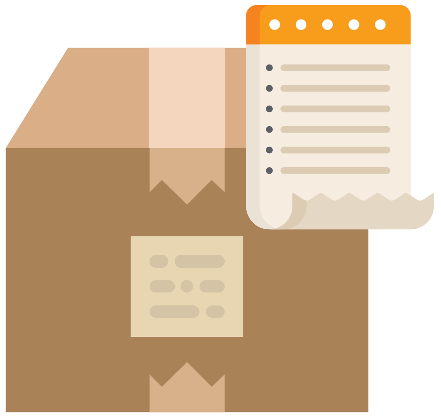
Compute Evolved Week

Building Microservices with the 12 Factor App Pattern on AWS

Nathan Peck
Developer Advocate, Container Services

12 Factor App Principles



+

Microservice Principles

=

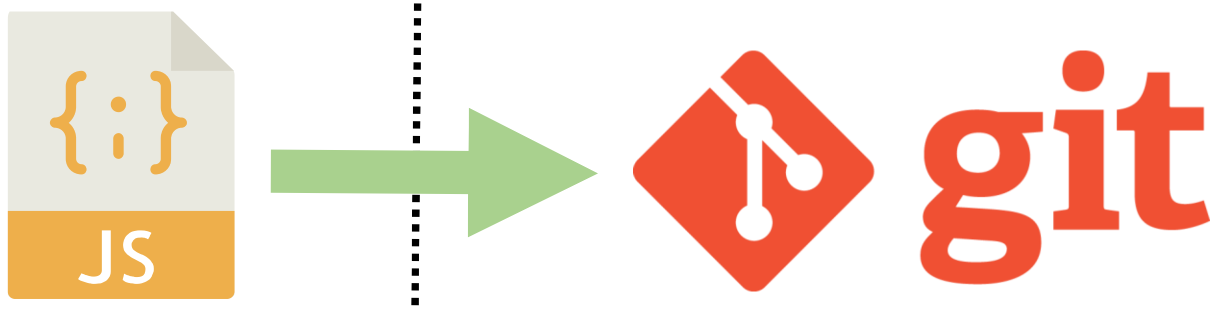
Great, Scalable Architecture

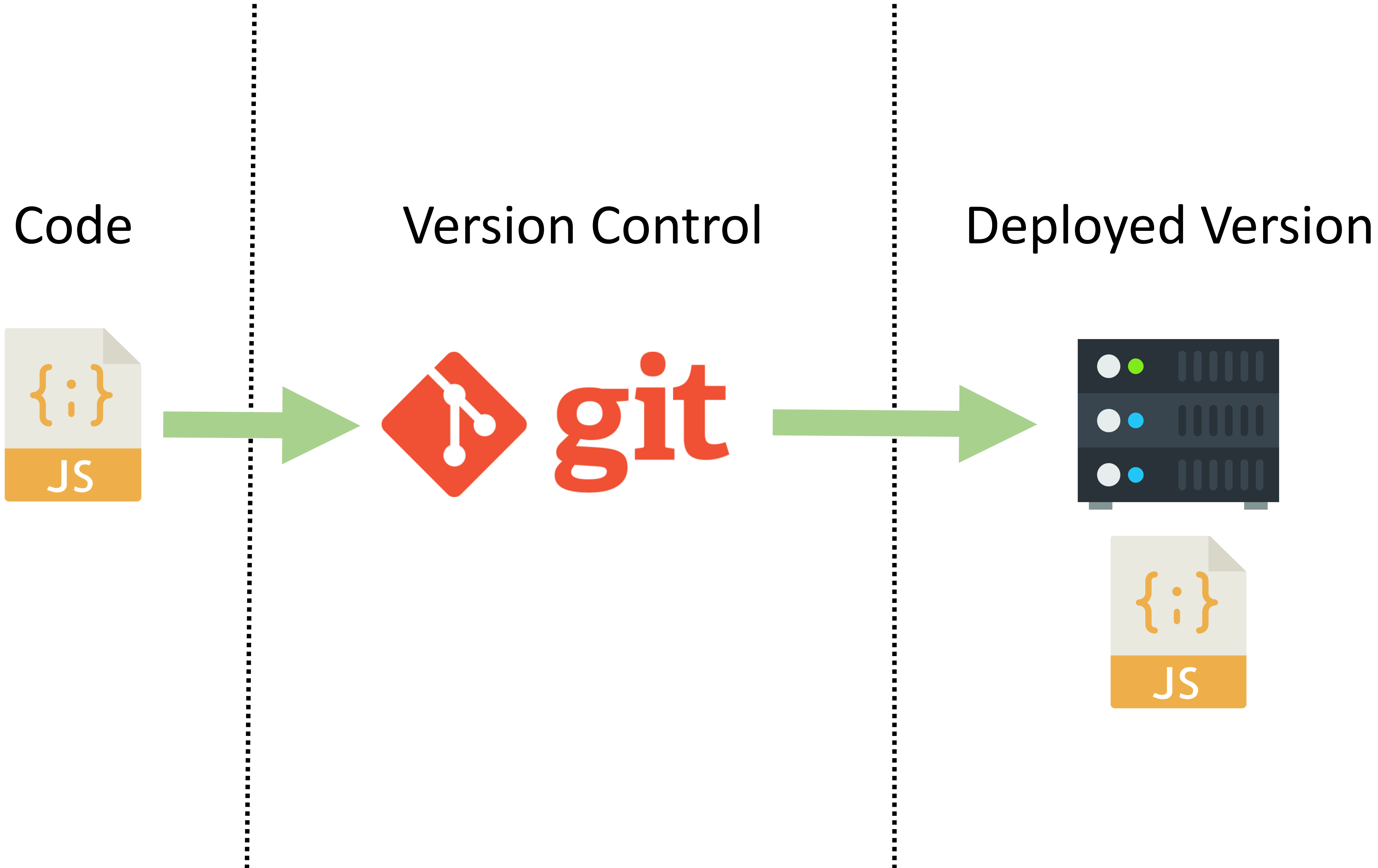
12 Factor Application: Codebase

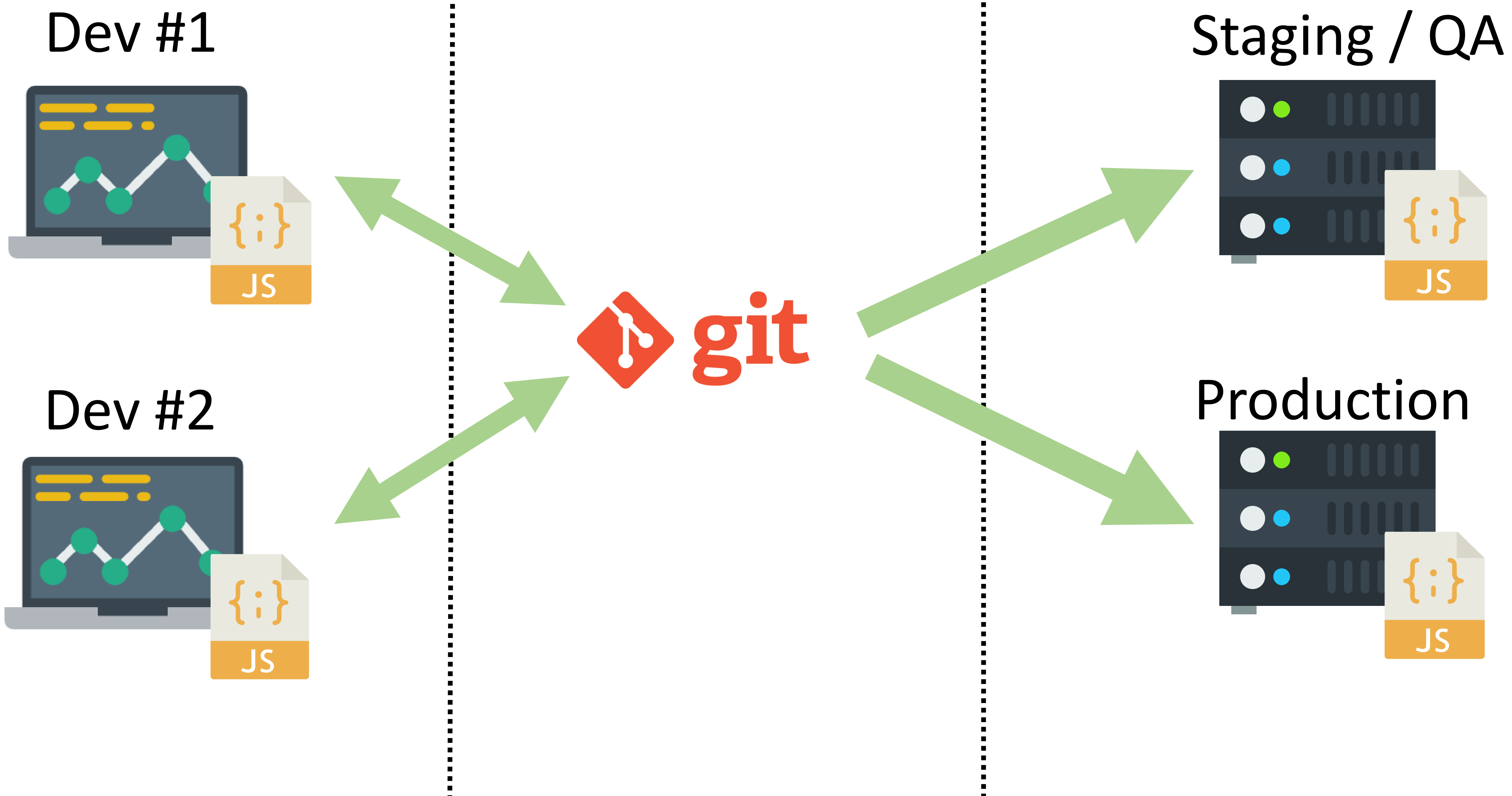
Code

