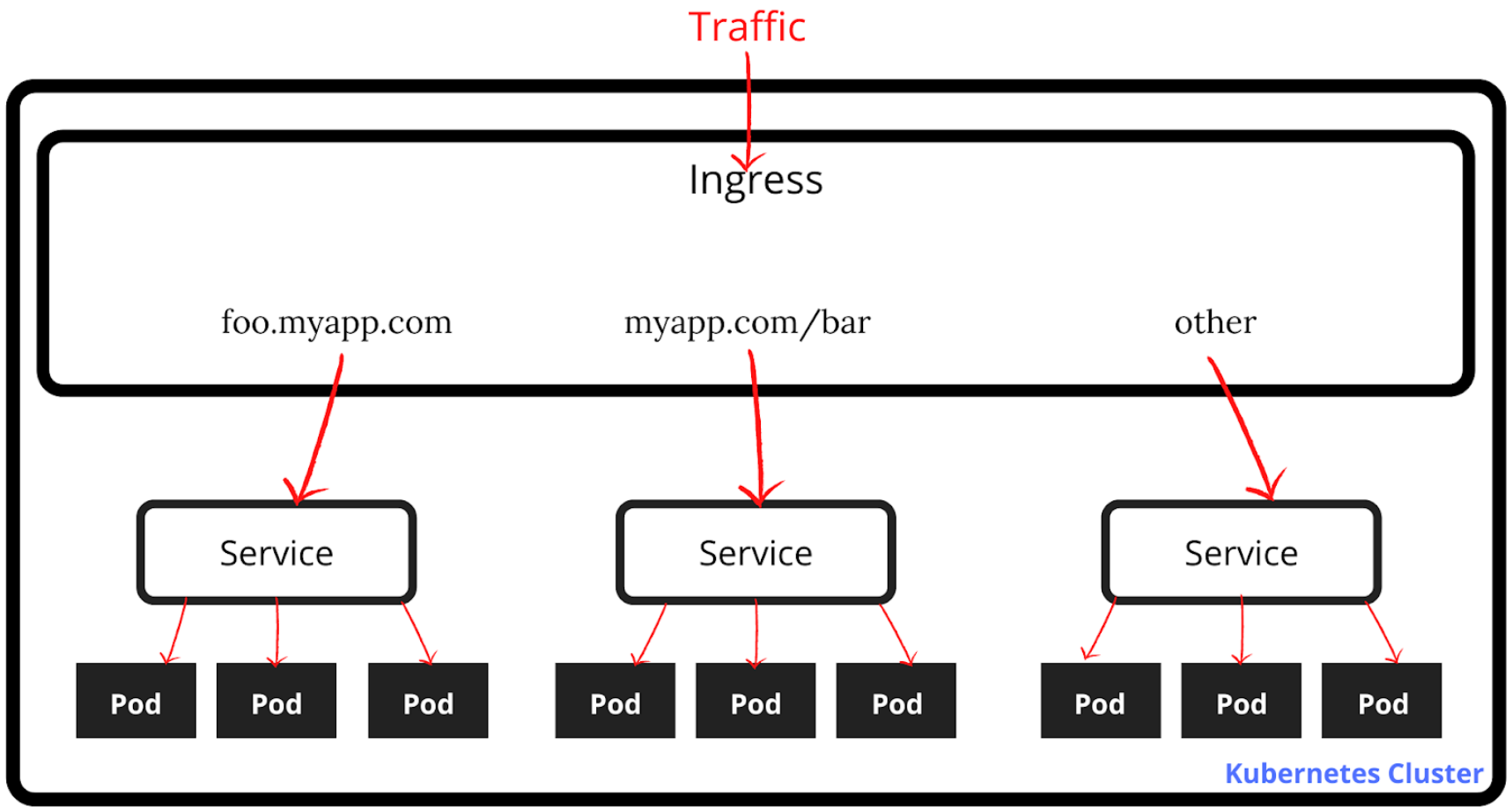


Kubernetes Architecture

- Kubernetes nodes can be physical hosts or VM's running a container-friendly Linux



Kubernetes Services



Common Deployment

- Modern microservices that are request based tend to use a common set of tools
- Containers
  - Small, self contained units of functionality
  - Designed to follow good design principles: cohesive, supple, etc
  - Dynamically horizontally scalable
- Pods
  - Wrapper around a container or set of containers to create a usable microservice
  - Adds the necessary interface and infrastructure so the containers can be managed by the orchestration tool
- Orchestration
  - A control system that manages the pods and running environment
  - Routes incoming requests to the appropriate microservice

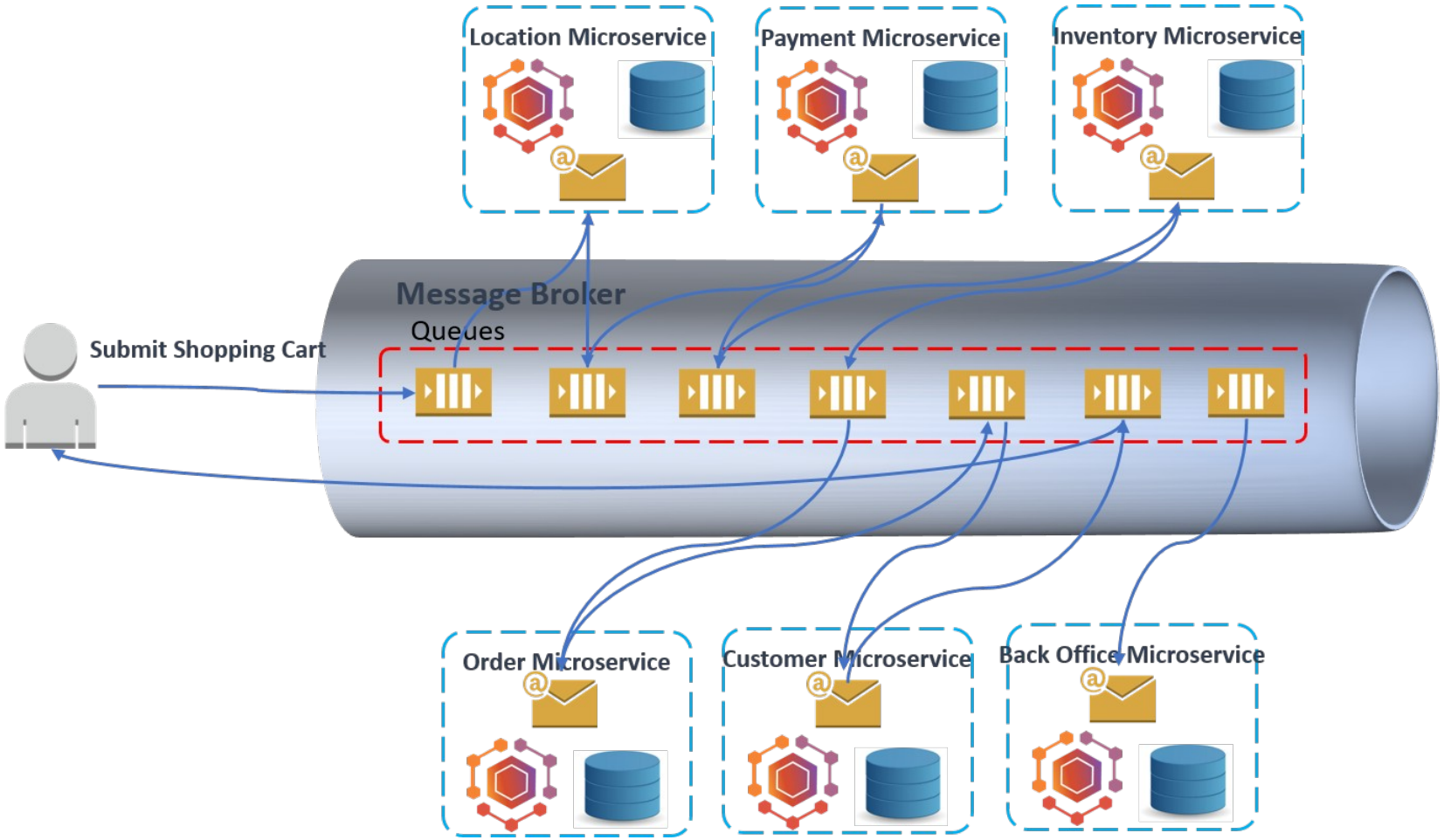
Event Based Microservice

- Responds to a stream of events
  - Asynchronous in design
- Events may require a response by the system
  - But response may or may not involve a reply to the event source
  - The response is often an event sent to another service
  - Architectural implementation of the Command and Chain of Responsibility design patterns
- Example: Heat sensor
  - A heat sensor in a power plant takes a heat reading
  - An event is sent to the monitoring microservice
  - Monitoring service creates new events in response
    - *Sends an event to the data recording service to log the data*
    - *If the reading is anomalous, it sends an event to an alert services*

Event Based Microservice

- Events may require a response by the system
  - But response may or may not involve a reply to the event source
  - The response is often an event sent to another services
- Typical use cases
  - Event Response
    - *Events might occur that require a response that is not a reply to the event source*
    - *Monitoring heat sensors in a nuclear reactor*
    - *The response is a reaction by the system to the event, not a reply to the sensor*
  - Event Streaming
    - *Events are just collected and processed for later use*
    - *Collecting GPS location of cell phones*
    - *Collecting point of sales data for data analytics*
    - *No reply or response required*

