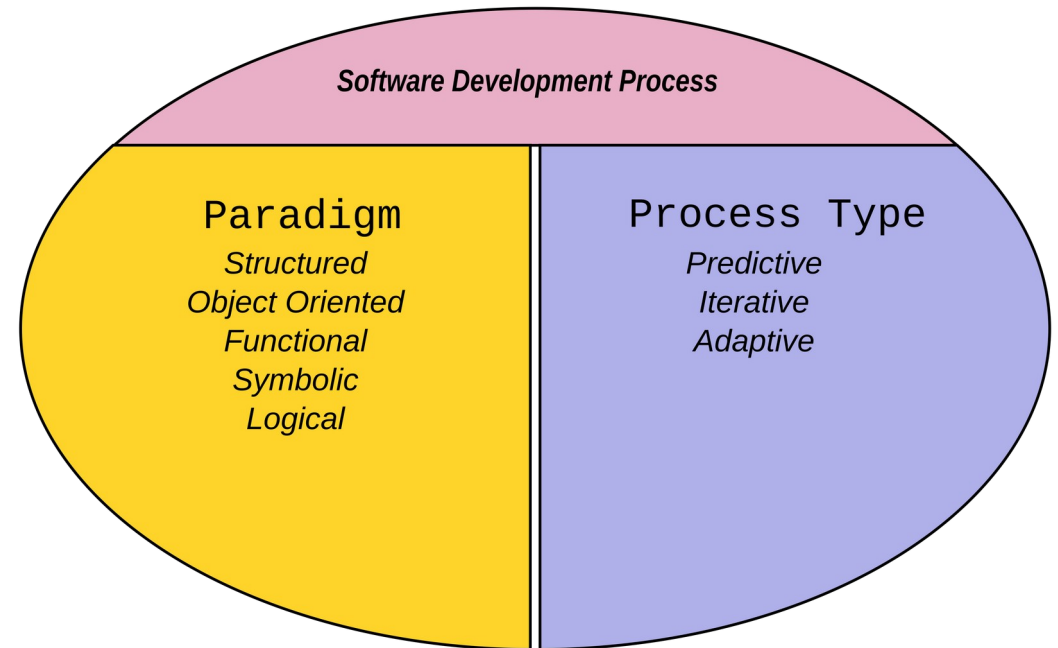**Program Architecture**

Recall

- In the software engineering module

  – Software Engineering was developed in response to a "driver"

  – A driver is a need or change in the industry, market or society that forces to act differently

- The driver for software engineering was the need to manage the increasing complexity of systems in the 1970s

- In this module, we examine the different programming paradigms that have come into use because of various drivers
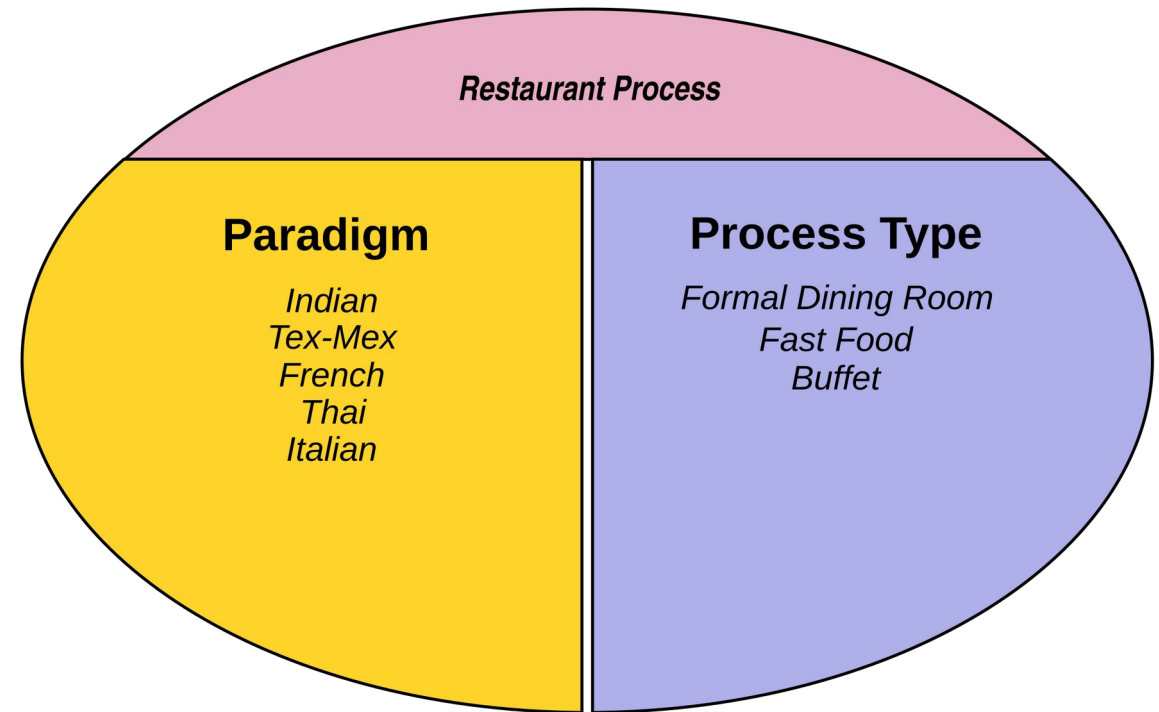
Recall Process and Paradigm

- Software development processes are made up of two parts:

  - A paradigm, which describes how to develop programs using a specific programming language or model

  - A development life cycle, which describes how to structure the paradigm activities in order to deliver the triangle under the constraints of the iron triangle

- Paradigms are defined generally as

  - The entire constellation of beliefs, values, techniques and so on shared by the members of a given community.

  - It also denotes one sort of element in that constellation, the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science

Software Development Process

Paradigm
Structured
Object Oriented
Functional
Symbolic
Logical

Process Type
Predictive
Iterative
Adaptive

Recall Process and Paradigm

- We can see this distinction in many different areas where we "make things"

- When we look at restaurants we see:

  – Culinary Paradigms: TexMex, Cajun, French, etc

  – Food Production Life Cycle: Fast Food, Fine Dining, Buffet, Take Out, etc.

- The paradigm describes how to cook the food

- The process is necessary for restaurants to meet the iron triangle

- Restaurants can be seen as applying engineering principles to food delivery

Restaurant Process

Paradigm

Indian
Tex-Mex
French
Thai
Italian

Process Type

Formal Dining Room
Fast Food
Buffet

Cyber Human Systems 1990s - 2010s

- Drivers

  - Rise of personal computing

  - Rise of the Internet and networking

  - Businesses going "on-line"

  - Cheaper hardware

- Programming

  - Increased use of OO languages like Java

  - Separated front-end users from the back-end business

  - Backends were usually mainframe based

  - New front end technologies exploded in the marketplace

  - Emphasis was on network based infrastructure

Drivers of the Current Era

- Exponential increases in the amount of data being generated and needing to be processed
  - Generally referred to as "Big Data"
  - Direct result of devices being connected in the Internet of Things
- Support for new paradigms of computing
  - Machine learning and analytics
  - Processing large amounts of data at scale and often in real time
- Rapidly increasing hardware capabilities
  - Virtualization, cloud computing, containerization and microservices
- Reduced time to market for new or upgraded applications
- Deployment of applications at massive scales
  - Reformulation of the basic ideas of software architecture
- No down-time or disruption of services
  - High reliability software
  - Critical systems driven by software
- The increasing use of functional and metaprogramming