Code

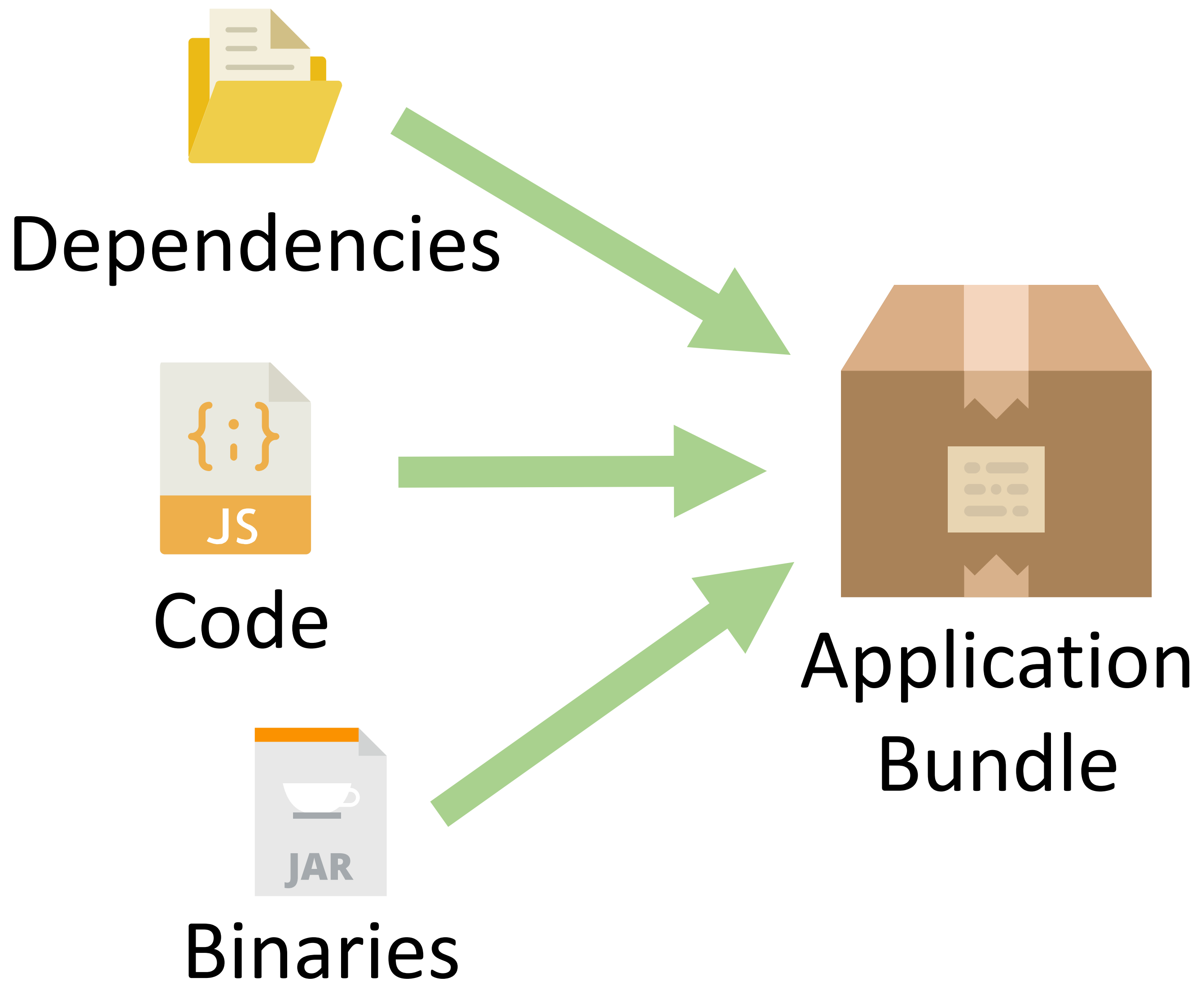
Version Control




Code
Version Control
Deployed Version
JS
git
JS

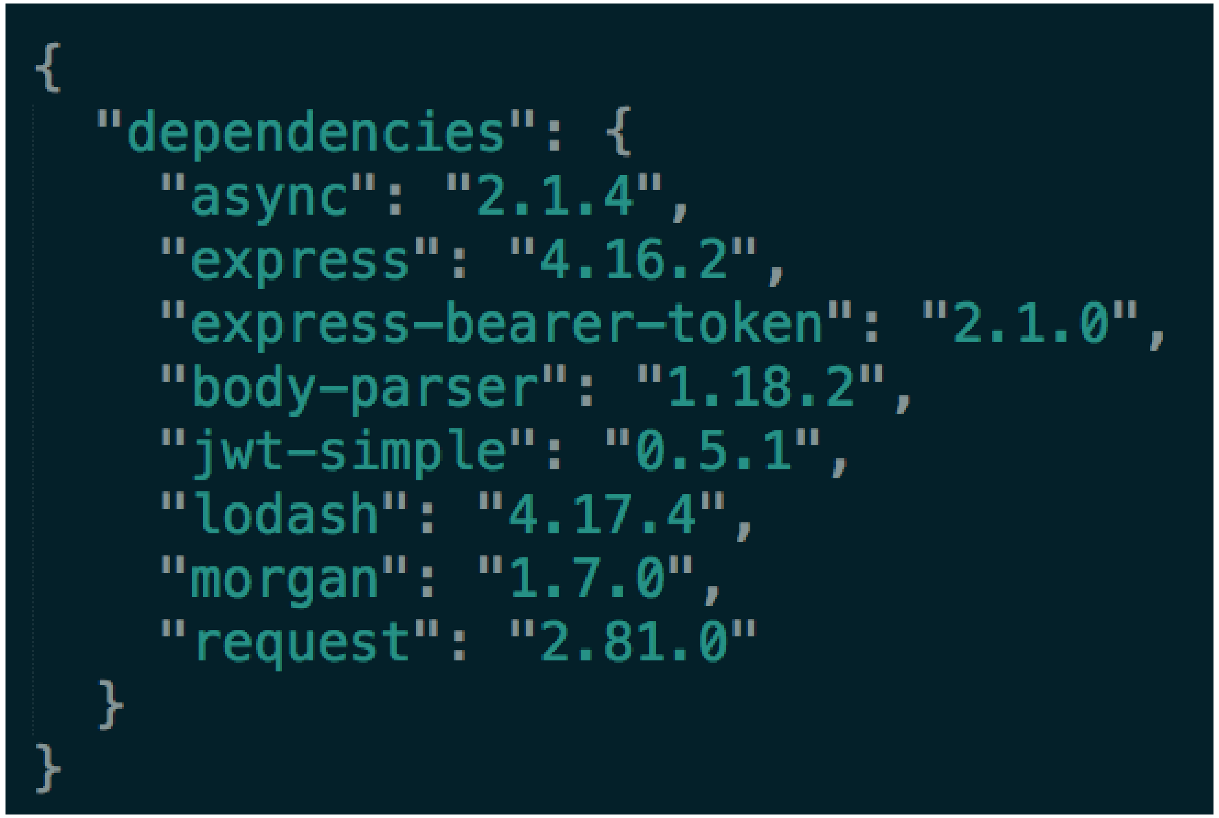

Dev #1
JS
Dev #2
JS
git
Staging / QA
JS
Production
JS