Event Based

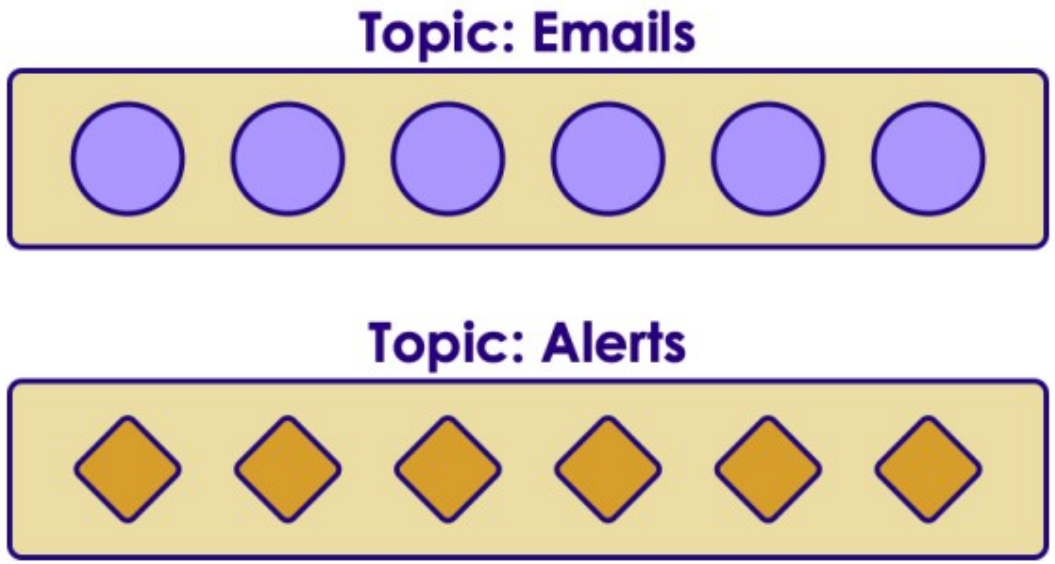

Location Microservice
Payment Microservice
Inventory Microservice
Message Broker
Queues
Submit Shopping Cart
Order Microservice
Customer Microservice
Back Office Microservice

Event Based Microservice

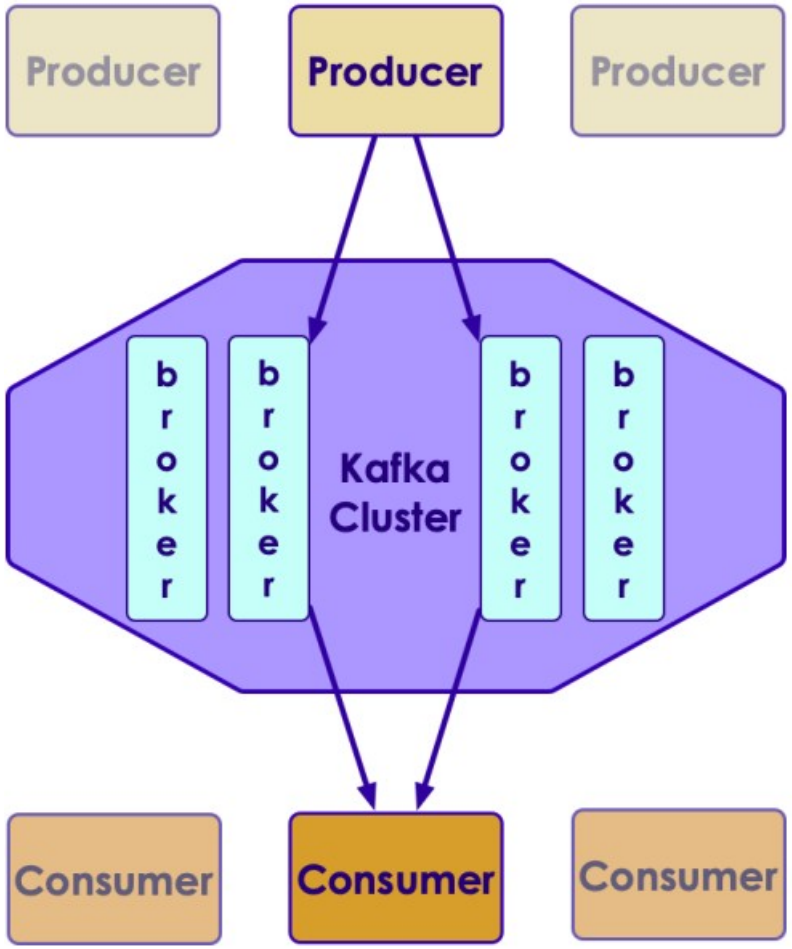
- Instead of messages, we think in terms of events
  - An event some data item of interest in the domain
- Instead of a cluster like Kubernetes, the main artifact is a stream or queue
  - Publishers put events onto the queue
  - Subscribers get events off of the queue
  - Generally referred to as the “pub-sub” model
- The primary technology used is Kafka
  - Acts as an asynchronous buffer between publishers and subscribers

Kafka Concepts

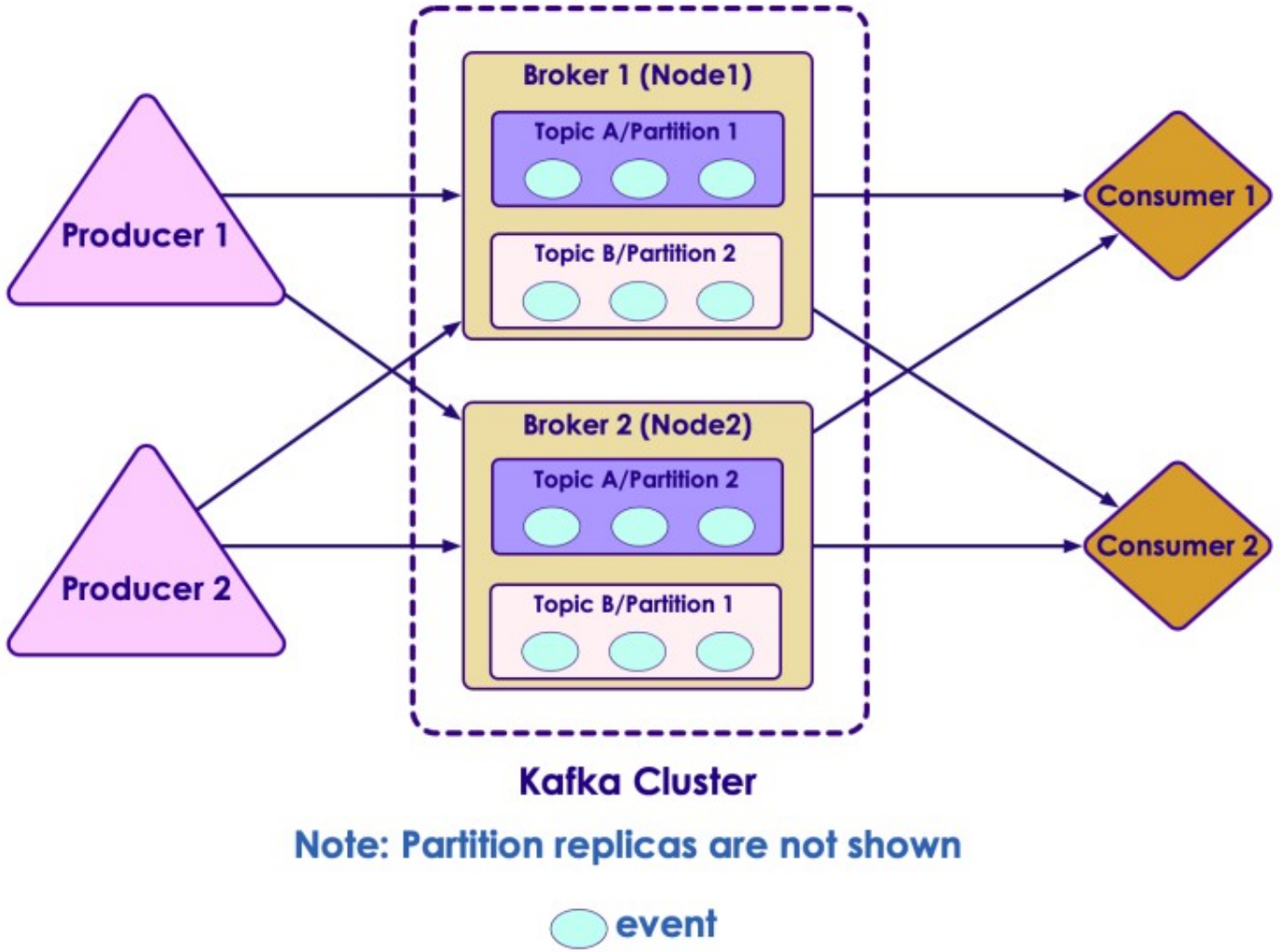
- In Kafka a basic unit of data is a 'message'
  - Message can be email / connection request / alert event
- Messages are stored in 'topics'
  - Topics are like 'queues'
  - Sample topics could be: emails / alerts