ProTech
protechtraining.com

Cyber Physical Systems
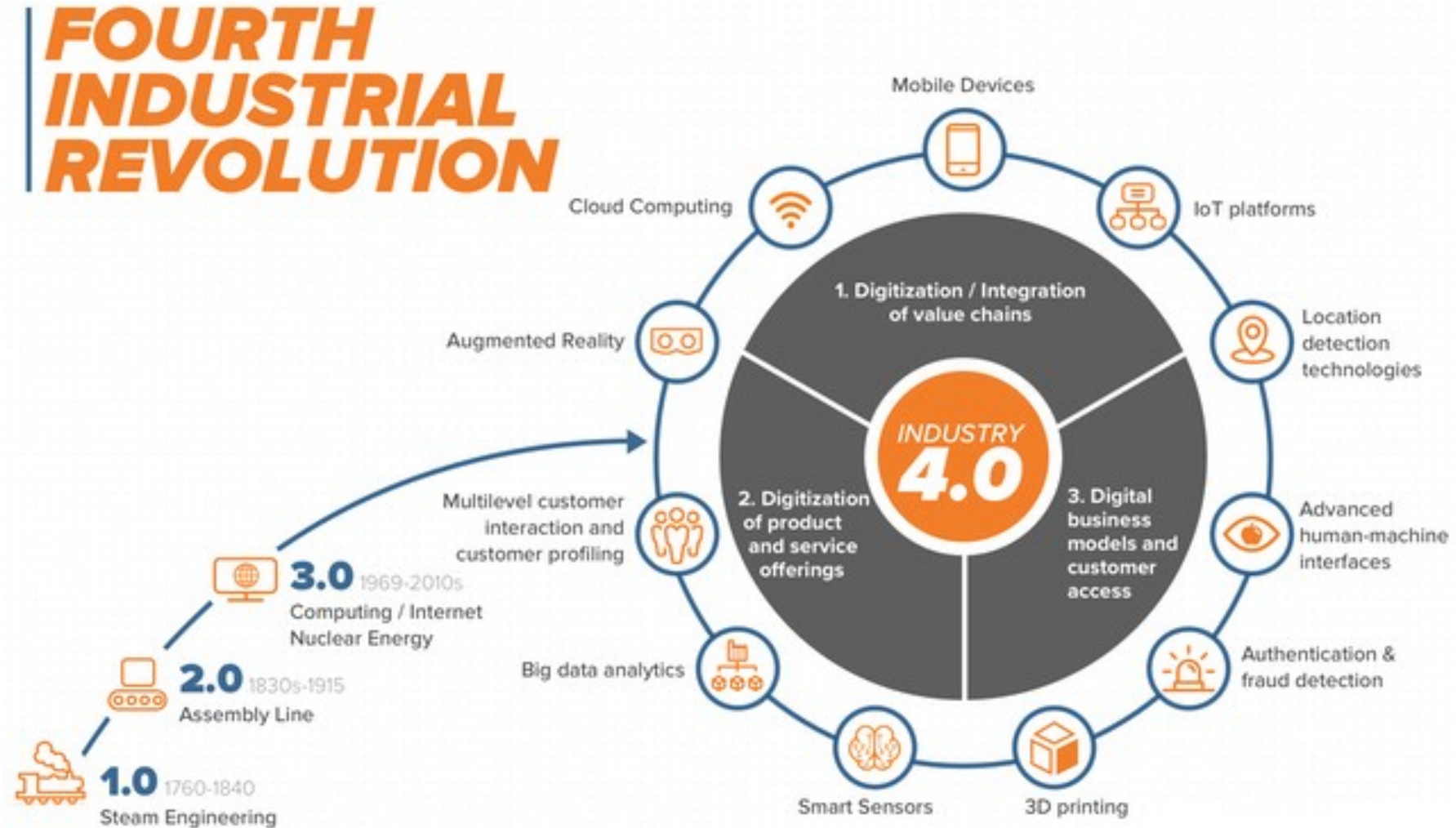
Rethinking Application Development

- Requires new ways of thinking about app development
  - Release time needs to be days, not months
  - Robustness is essential – no downtime

- Massive scales
  - Hundreds of millions of transactions per second
  - Zeta-bytes of data
  - Data is generated in various forms from user interactions and connected devices

- Applications microservices
  - Clusters of smaller scalable components working together
  - Requires writing code for containerized deployment as opposed to stand alone applications
  - Often results in a "death star" architecture
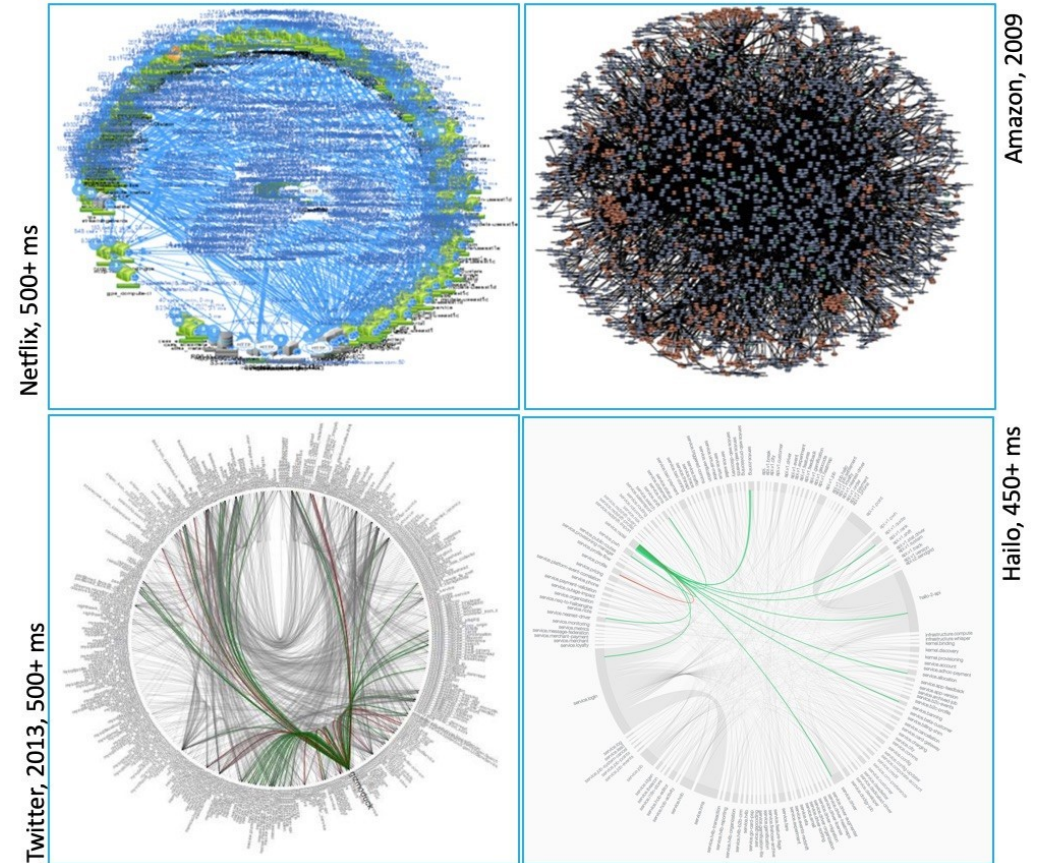
Programming Paradigms

- A programming paradigm is a set of:
  - Assumptions about what the components of a program should be
  - Techniques and principles for building programs
  - Assumptions about what sort of problems are solved best by that paradigm
  - Best practices for software design and code style

- There are multiple programming paradigms
  - Most have been around since the start of high-level programming in the 1960s
  - A paradigm becomes "main-stream" and supported in programming languages when a set of problems arise that the paradigm is better suited to solve than the other paradigms in use

- The main-stream paradigms in use today – and supported in Java are:
  - Structured or procedural programming
  - ObjectOriented Programming
  - Functional Programming

Programming Language "Styles"

- Imperative programming
  - Code is a series of instructions that specify the computational steps to be executed
  - Usually said to be expressible as a Turing machine (for you math nerds)
  - Intended to be compiled directly into assembly code
  - Imperative code is usually bundled into reusable "procedures"
  - DRY principle – "Do not repeat yourself"

- Declarative programming
  - Code is a description of what a final result should be
  - Usually done by calling an existing procedure

- This is a continuum of style, not discrete, mutually exclusive categories
  - Most code is a mixture of imperative and declarative styles
  - Eg. x = sin(y) * 3.14159
    - The call to sin() is declarative since we don't know (or care) how the sin is computed
    - The multiplication is imperative code

Structured Programming

- Code is structured into reusable modules
  - Called subroutines or functions or procedures
  - Standard libraries of procedures are part of the programming language

- Users can define their own procedures and libraries
  - Procedures are the highest level of organization
  - Implementation of DRY
  - Image is a FORTRAN subroutine

```fortran
C *******************************************************************
C
C
      SUBROUTINE TRISOL(A,B,C,D,H,N)
C
C ************** TRI-DIAGONAL MATRIX SOLVER ************************
C
C *** THIS TRIDIAGONAL MATRIX SOLVER USES THE THOMAS ALGORITHM ********
C
      dimension A(250),B(250),C(250),D(250),H(250),W(250),R(250),G(250)
      W(1)=A(1)
      G(1)=D(1)/W(1)
      do 100 I=2,N
      I1=I-1
      R(I1)=B(I1)/W(I1)
      W(I)=A(I)-C(I)*R(I1)
      G(I)=(D(I)-C(I)*G(I1))/W(I)
100   continue
      H(N)=G(N)
      N1=N-1
      do 200 I=1,N1
      II=N-I
      H(II)=G(II)-R(II)*H(II+1)
200   continue
      return
      end
```

OO Programming

- Procedures are methods within class definitions
  - There are no stand-alone procedures
  - Methods tend to be where the imperative style of coding is mostly found
- Some classes allow for static methods
  - The class is used to define an API that works just like a library in a structured programming language
- OO programming is about where we write and store our reusable code
  - Specifically, in class definitions
  - Image is Simula67 code from the mid 1960s
  - From the official Simula reference material

```
Begin
    Class Glyph;
        Virtual: Procedure print Is Procedure print;;
    Begin
    End;

    Glyph Class Char (c);
        Character c;
    Begin
        Procedure print;
            OutChar(c);
    End;

    Glyph Class Line (elements);
        Ref (Glyph) Array elements;
    Begin
        Procedure print;
        Begin
            Integer i;
            For i:= 1 Step 1 Until UpperBound (elements, 1) Do
                elements (i).print;
            OutImage;
        End;
    End;

    Ref (Glyph) rg;
    Ref (Glyph) Array rgs (1 : 4);

    ! Main program;
    rgs (1):- New Char ('A');
    rgs (2):- New Char ('b');
    rgs (3):- New Char ('b');
    rgs (4):- New Char ('a');
    rg:- New Line (rgs);
    rg.print;
End;
```

Functional Programming

- Procedures are defined like mathematical functions

  - Functions act like data

  - Said to be first class citizens

  - Programs are treated like function composition in math

  - Variables are immutable

- LISP introduced functional programming in the 1950s

  - Listing shown is 1960s APL code for computing a matrix determinant

```
     ∇DET[□]∇
   ∇ Z←DET A;B;P;I
[1]    I←□IO
[2]    Z←1
[3]  L:P←(|A[;I])ι⌈/|A[;I]
[4]    →(P=I)/LL
[5]    A[I,P;]←A[P,I;]
[6]    Z←-Z
[7] LL:Z←Z×B←A[I;I]
[8]    →(0 1 ∨.=Z,1↑ρA)/0
[9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]   →L
[11] ⍝EVALUATES A DETERMINANT
   ∇
```

Other Paradigms

- Two other paradigms that we will not deal with since they are not mainstream. Yet.