12 Factor Application: Dependencies


Dependencies
JS
Code
JAR
Binaries
Application Bundle

Dependency Declaration: Node.js

`package.json`

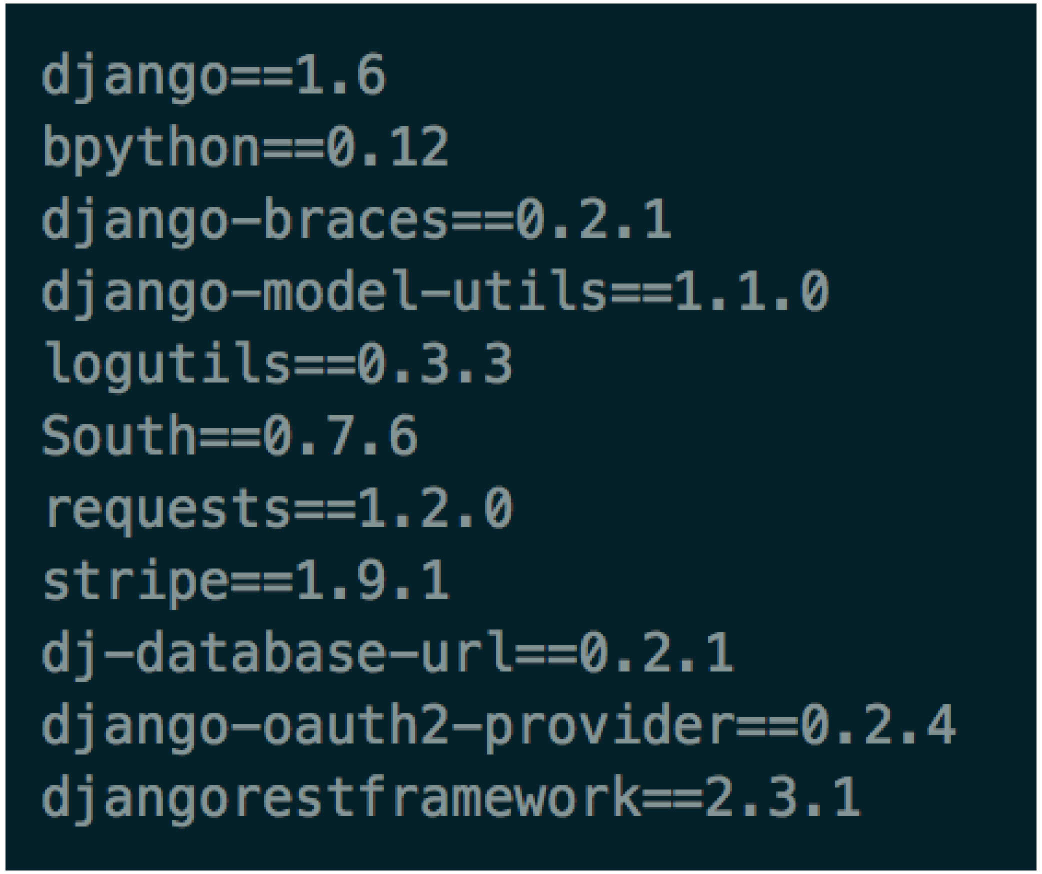


```json
{
  "dependencies": {
    "async": "2.1.4",
    "express": "4.16.2",
    "express-bearer-token": "2.1.0",
    "body-parser": "1.18.2",
    "jwt-simple": "0.5.1",
    "lodash": "4.17.4",
    "morgan": "1.7.0",
    "request": "2.81.0"
  }
}
```