Kafka Architecture



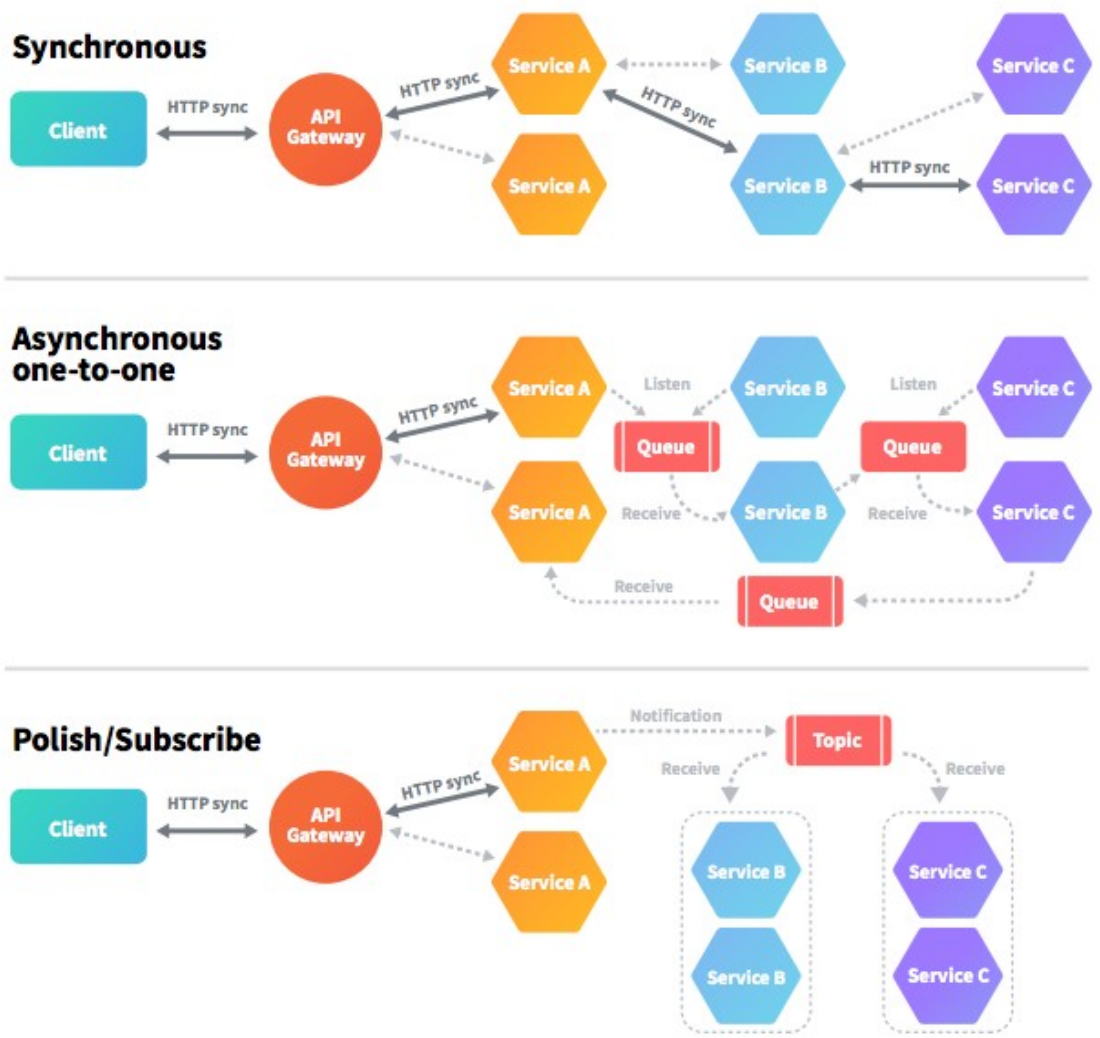
Kafka Architecture



Hybrid Microservice

- Generally, microservice architecture is hybrid
- Parts of the service are message based
- Parts of the service are event based
- Coupling
  - Message based are more tightly coupled – they require the message be sent to an endpoint
  - Event based are more loosely coupled because requests are buffered in a queue

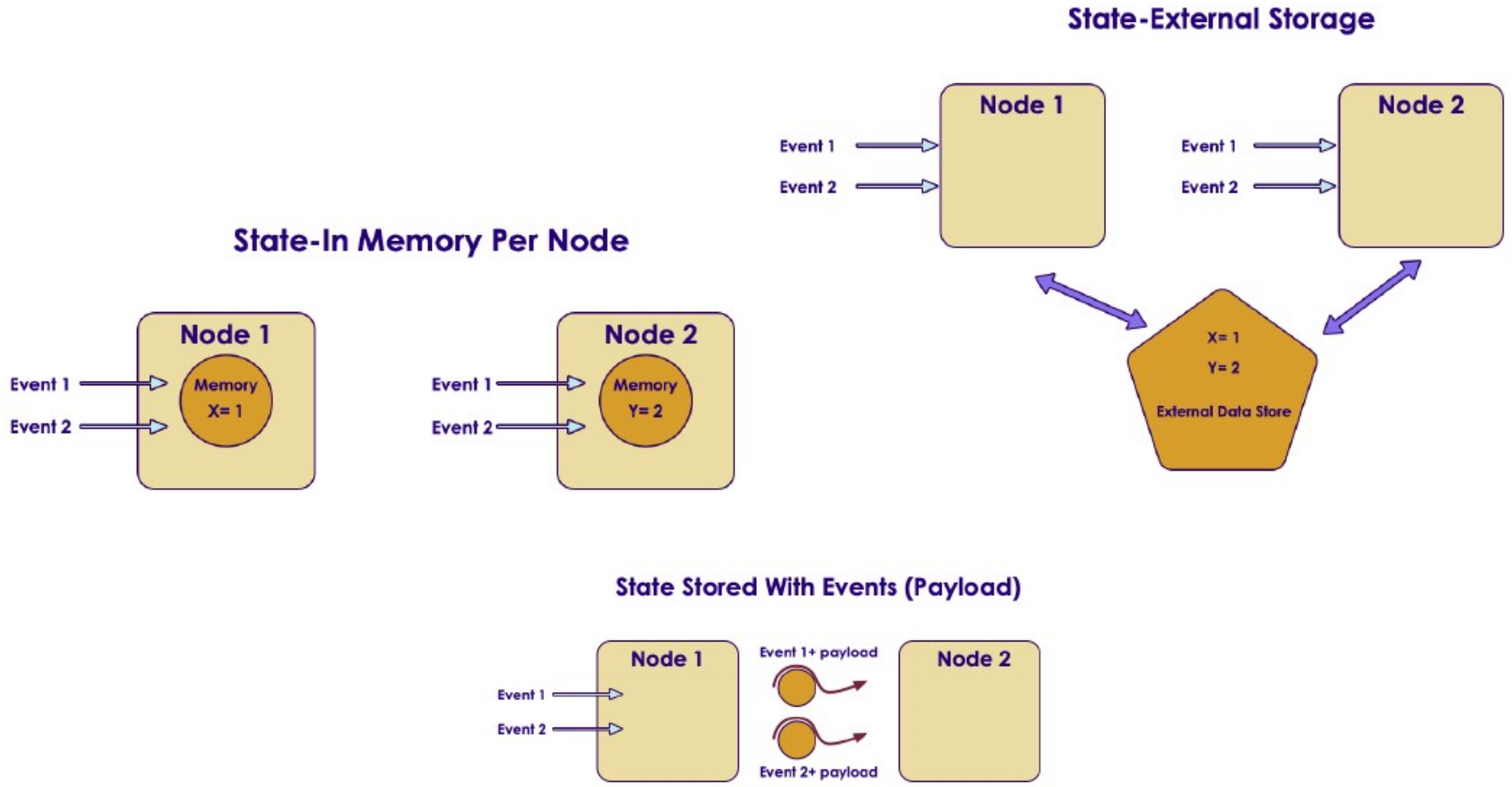
Hybrid Interaction Patterns



Design Consideration – Transactional State

- Does an incoming message require a synchronous response?
- Does the incoming message represent a request or an event?
  - Requests usually kick off a business process or use case
  - Events just happen “out there” and can be dealt with in isolation
  - Events tend to stream from event sources, requests originate from clients
- Do we need to maintain state?
  - Business processes often require tracking transactional state
  - Events are usually stateless
- What sort of scaling requirements do we have?
  - How we manage transactional state may depend on our scaling requirements

State Representation



The 12 Factor App Methodology

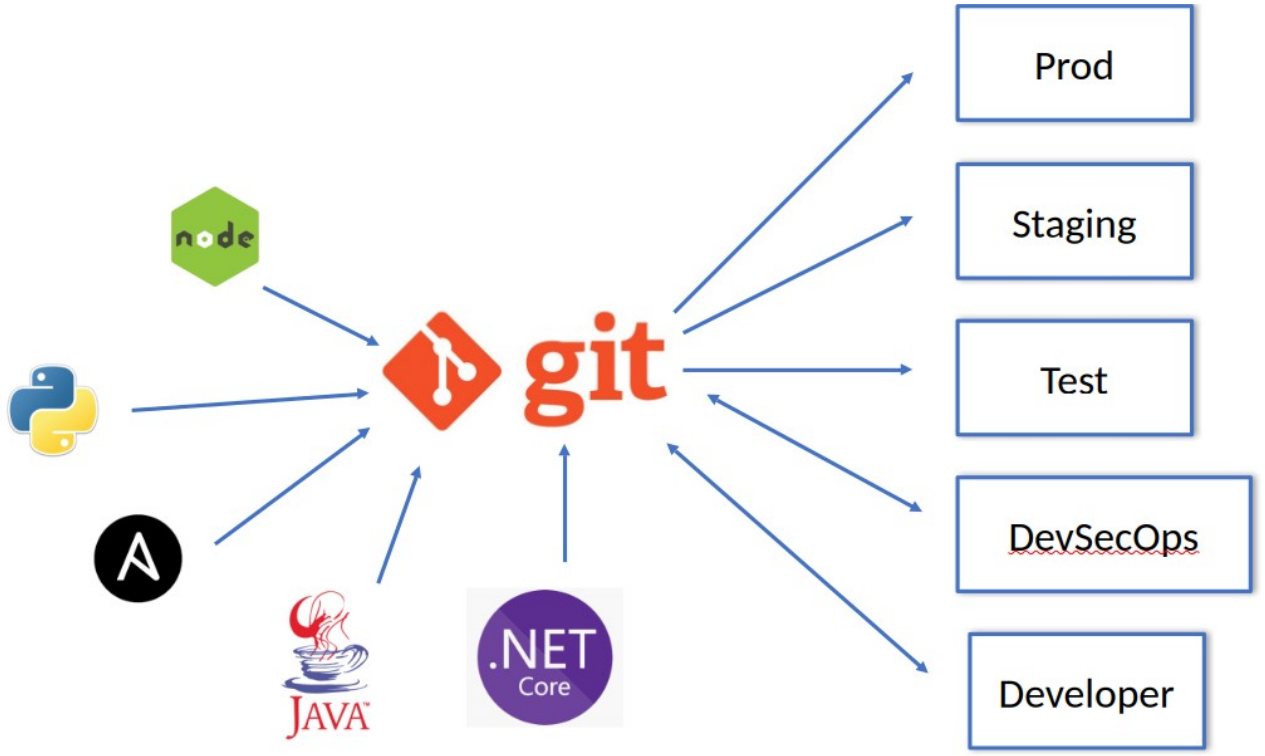
- Methodology for building micro-services apps (software as a service)
  - Drafted by developers at Heroku
  - First presented by Adam Wiggins circa 2011
  - Motivated by the problem of scaling the codebase and the operational environment
- Intended to apply to building applications that:
  - Use declarative formats for setup automation (sort of like our module definitions)
  - Have a clean contract with the underlying operating system (portable)
  - Are suitable for deployment on modern cloud platforms
  - Minimize divergence between development and production (DevOps)
  - Scale up without significant changes to tooling, architecture, or development practices