- Symbolic Programming or meta-programming
  - Treats code and data as the same thing – sets of symbols
  - Allows code to rewrite itself or to write new code
  - Introduced in LISP in the 1960s

- Logical Programming
  - Treats code as propositional logic
  - Implemented by the Prolog language in 1972
  - Image shows a basic Prolog program

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

Modern Programming Languages

- Most modern programming languages support more than one paradigm

    - Modern languages like Rust, Go and Julia are designed to support multiple paradigms

- Legacy Languages are often revised to add support for a paradigm

    - COBOL added object-oriented support

    - Java added support for functional programming in Java 8

- Why paradigms go mainstream

    - Most of the different paradigms have existed for over 50 years

    - They are designed to solve a particular class of problems

    - The types of problems industry deals with change over time

        - Changes often result from changes in technology and user requirements
        - Existing paradigms may not be able to solve these new problems
        - A different paradigm is main-streamed that can solve the problems

How Programs are Organized

- The basic unit of code in all languages is the statement

  - A single self contained expression that can be executed, for example, a print statement

  - An expression is code that can be evaluated to a value like "3 + 5"

  - In some languages, all statements are expressions, i.e., every statement returns a value

  - In some languages, every expression can also be a statement.
    - This is very language dependent

- Where the paradigms differ is in the smallest unit of code that can be executed

  - Normally we can't just execute a single statement, unless we are in some interactive environment

  - The smallest unit tends to be a bundle of statements
    - A function, a class method or a subroutine.

Structured Code

- For structured code, like C
  - The smallest unit of code is the subroutine.
  - In C type languages, these are called functions, but they are not the same as functions in functional languages.
  - Functions can be organized into files which are the main program unit (in C)
  - But these are compiled and linked into a single executable with a top down flow of control
  - There is a specific routine called the mainline that specifies where execution starts
    - In C-type languages, this entry point is the 'main()' function.
  - Libraries consist of collections of subroutines that can be linked into the executable
  - Data is organized into data structures like arrays and structs

C Example − Bank Account

- Starts with a header file that

    - Defines or describes the functions that are used in a file

    - Describes some data structures

- Header files are used

    - To describe the functions that are in a file

    - To tell other file where functions in other files are

    - Important for functions calling functions in other files

```c
/* account.h */
#ifndef ACCOUNT_H
#define ACCOUNT_H
typedef struct {
    double balance;
} Account;


void init_account(Account* acc, double initial);
void deposit(Account* acc, double amount);
int  withdraw(Account* acc, double amount);
#endif
```

C Example – Bank Account

- All of the related code is defined in a source file

```
/* account.c */
#include "account.h"

void init_account(Account* acc, double initial) { acc->balance = initial; }

void deposit(Account* acc, double amount) { acc->balance += amount; }

int withdraw(Account* acc, double amount) {
    if (amount > acc->balance) return 1;
    acc->balance -= amount;
    return 0;
}
```

C Example − Bank Account

- The mainline calls the functionality of the account file to execute a task

```c
/* main.c */
#include <stdio.h>
#include "account.h"

int main(void) {
    Account acc;
    init_account(&acc, 100.0);
    deposit(&acc, 50.0);
    if (withdraw(&acc, 20.0) != 0) {
        printf("Insufficient funds\n");
    }
    printf("Balance: %.2f\n", acc.balance);
    return 0;
}
```

Object-Oriented Code

- For object-oriented code, like Java

  – The smallest unit of code is the method.

  – Methods are grouped together inside classes, which combine both data (fields) and behavior (methods) into a single unit.

  – Classes are organized into packages or namespaces which represent components

- Control flow is often distributed:

  – Instead of a strict top-down order, objects interact by sending messages (method calls).

- Libraries consist of collections of classes that can be reused across programs.

- Data is encapsulated inside objects

  – Access restricted and controlled through methods and visibility rules (e.g., public, private).

Java Example – Bank Account

- Starts with a class definition

    - Encapsulates data and functionality

    - Defines a type

    - Used to create multiple objects of type "bank account"

```java
// Account.java
public class Account {
    private double balance;

    public Account(double initial) {
        this.balance = initial;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        if (amount > balance) throw new IllegalArgumentException("Insufficient funds");
        balance -= amount;
    }
    public double getBalance() { return balance; }
}
```

ProTech
protechtraining.com

Java Example − Mainline

- Still has a main() method

  - Used to boot the application

  - And create the objects from the class definitions that are needed to make the app run

  - The objects execute by passing messages.

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        Account acc = new Account(100.0);
        acc.deposit(50.0);
        acc.withdraw(20.0);
        System.out.printf("Balance: %.2f%n", acc.getBalance());
    }
}
```

Haskell Example − Module

- This declares a module called Account.

  - Defines new type called Account

  - And the functions new, deposit, withdraw, and the accessor for the balance.

  - Other helper code (if any) that's not listed stays private to the module.

```
-- Account.hs
module Account (Account, new, deposit, withdraw, balance) where

newtype Account = Account { balance :: Double } deriving (Show)

new :: Double -> Account
new initial = Account initial

deposit :: Double -> Account -> Account
deposit amt (Account b) = Account (b + amt)

withdraw :: Double -> Account -> Either String Account
withdraw amt (Account b)
  | amt > b   = Left "Insufficient funds"
  | otherwise = Right (Account (b - amt))
```

Haskell Example − Mainline

The program imports the Account module

- Creates a new account acc0 with an initial balance of 100.0.

- It deposits 50.0, producing acc1 with a balance of 150.0.

- Functional code does not change the value of a variable, it creates a new variable from the old one with the change

- Tries to withdraw 20.0:

- If the withdrawal fails (insufficient funds), it prints the error message.

- If successful, it prints the resulting balance (in this case: 130.0).

```
-- Main.hs
module Main where
import Account

main :: IO ()
main = do
  let acc0 = new 100.0
      acc1 = deposit 50.0 acc0
  case withdraw 20.0 acc1 of
    Left err    -> putStrLn err
    Right acc2 -> putStrLn $ "Balance: " ++ show (balance acc2)
```

Hierarchy of Constructs Example
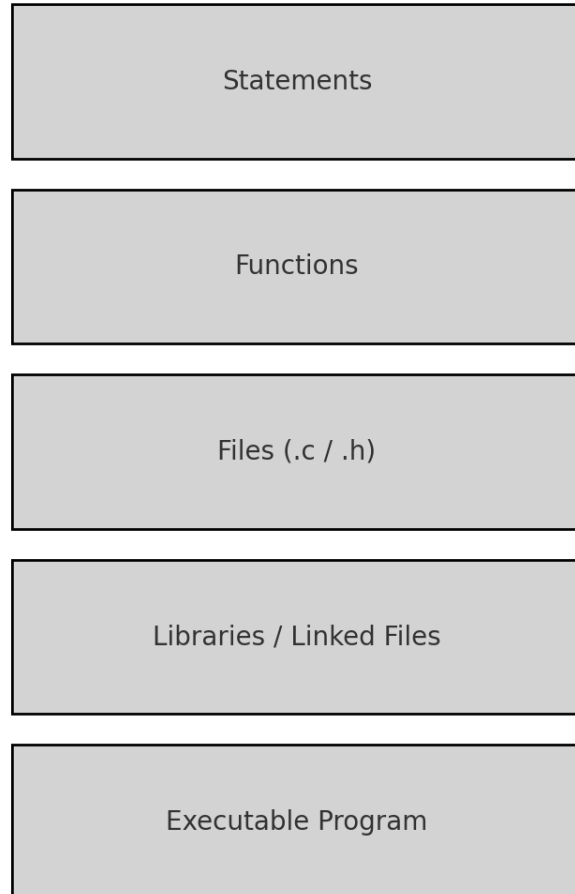
C (Structured)	Java (OO)	Haskell (Functional)
	Statements	Expressions
Statements	Methods	Functions
Functions	Classes / Interfaces	Types / ADTs
Files (.c / .h)	Packages	Modules
Libraries / Linked Files	Modules	Packages
Executable Program	Application (JAR/WAR)	Program (Main Module)
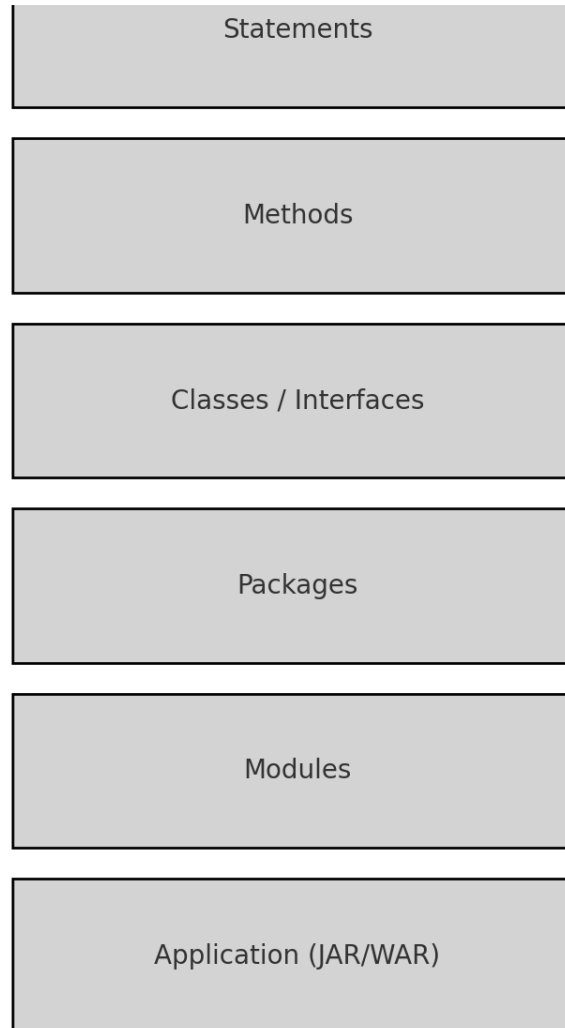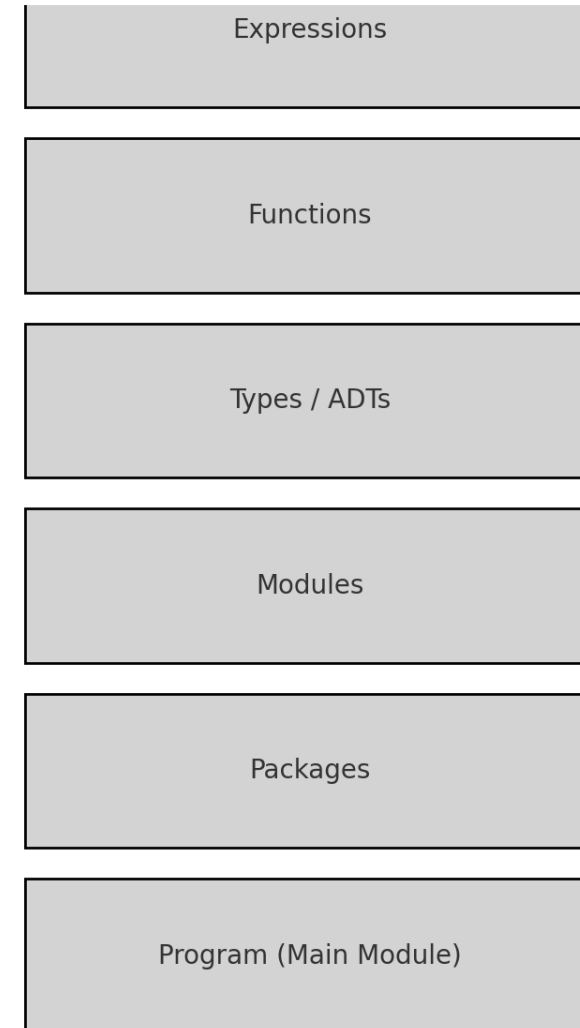
Larger Program Structures

- Most programming languages have some sort of module structure

  - These often correspond to subsystems or parts of an overall application

  - Implementation of the design concepts we saw earlier

  - Important for scaling up the size of the codebase in a modular manner

- Module structures generally have some sort of dependency map to describe how the modules are related

  - Example is a golang mod file.