`npm install`

Dependency Declaration: Python

`requirements.txt`

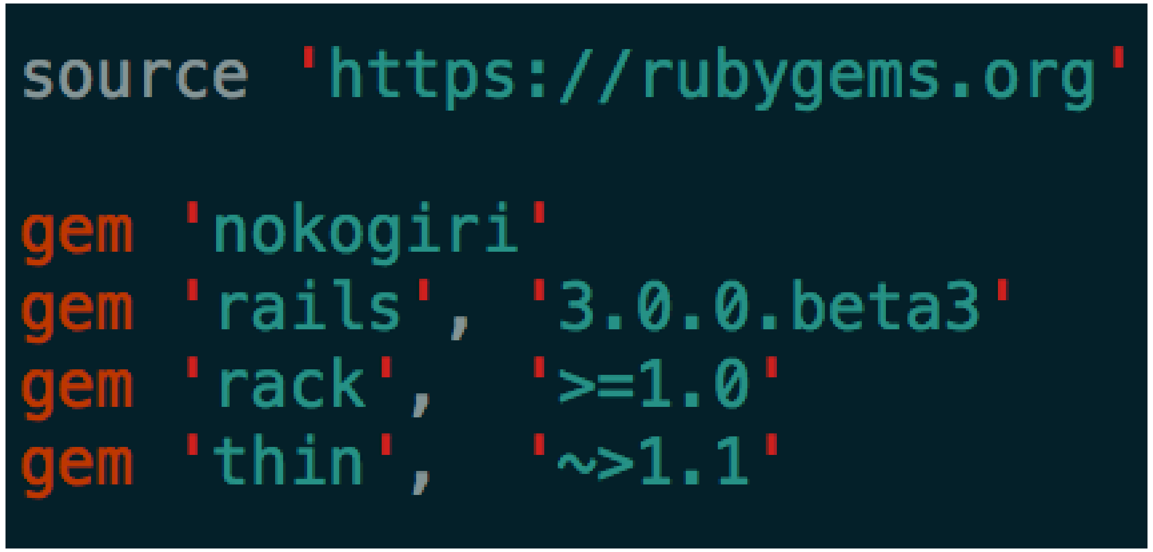


```
django==1.6
bpython==0.12
django-braces==0.2.1
django-model-utils==1.1.0
logutils==0.3.3
South==0.7.6
requests==1.2.0
stripe==1.9.1
dj-database-url==0.2.1
django-oauth2-provider==0.2.4
djangorestframework==2.3.1
```

`pip install`

Dependency Declaration: Ruby

`Gemfile`



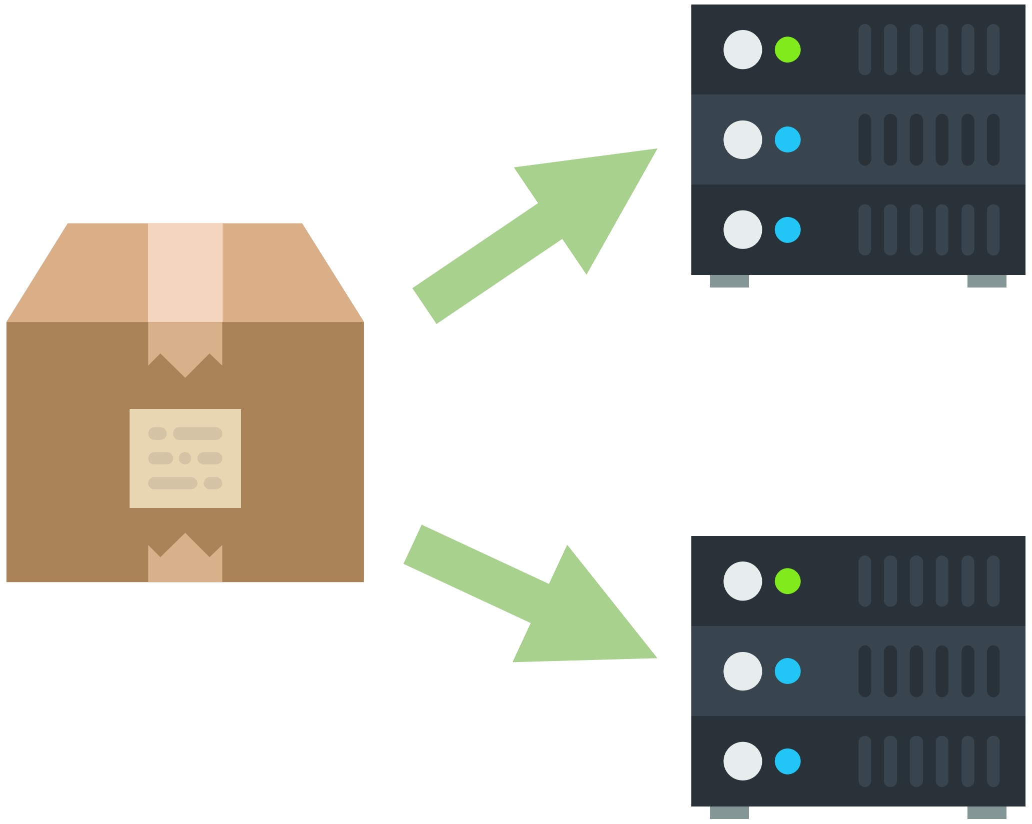
```
source 'https://rubygems.org'

gem 'nokogiri'
gem 'rails', '3.0.0.beta3'
gem 'rack',  '>=1.0'
gem 'thin',  '~>1.1'
```