12 Factor Overview

Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management tasks as one-off processes

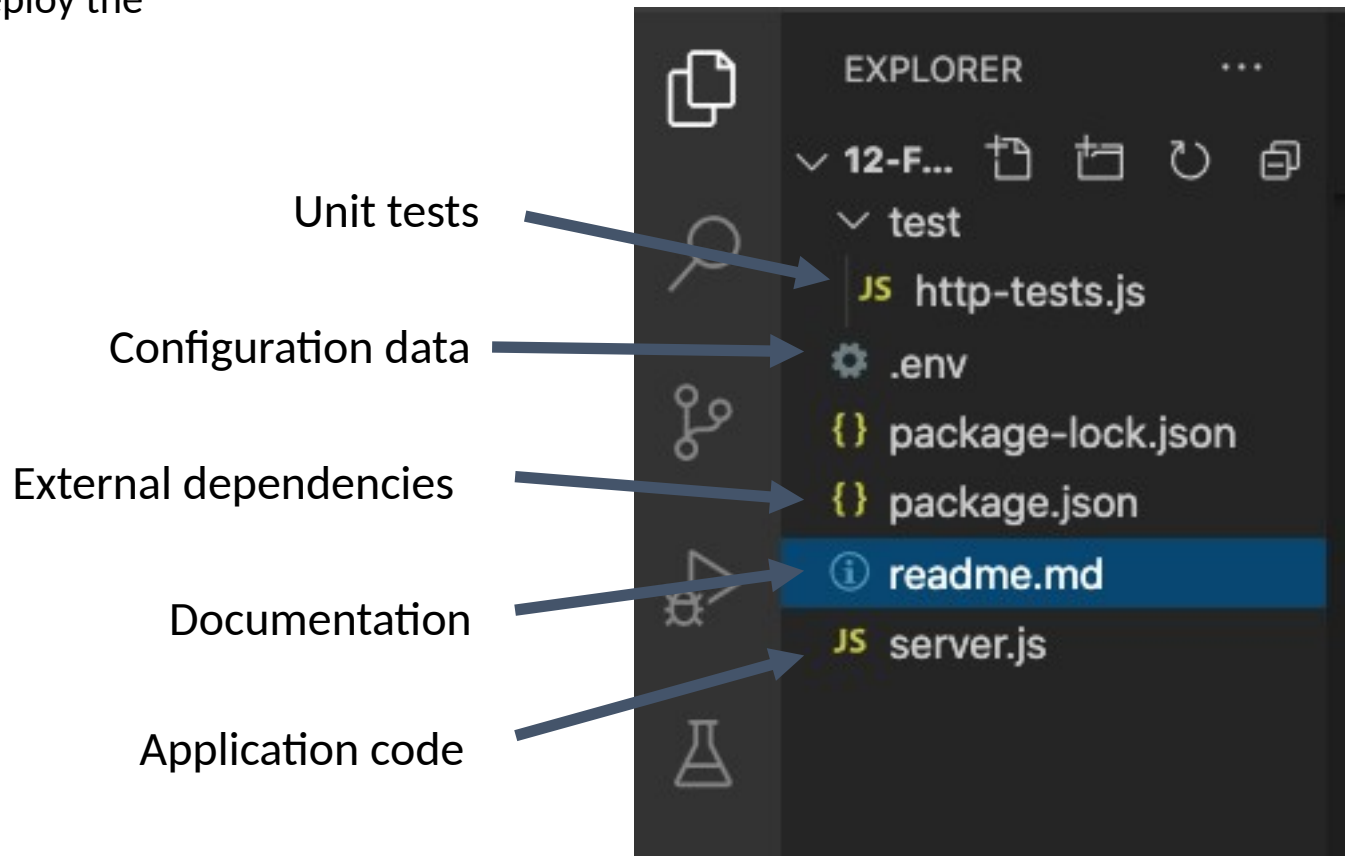
I. Codebase

One codebase needs to be defined for the enterprise. That code base is used to create, build, track, revise, control and conduct various deployments. Automation should be implemented in deployment so that everything can run in different environments without extra configuration work.



I. One Codebase, Many Deploys

The codebase contains all the artifacts necessary to build, test and deploy the application



II. Dependencies

Explicitly declare and isolate dependencies

Isolating dependencies means clearly declaring and isolating the dependencies. An app is a standalone structure, which needs to install dependencies. Whatever dependencies are required are declared in the code configuration.

```
"dependencies": {
  "apollo-link": "^1.2.11",
  "apollo-link-ws": "^1.0.17",
  "apollo-server": "^2.4.0",
  "graphql": "^14.2.1",
  "graphql-tag": "^2.10.1",
  "lodash": "^4.17.11",
  "node-fetch": "^2.3.0",
  "subscriptions-transport-ws": "^0.9.16",
  "uuid": "^3.3.2",
  "ws": "^6.2.0"
},
"devDependencies": {
  "chai": "^4.2.0",
  "faker": "^4.1.0",
  "graphql-request": "^1.8.2",
  "mocha": "^5.2.0",
  "supertest": "^3.4.2"
}
```

node: package.json

```
ordereddict==1.1
argparse==1.2.1
python-
dateutil==2.2
matplotlib==1.3.1
nose==1.3.0
numpy==1.8.0
pymongo==3.3.0
psutil>=2.0
```

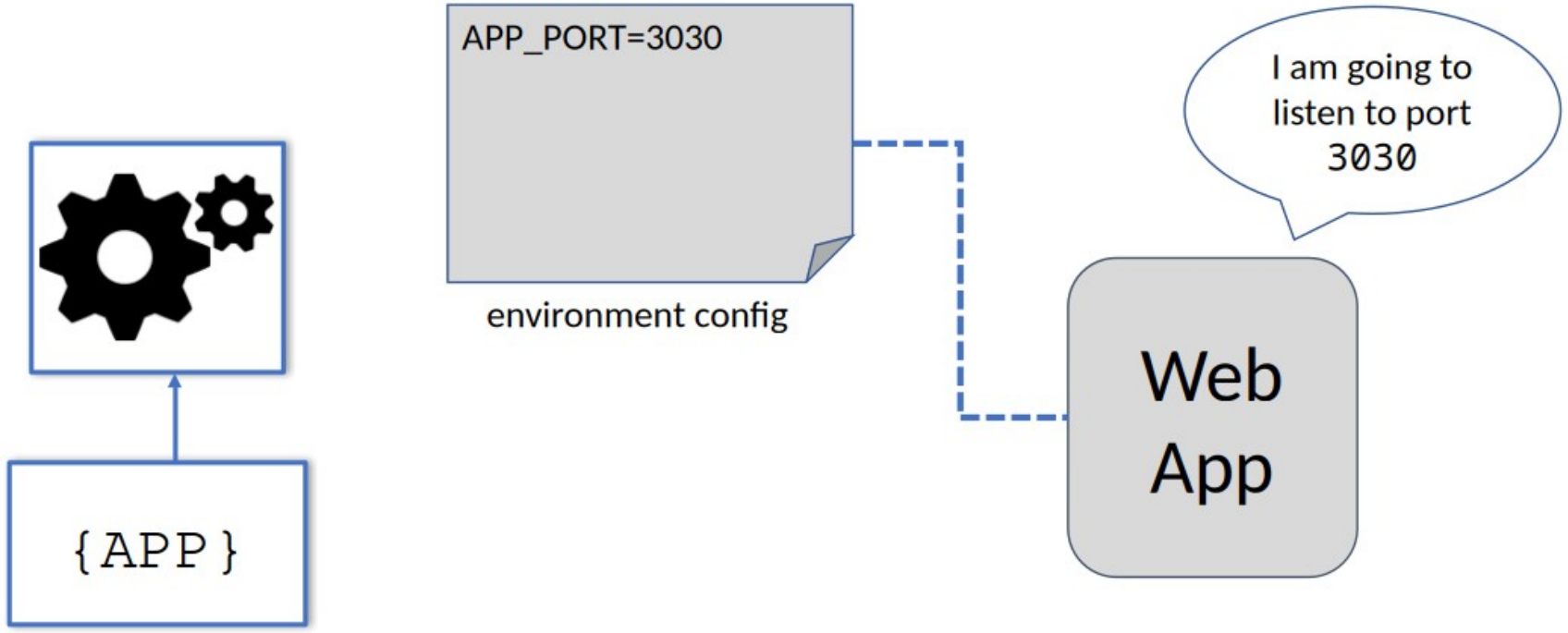
python: requirements.txt

III. Configuration

- The app configuration is anything likely to vary between deploys
  - Resource handles and references
  - Credentials
  - Environment dependent values like the hostname for the deploy
- There is no config information stored in the code
  - The code can be deployed into different environments without changes
- Litmus test:
  - If the codebase were made open source, no credentials would be compromised