```go
module github.com/acme/payments     // the module path (often a repo URL)

go 1.22                             // Go version used

require (                          // list of dependencies
    github.com/google/uuid v1.6.0
    golang.org/x/crypto v0.27.0
    github.com/julienschmidt/httprouter v1.3.0
)

replace github.com/google/uuid v1.6.0 => github.com/acme/uuid v1.6.0-patched
// optional: override where a dependency comes from (useful for local forks)

exclude golang.org/x/crypto v0.26.0
// optional: tell Go not to use a specific version
```

Containerization and Microservices

- The rise of microservices has led to some new ideas on program architecture

    - Code is written in small units called services

    - Deployed in runtime containers, often Docker

    - The actual application architecture is not stored in the codebase

    - But stored in some orchestration tool like Kubernetes

- Microservices represent an alternative approach to application architecture

    - Not suitable for all types of applications, but essential for those that are going to be deployed as microservices

The 12 Factor App Methodology

- Methodology for building microservices SaaS 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 codebase 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.
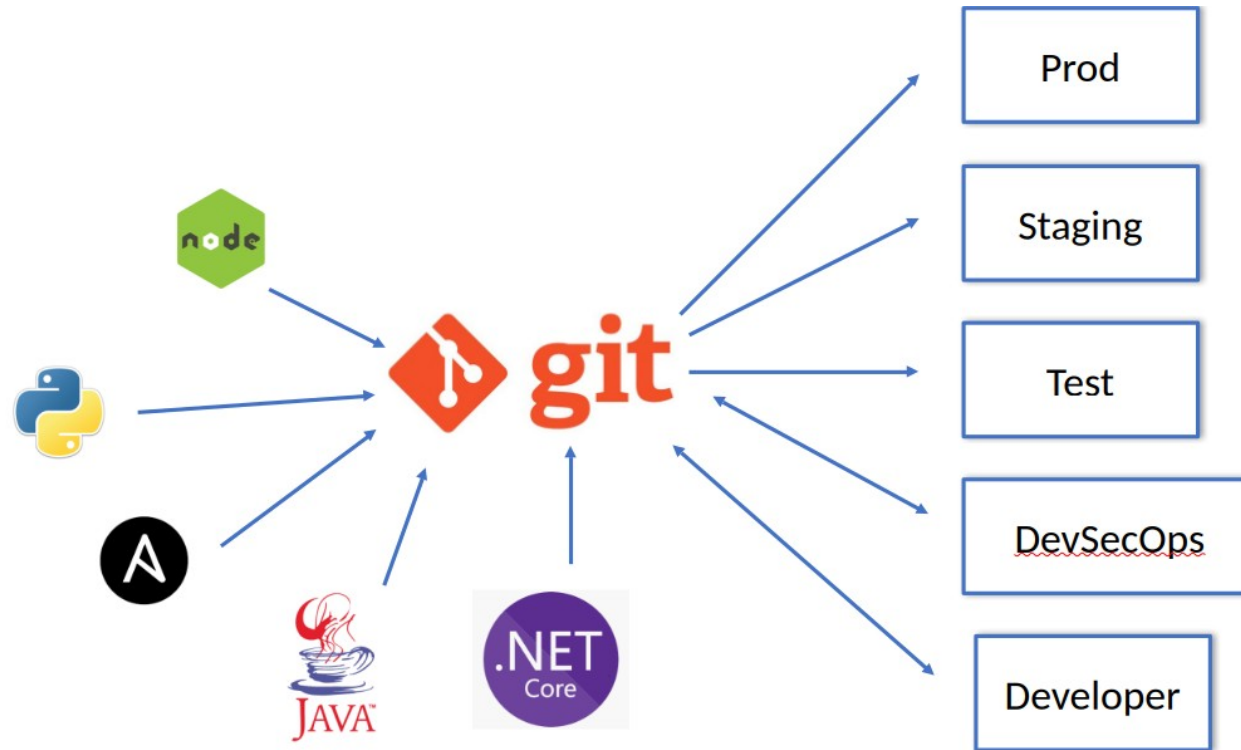
I. One Codebase, Many Deploys

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

Unit tests

Configuration data

External dependencies

Documentation

Application code
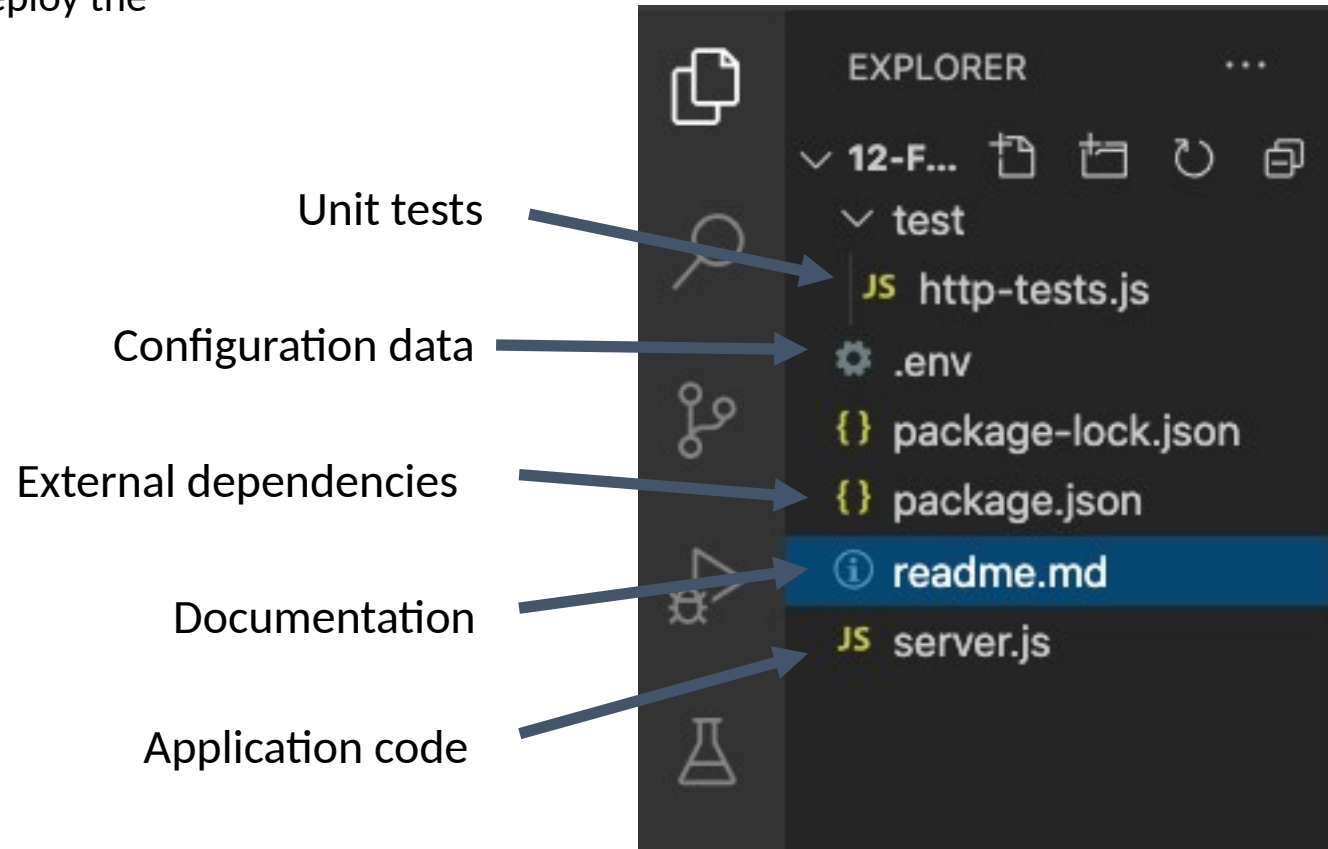
II. Dependencies

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.

```json
"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"
}
```

```
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
```

node: package.json   python: requirements.txt

ProTech
protechtraining.com

III. Configuration

- The app configuration is anything likely to vary between deploys

    – Resource handles and references

    – Credentials

    – Environment-dependent values, such as the hostname.

- 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
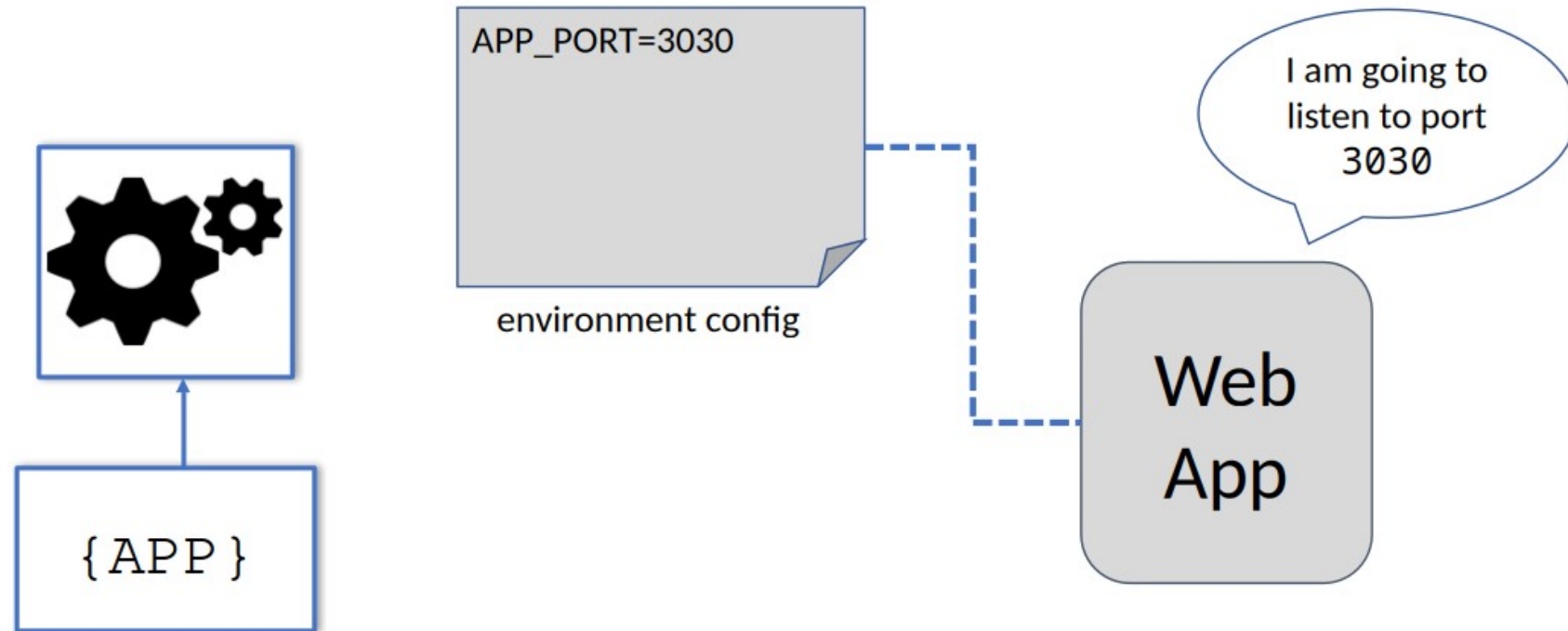
III. Configuration

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.