III. Configuration

The concept of separation configuration from the app code can be applied to a variety of frameworks; for example, Docker Compose, Kubernetes or custom orchestration systems



III. Configuration

Examples of config files

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: echocolor-red
spec:
 replicas: 1
 selector:
  matchLabels:
   app: echocolor
   color: red
 template:
  metadata:
   labels:
    app: echocolor
    color: red
  spec:
   containers:
    - name: echocolor-red
      image: reselbob/echocolor:v0.1
      ports:
       -
        containerPort: 3000
      env:
       - name: COLOR_ECHO_COLOR
         value: RED
```

Kubernetes Manifest

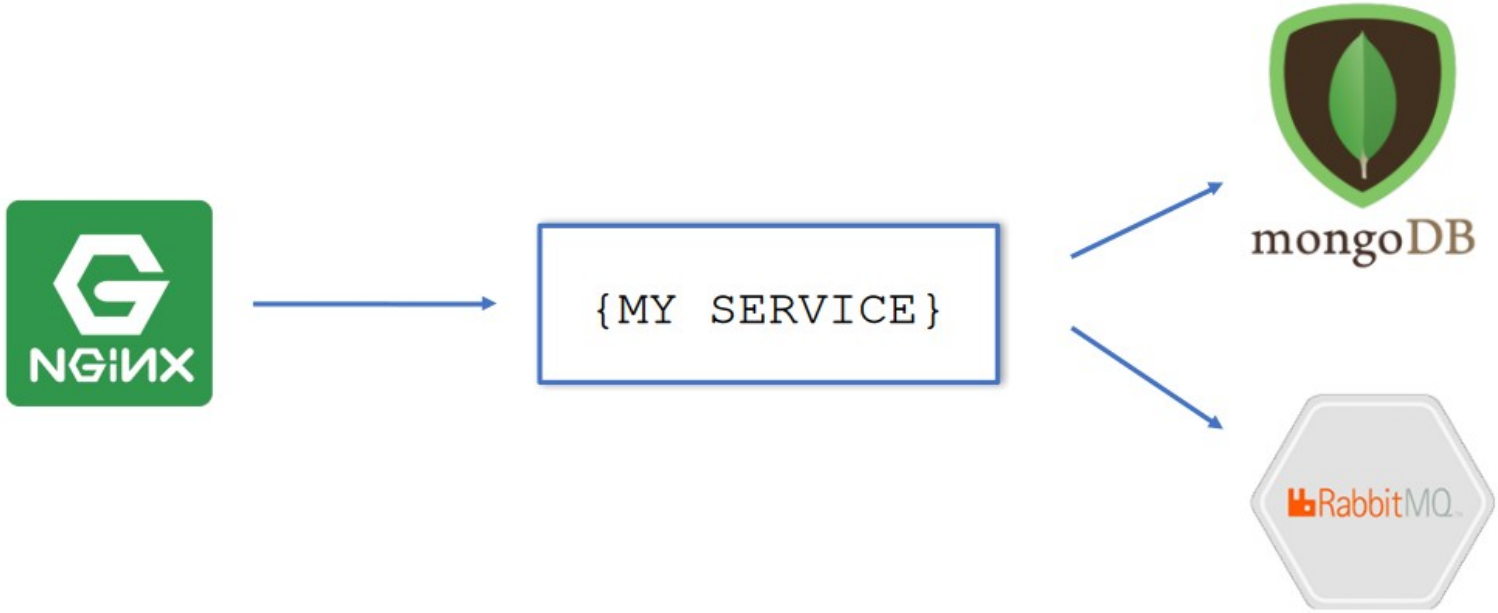
```
version: '3'
services:
 customer:
  build: ./customer
  ports:
   - "4000:3000"
  networks:
   - westfield_mall
  depends_on:
   - jaeger
 burgerqueen:
  build: ./burgerqueen
  networks:
   - westfield_mall
  depends_on:
   - jaeger
 hobos:
  build: ./hobos
  networks:
   - westfield_mall
 iowafried:
  build: ./iowafried
  networks:
   - westfield_mall
 payments:
  build: ./payments
  networks:
    - westfield_mall
 jaeger:
  image: jaegertracing/all-in-one:latest
  ports:
   - "6831:6831/udp"
   - "6832:6832/udp"
   - "16686:16686"
  networks:
   - westfield_mall
networks:
 westfield_mall:
```

Docker Compose

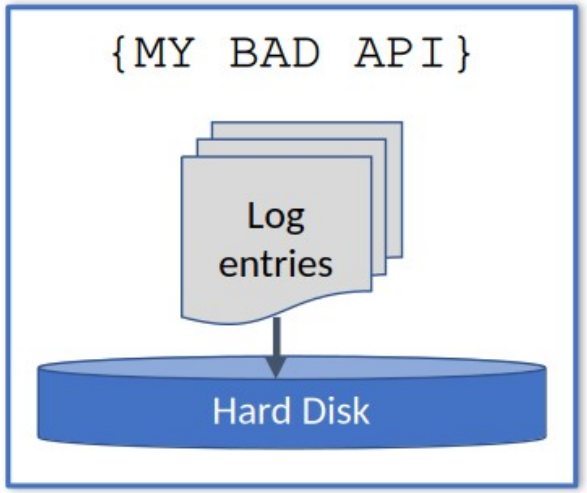
IV. Backing Services

A backing service is any service the app consumes during operation. All the backing services should be treated as attached resources to provide the flexibility to attach and detach on demand. For example, the app may require different deploy-dependent storage resources.

For example, a dev will want a lot of log files, which are stored in the dev's particular backing service for storage, while a QA engineer will not.

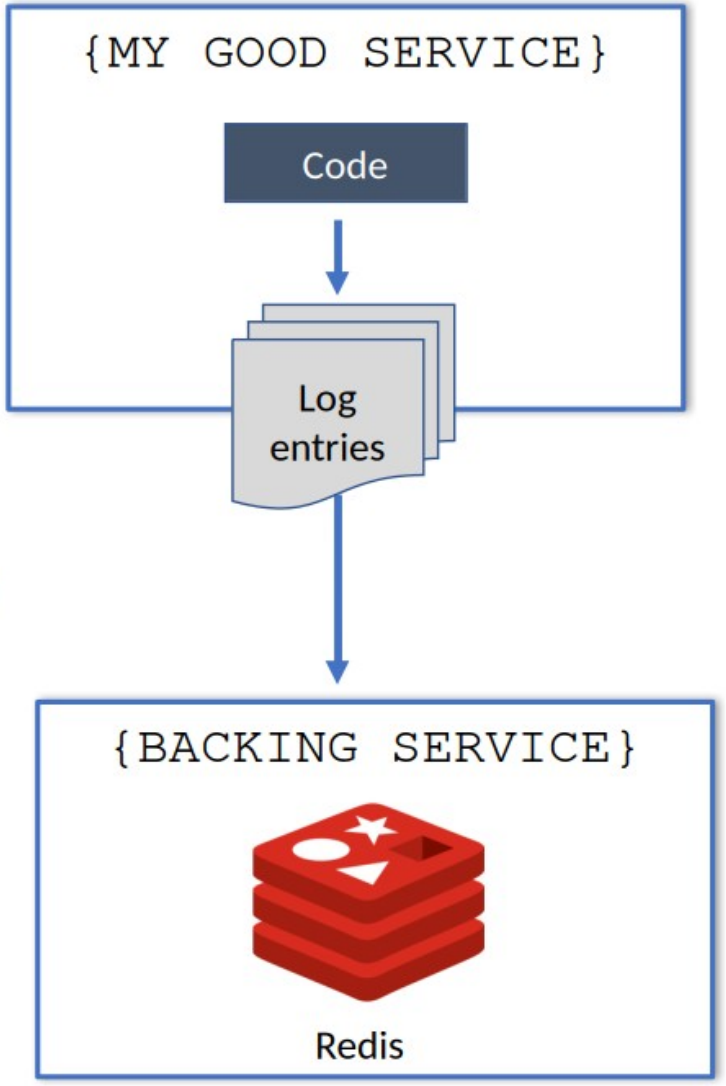


IV. Backing Services



When the hard disk fills or if the server disk is corrupted all the log files are lost.

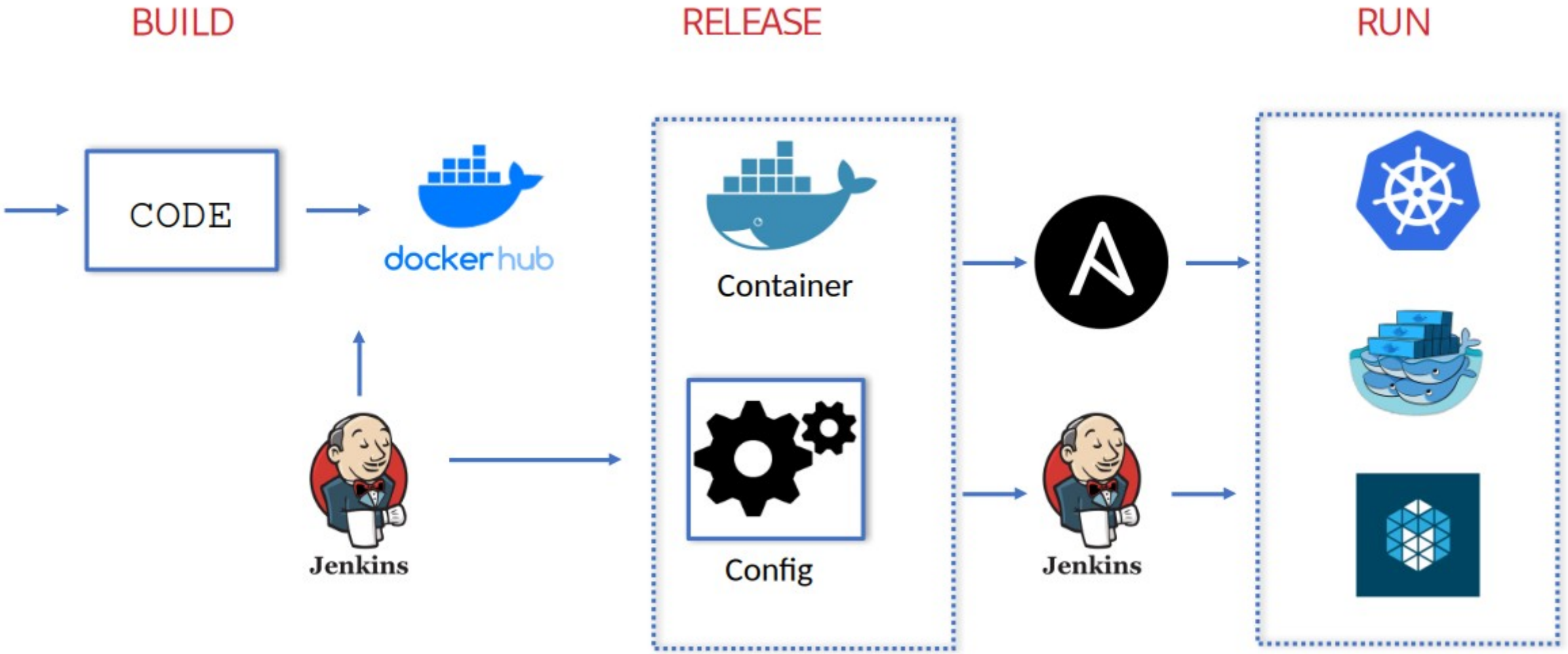
The app concerns itself only with its operational purpose. Extraneous tasks are delegated to the backing service which is specifically designed to implement the task.



V. Build, Release and Run

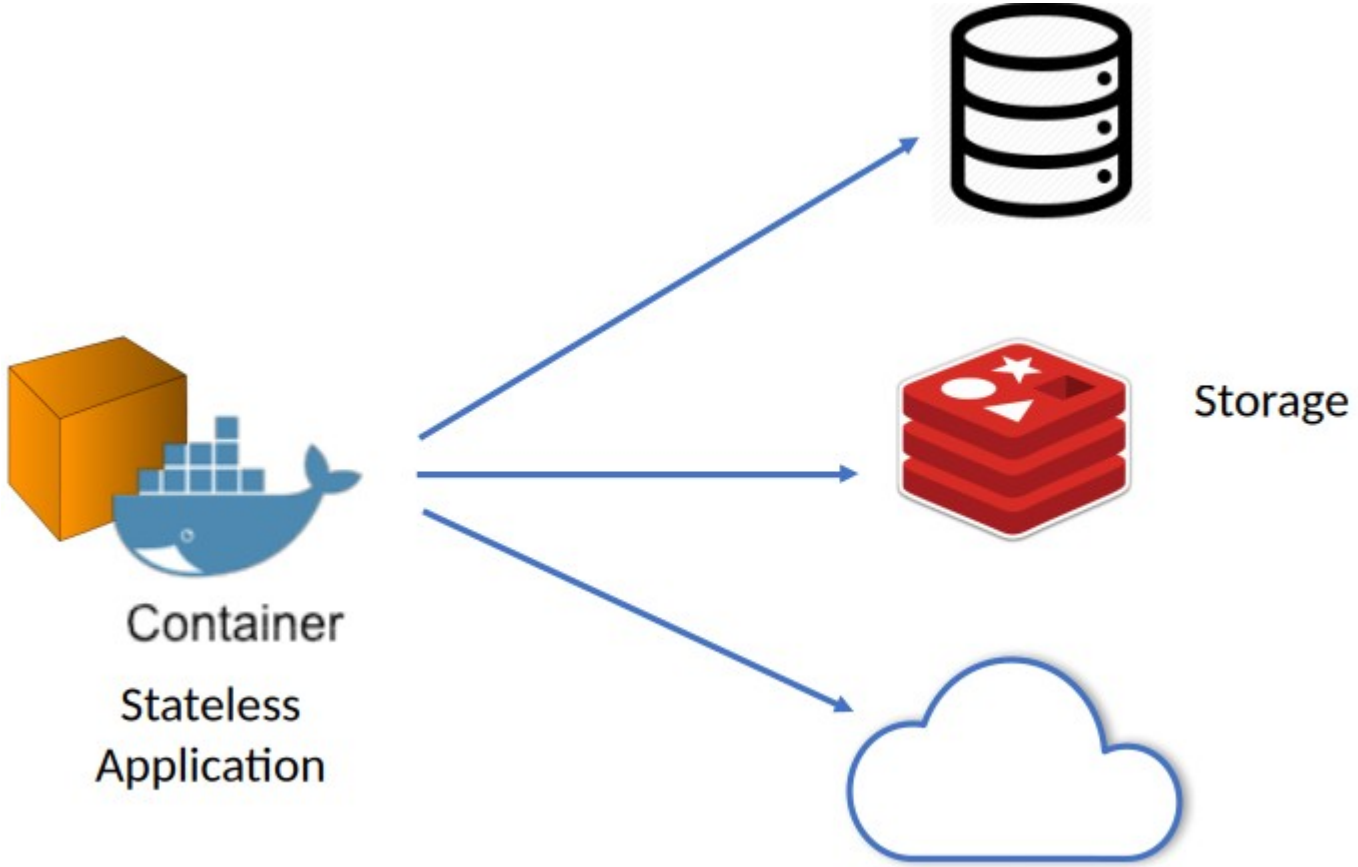
- A codebase is transformed into a deploy through three stages:
  - Build: converts a code repo into an executable bundle known as a build
  - Release: combines a build with the deploy's current config so that the release is ready for immediate execution
  - Run: runs the app in the execution environment
- These stages are strictly separated and distinct

V. Build, Release and Run



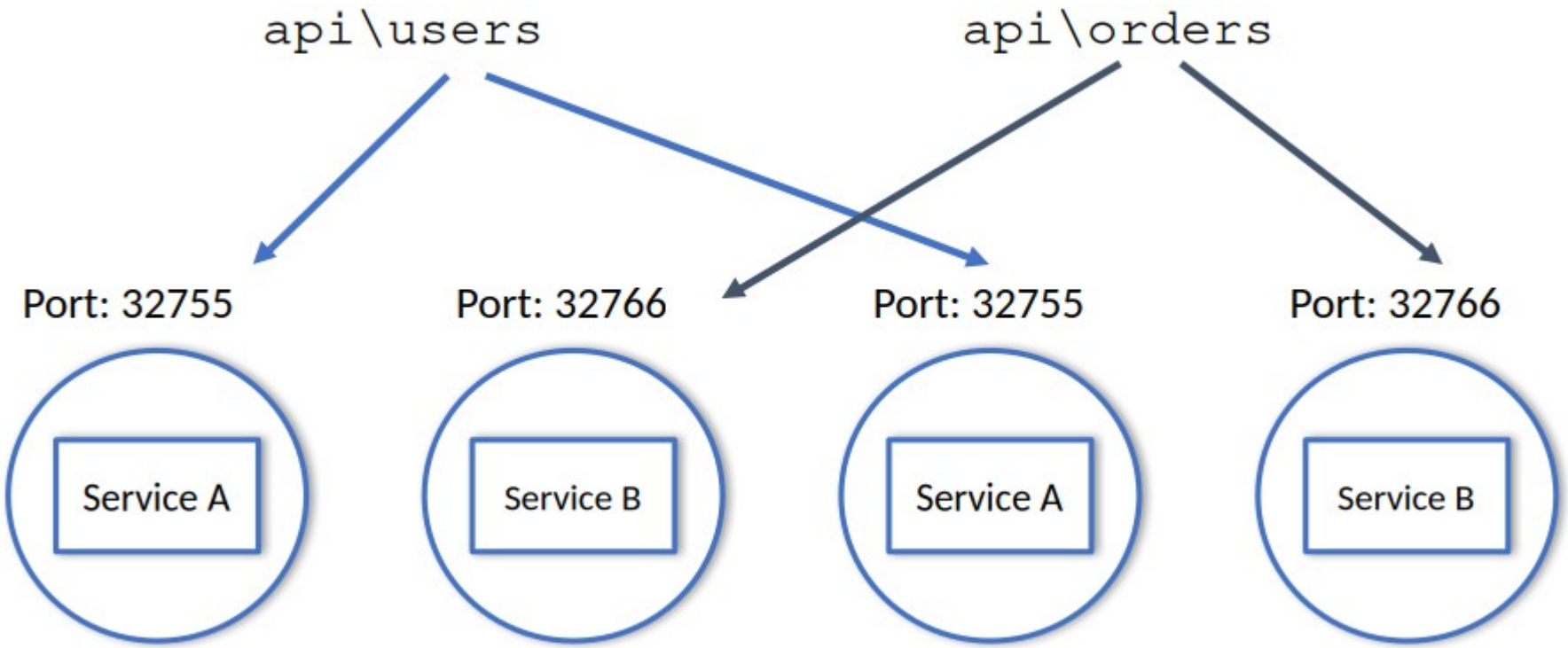
VI. Processes