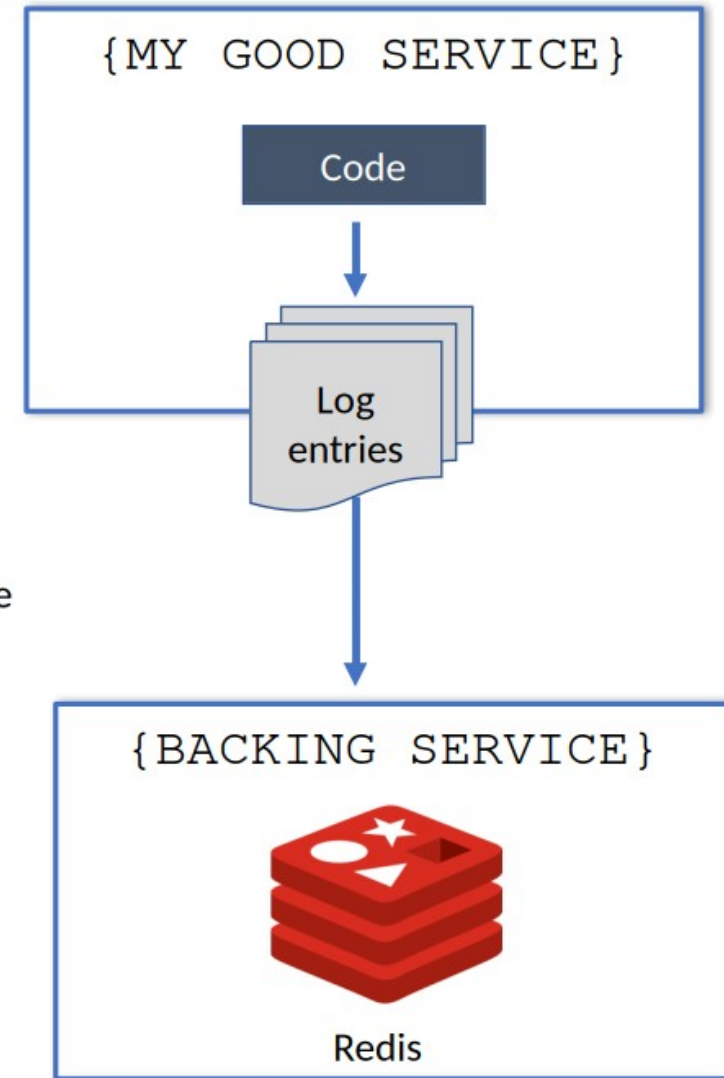
IV. Backing Services

{MY BAD API}

Log entries
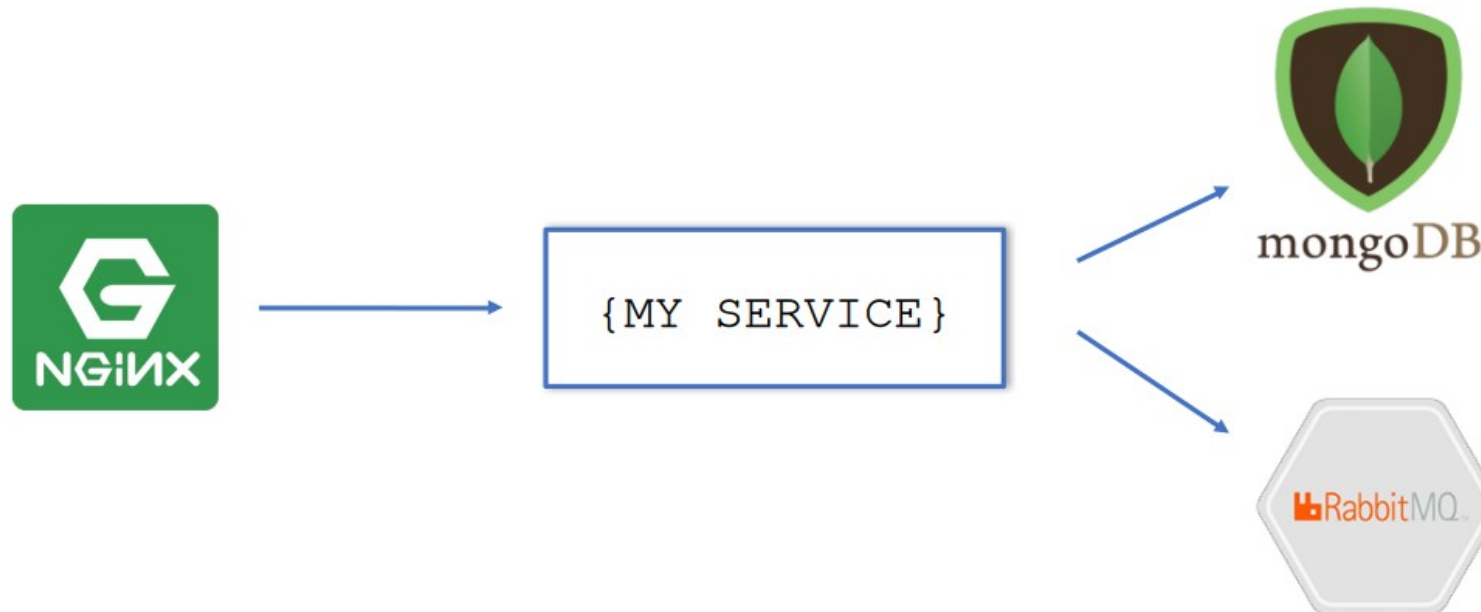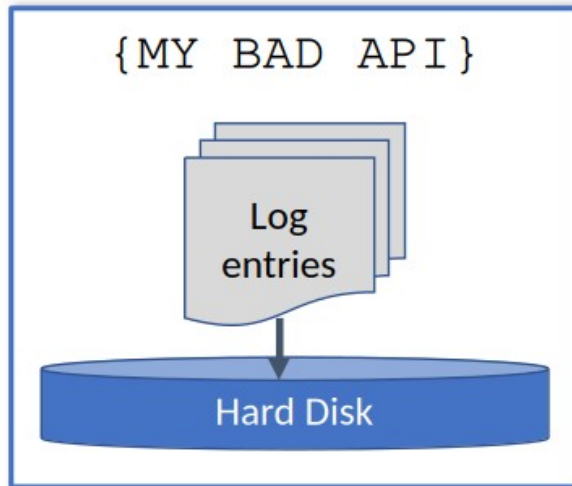
Hard Disk

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.

{MY GOOD SERVICE}

Code

Log entries

{BACKING SERVICE}

Redis

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
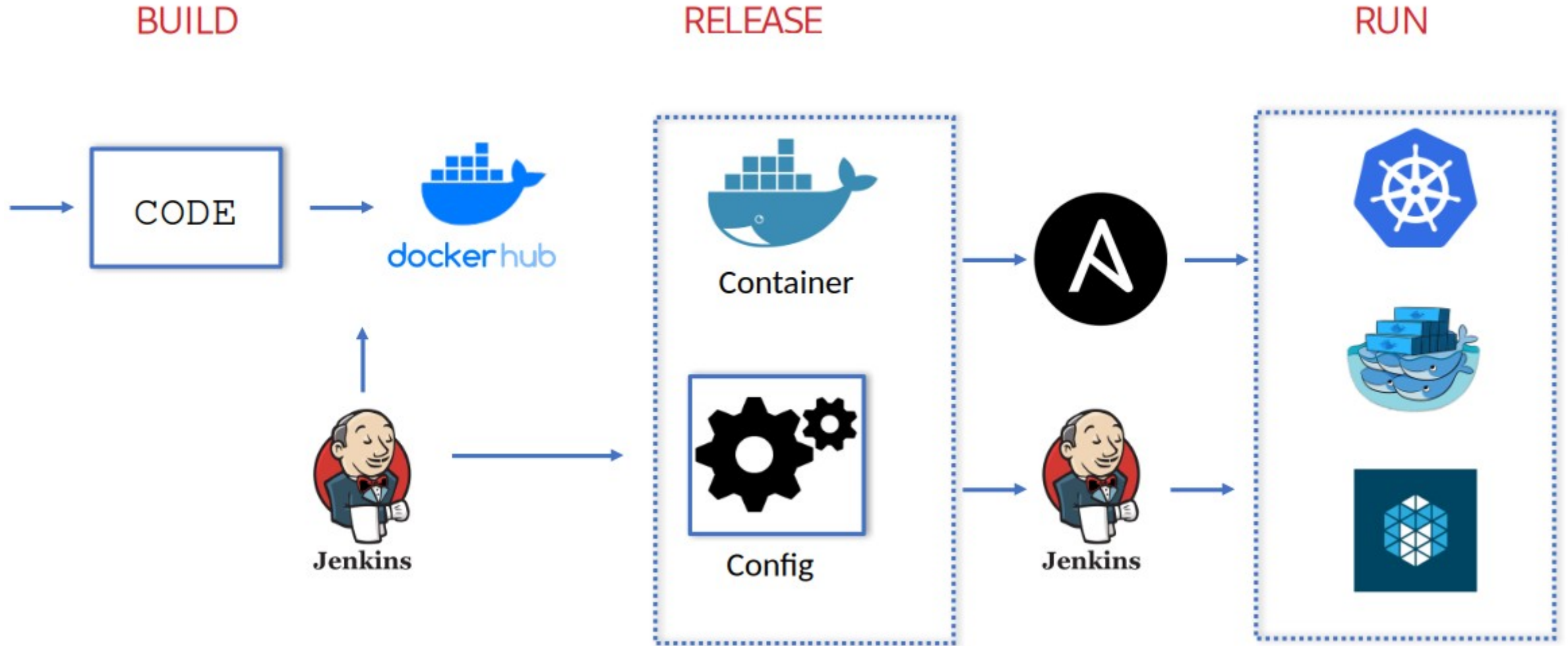
VI. Processes

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
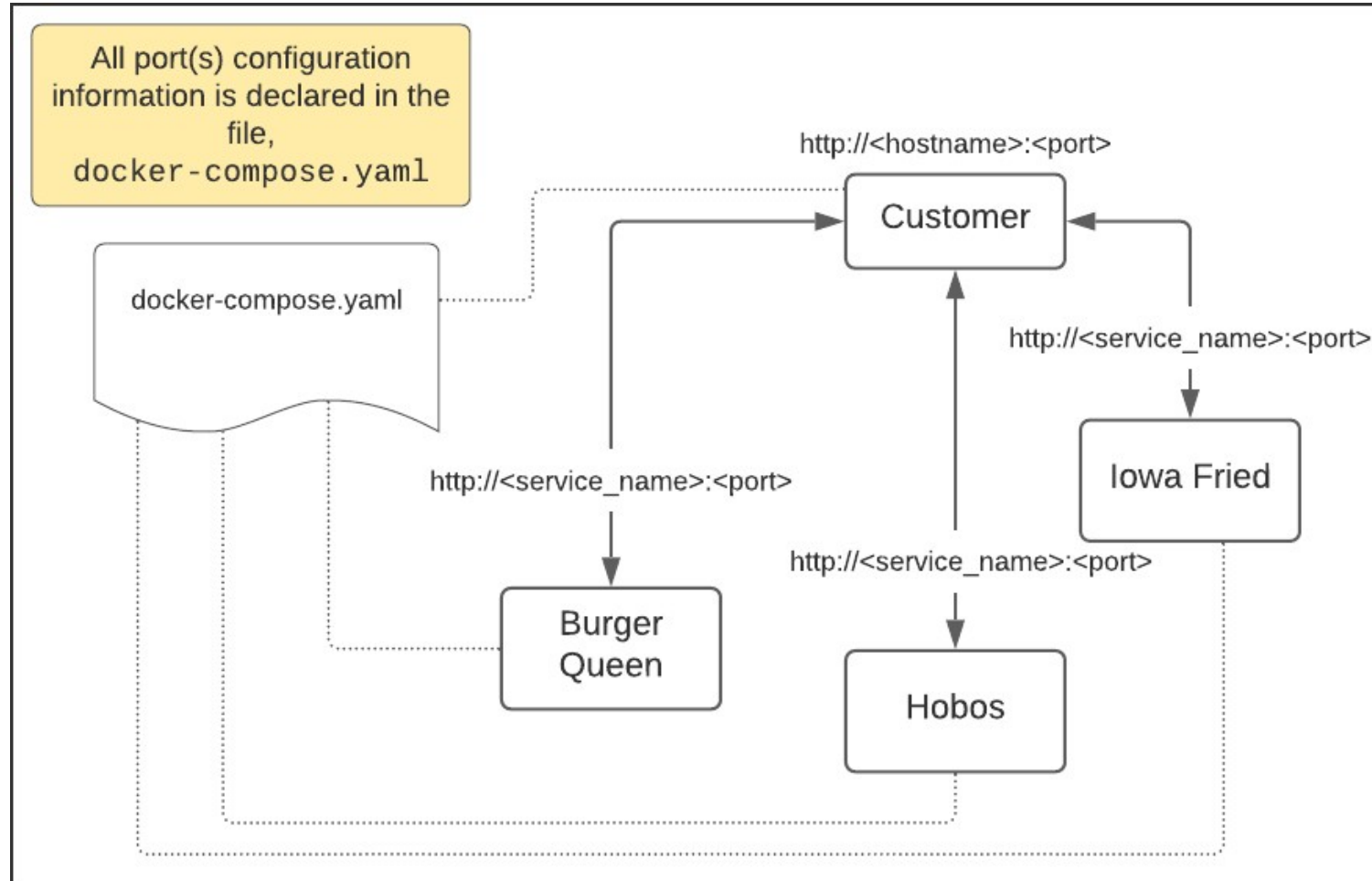

Container

Stateless
Application

Storage
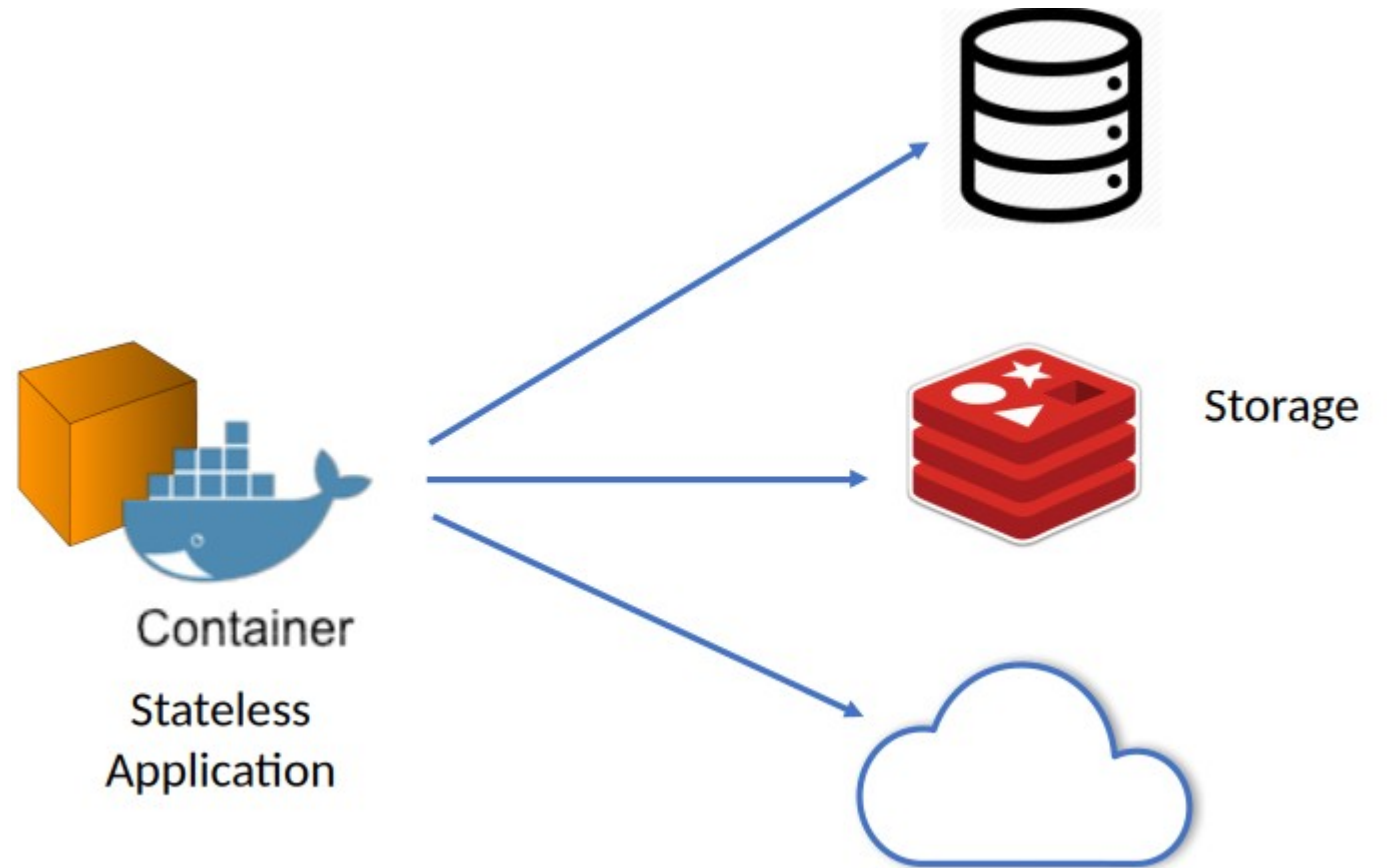
VII. Port Binding

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.
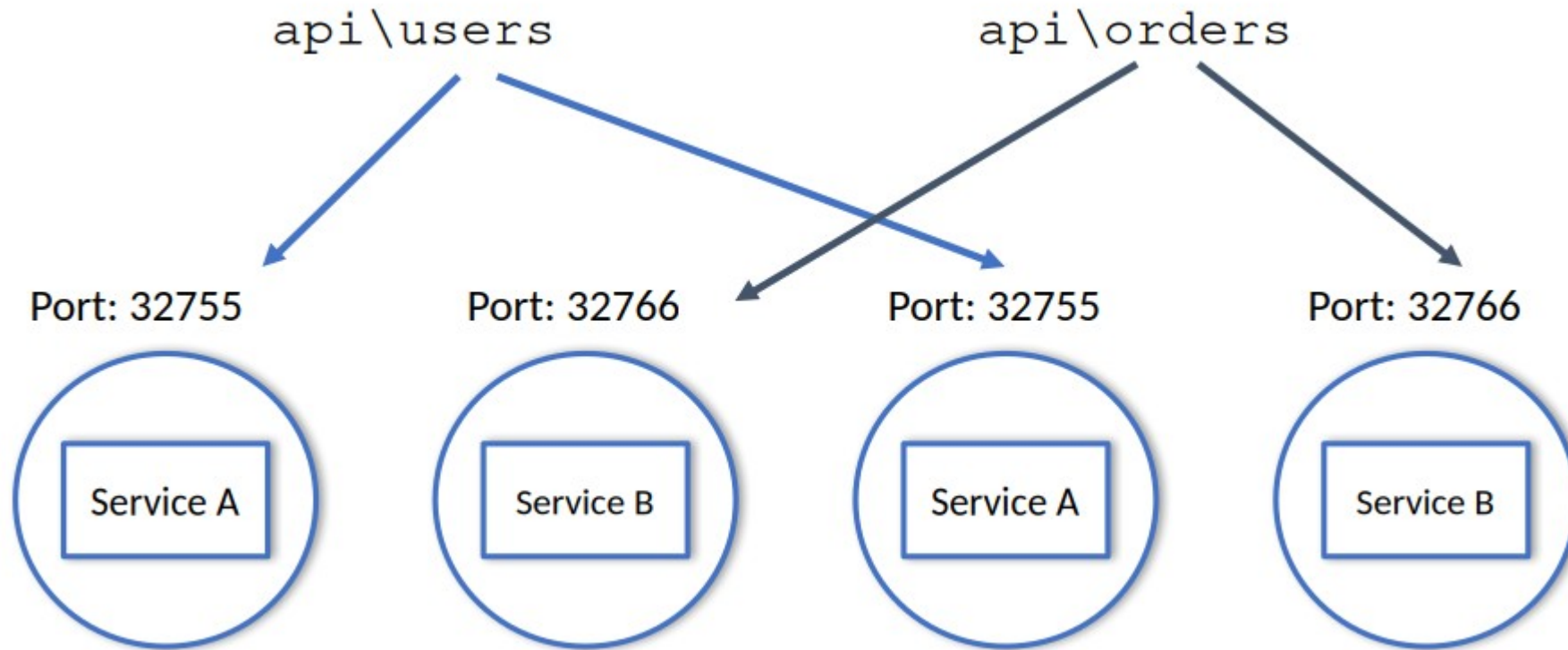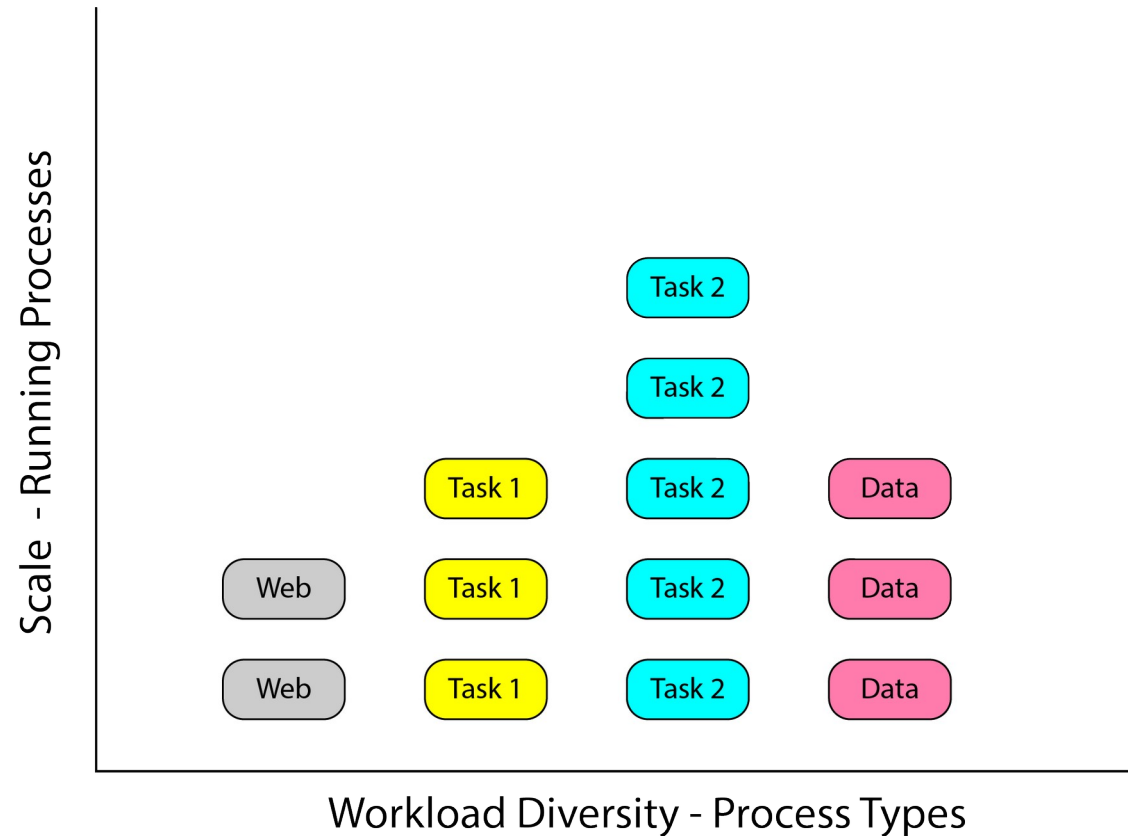
VII. Port Binding

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 shut down 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 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
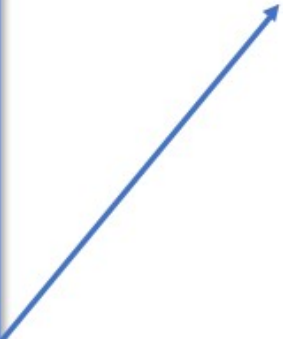
XI. Logs as Event Streams

- 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
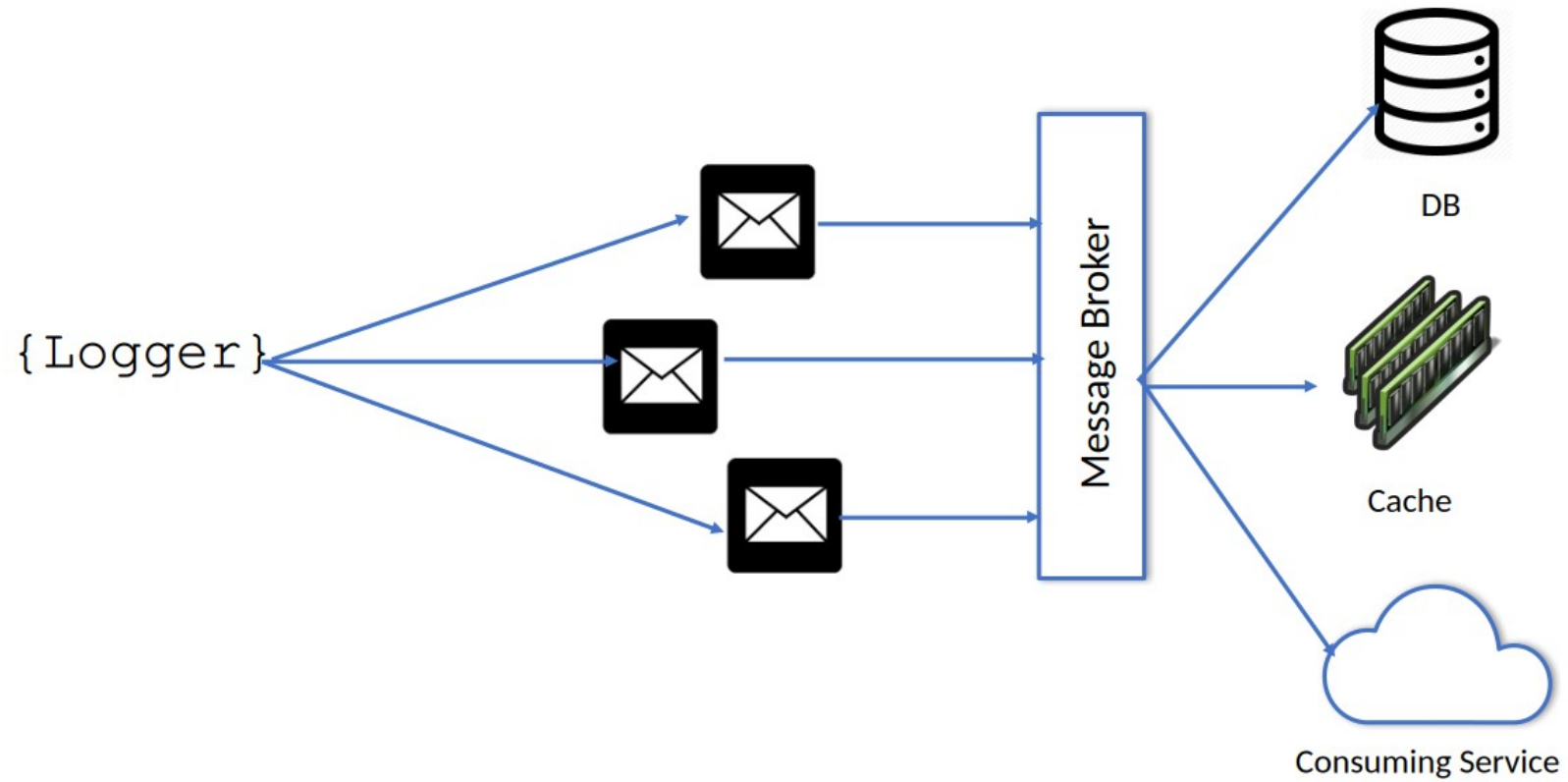
XI. Logs as Event Streams

Logging from inside a process