Each app is executed as one or more stateless processes. Processes share nothing. All persistent or shared state is stored in a stateful backing service

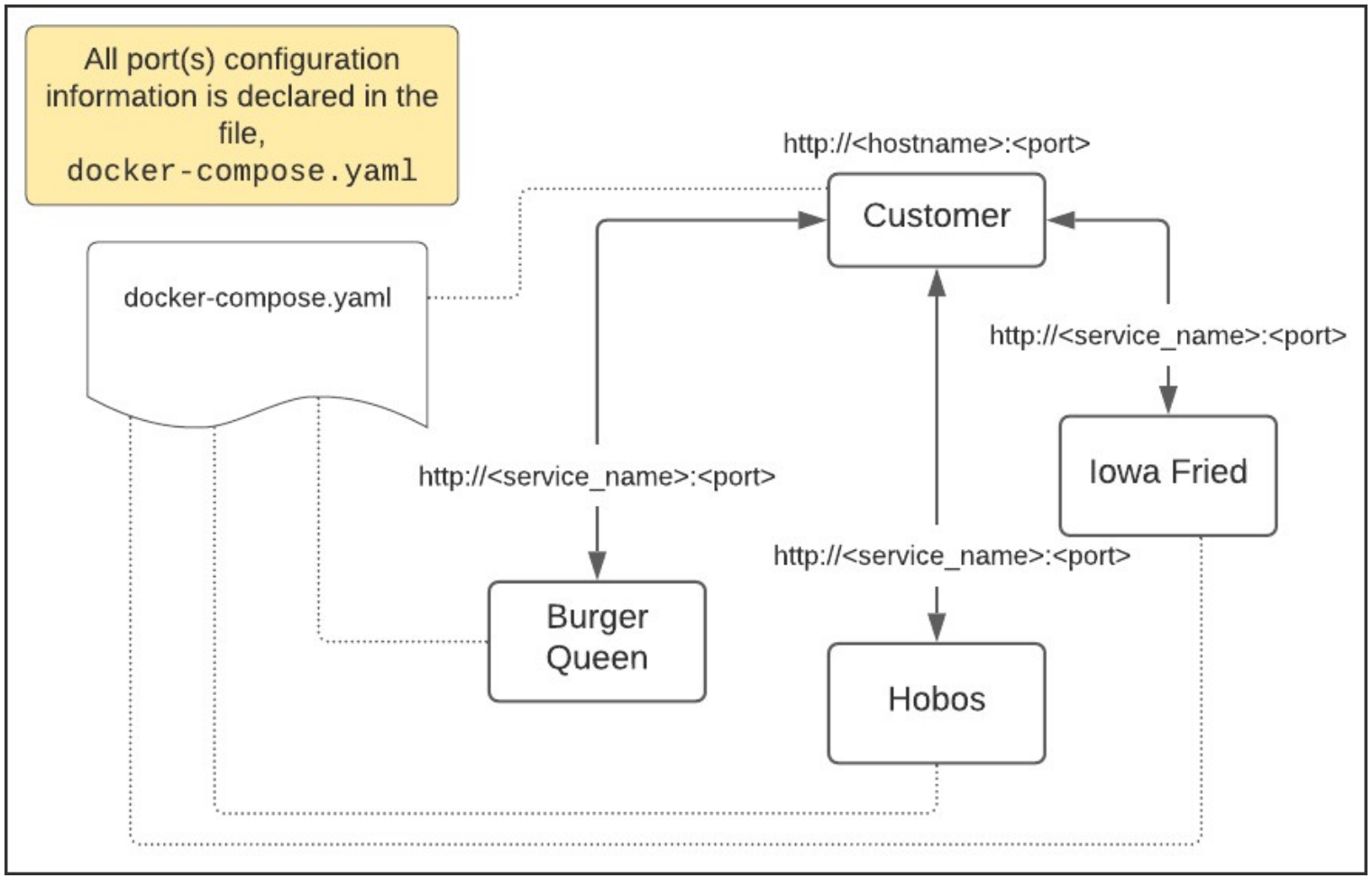


VII. Port Binding

All services are exposed via port binding. All services support URL based protocols internally with no webservers, message brokers or other network-aware service injected into the code.



VII. Port Binding



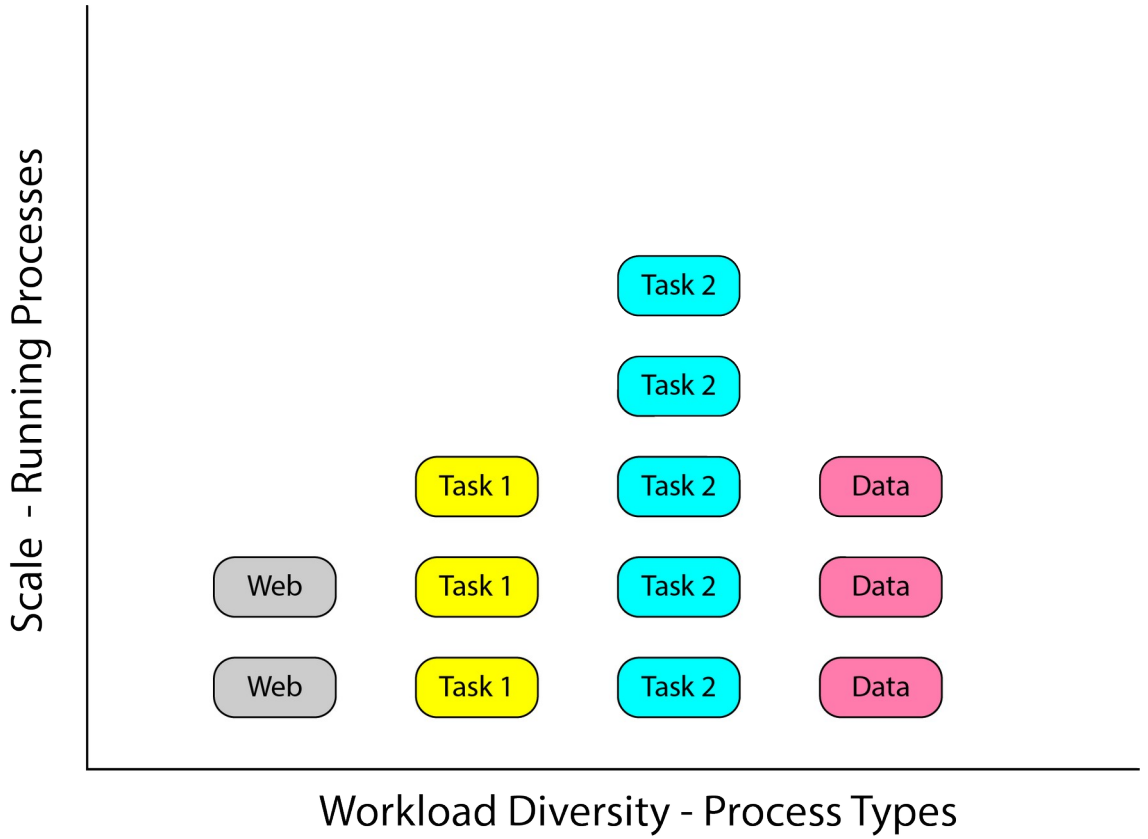
VIII. Concurrency

Scale out via the Process Model

Each task is performed by a process type

Low coupling and high cohesion supports horizontal scaling

Direct benefit of the process model



IX. Disposability

- Disposable processes mean:
  - They can be started and stopped at a moment's notice
  - They minimize startup time
  - They shutdown gracefully
- There is a way to return a job to the work queue
  - In case of underlying failure and time out of a process
- All jobs should be re-entrant
  - Typically, by wrapping the job in a transaction
  - Or making the operation idempotent

X. Dev/Prod Parity

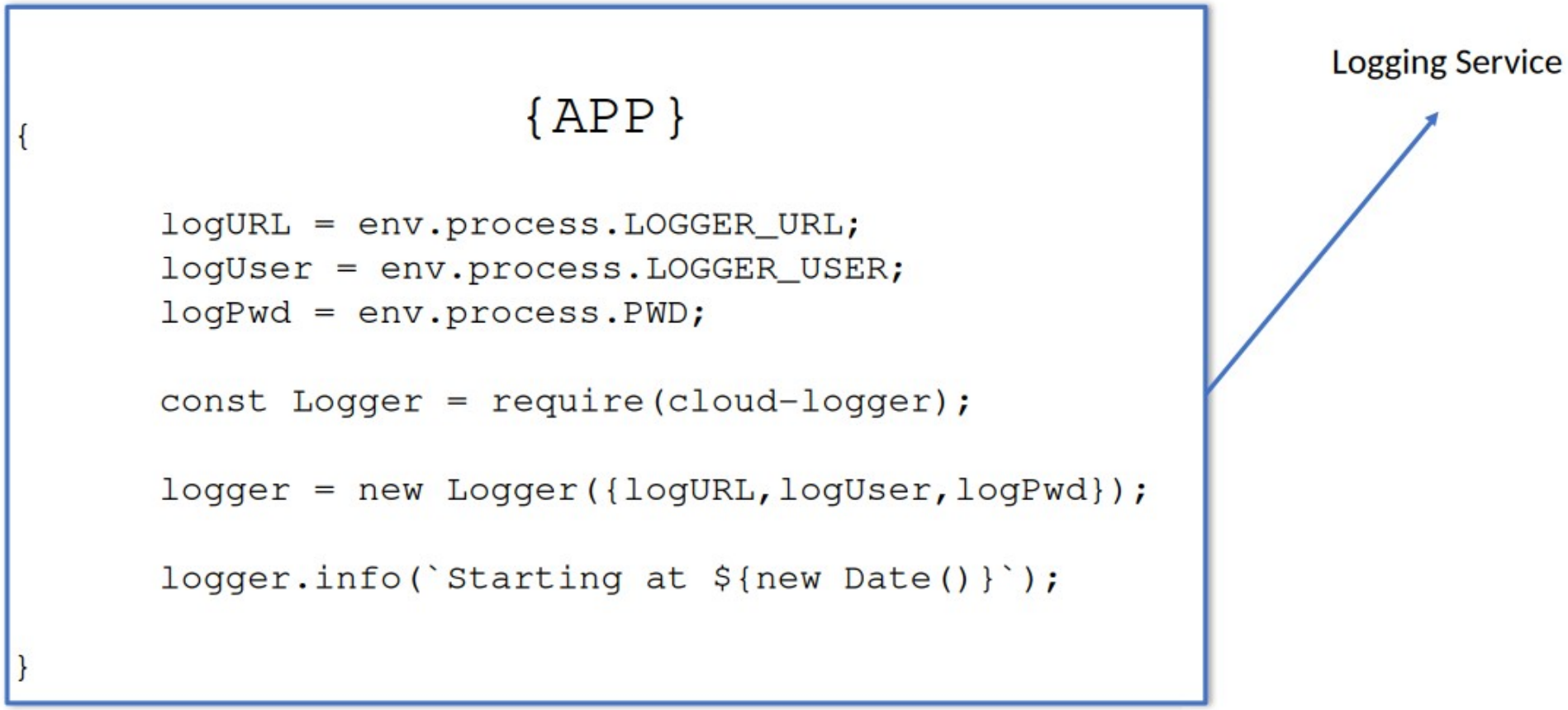
- Traditionally, a gap exists between dev and prod in three areas
  - Time: It may take days to months before code leaves dev and goes into prod
  - Personnel: Developers write code, ops engineers deploy it
  - Tools: Dev and ops may be using different tech stacks
- DevOps and CI/CD are used to reduce the gaps
  - Code goes into production in minutes or hours