```
                            {APP}
{

    logURL = env.process.LOGGER_URL;
    logUser = env.process.LOGGER_USER;
    logPwd = env.process.PWD;


    const Logger = require(cloud-logger);


    logger = new Logger({logURL,logUser,logPwd});


    logger.info(`Starting at ${new Date()}`);


}
```
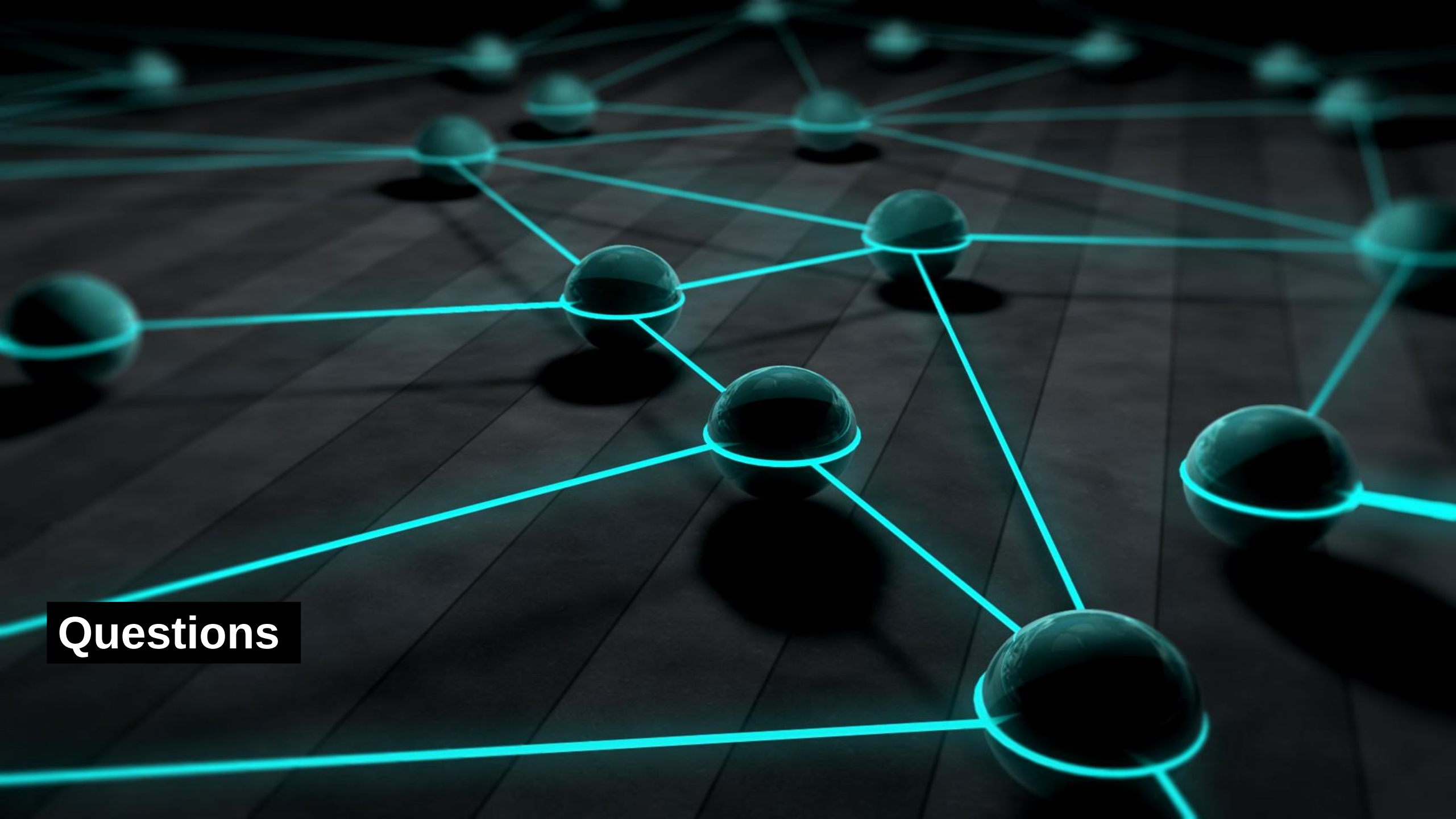
Logging Service

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