  - Integration of prod and dev teams
  - Keep similar tooling in both prod and dev
- Avoid the use of different types of backing services in prod and dev
  - Dev environments should approximate closely the prod environment

XI. Logs as Event Streams

- Logs are streams of aggregated, time-ordered events
  - Collected from the output streams of all running processes and backing services
  - Have no fixed beginning or end, but flow continuously while the app is operating
- The execution environment is responsible for
  - Capturing each event stream log
  - Collating with other event logs
  - Routing to storage for analysis and storage
- The process only need write events to stdout

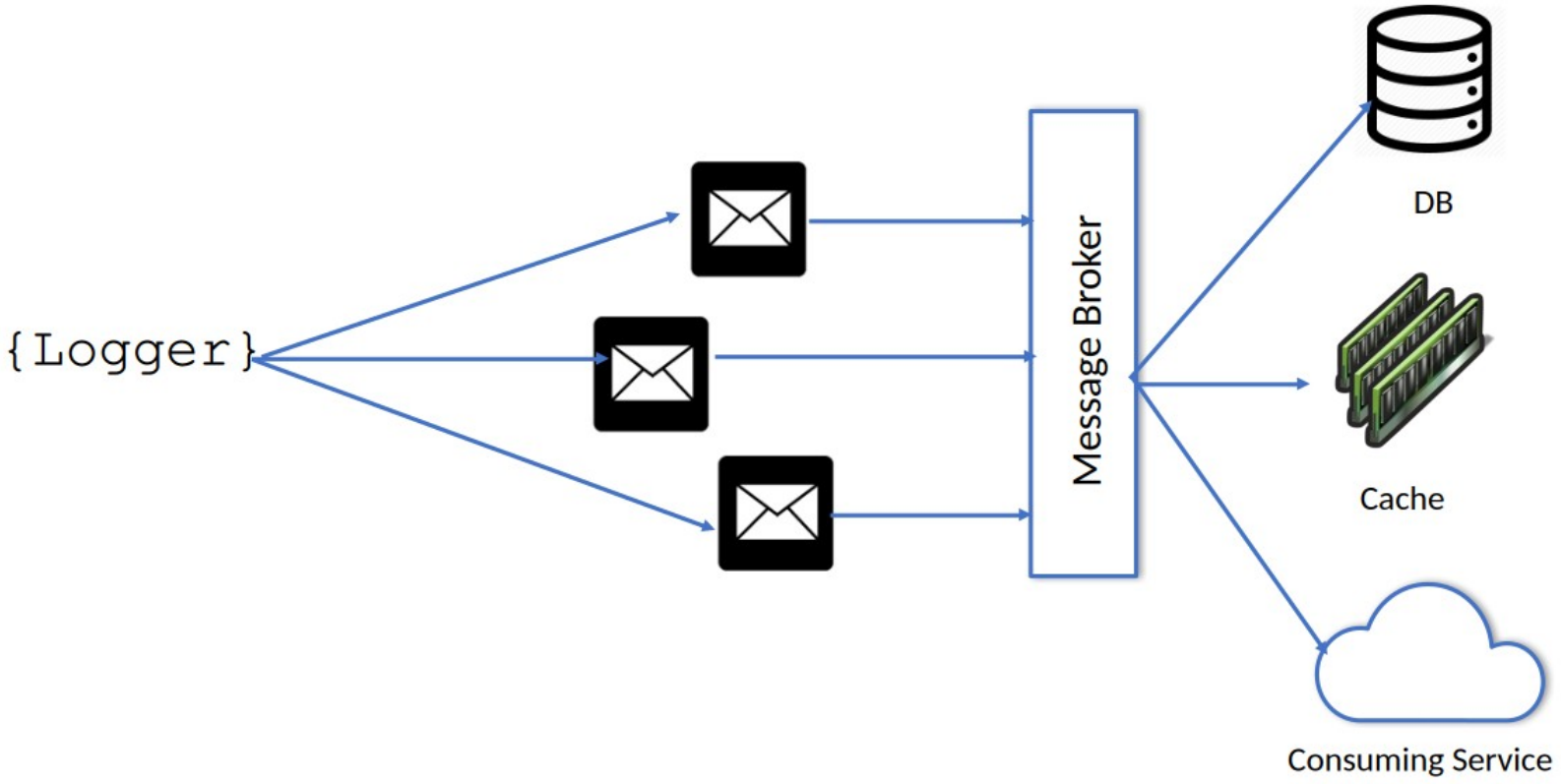
XI. Logs as Event Streams

Logging from inside a process



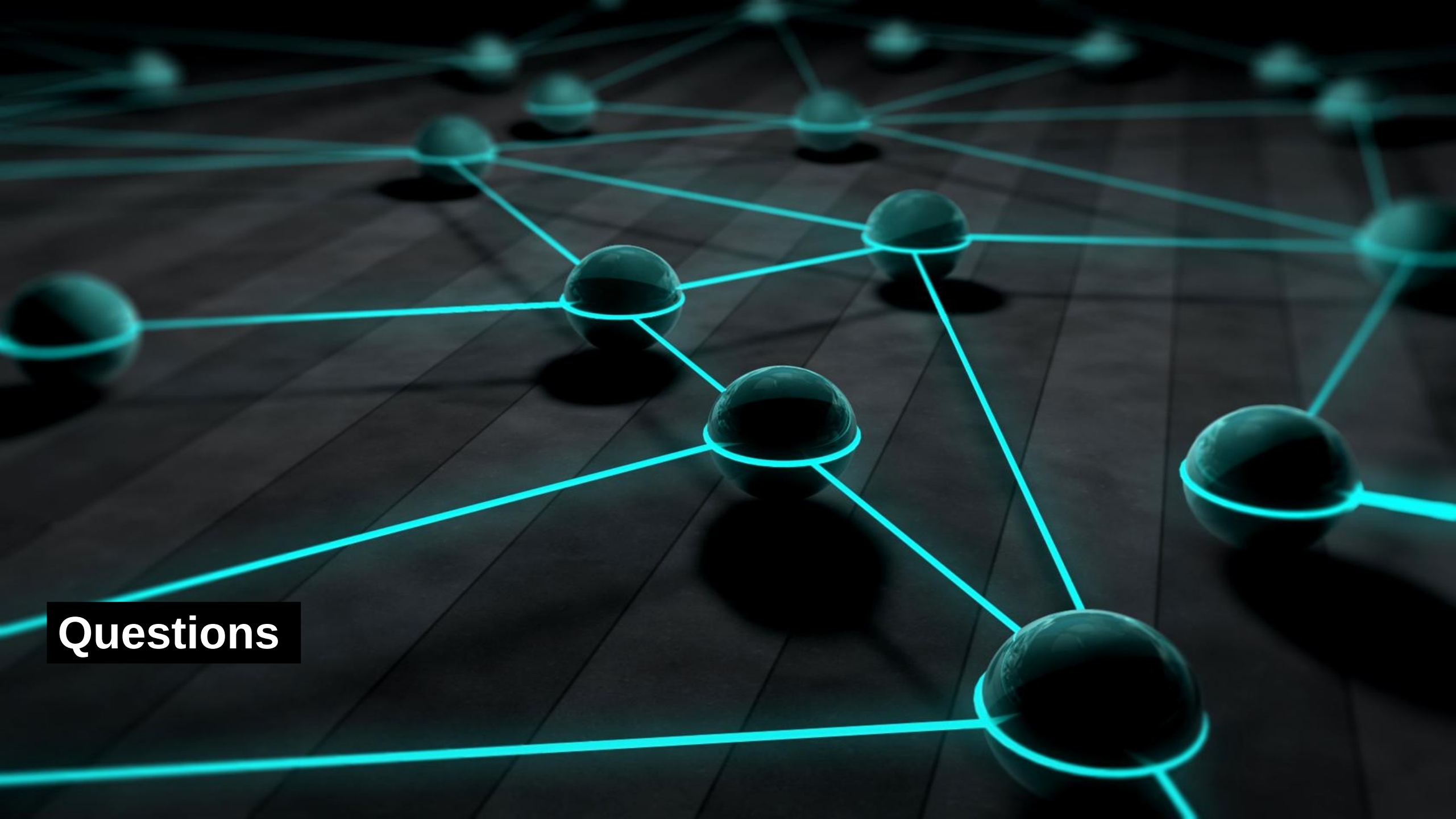
XI. Logs as Event Streams

Logging from inside a process



XII. Admin Processes

- In addition to regular processes, there will be one-off admin processes such as
  - Running database migrations
  - Running a console REPL
  - Running a one-time script
- Admin processes should run in the same environment as the app
  - Should use the same codebase and config
  - Admin code ships with release code to avoid synchronization issues
- A REPL shell should be available in the dev language
  - Allows admin scripts to be run easily


Questions