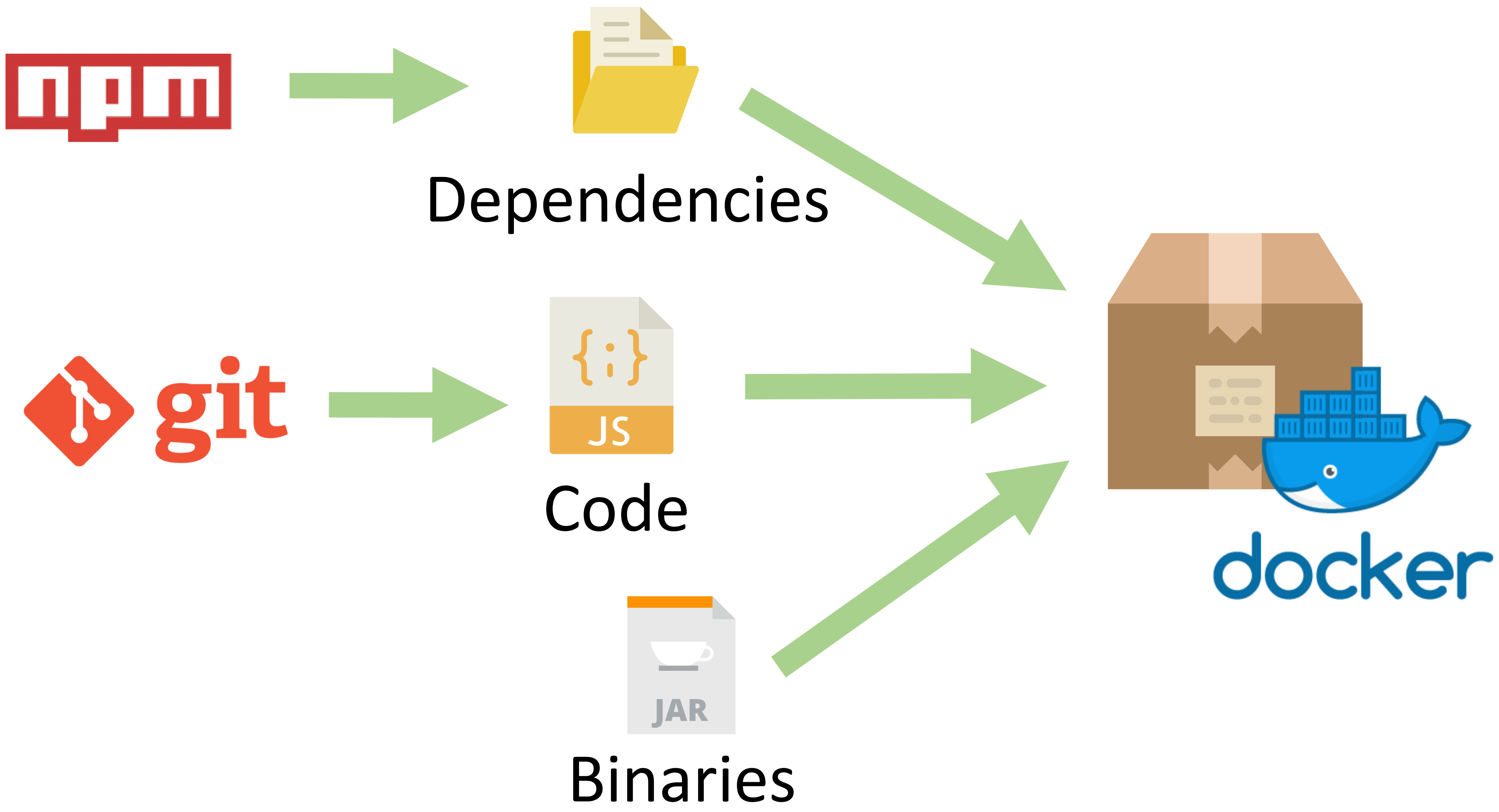
```
bundle install
```

Dependency Isolation

Never depend on the host to have your dependency.

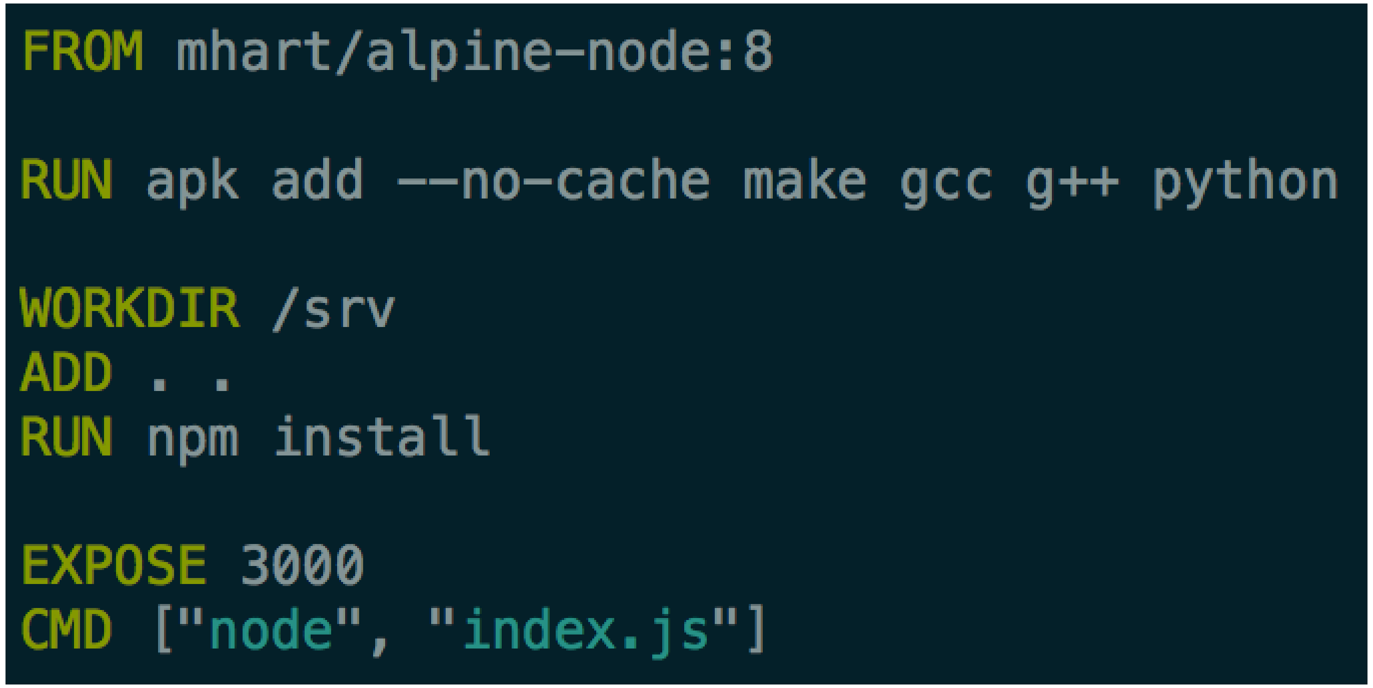
Application deployments should carry all their dependencies with them.




npm
Dependencies
git
JS
Code
JAR
Binaries
docker

Dependency Declaration & Isolation: Docker

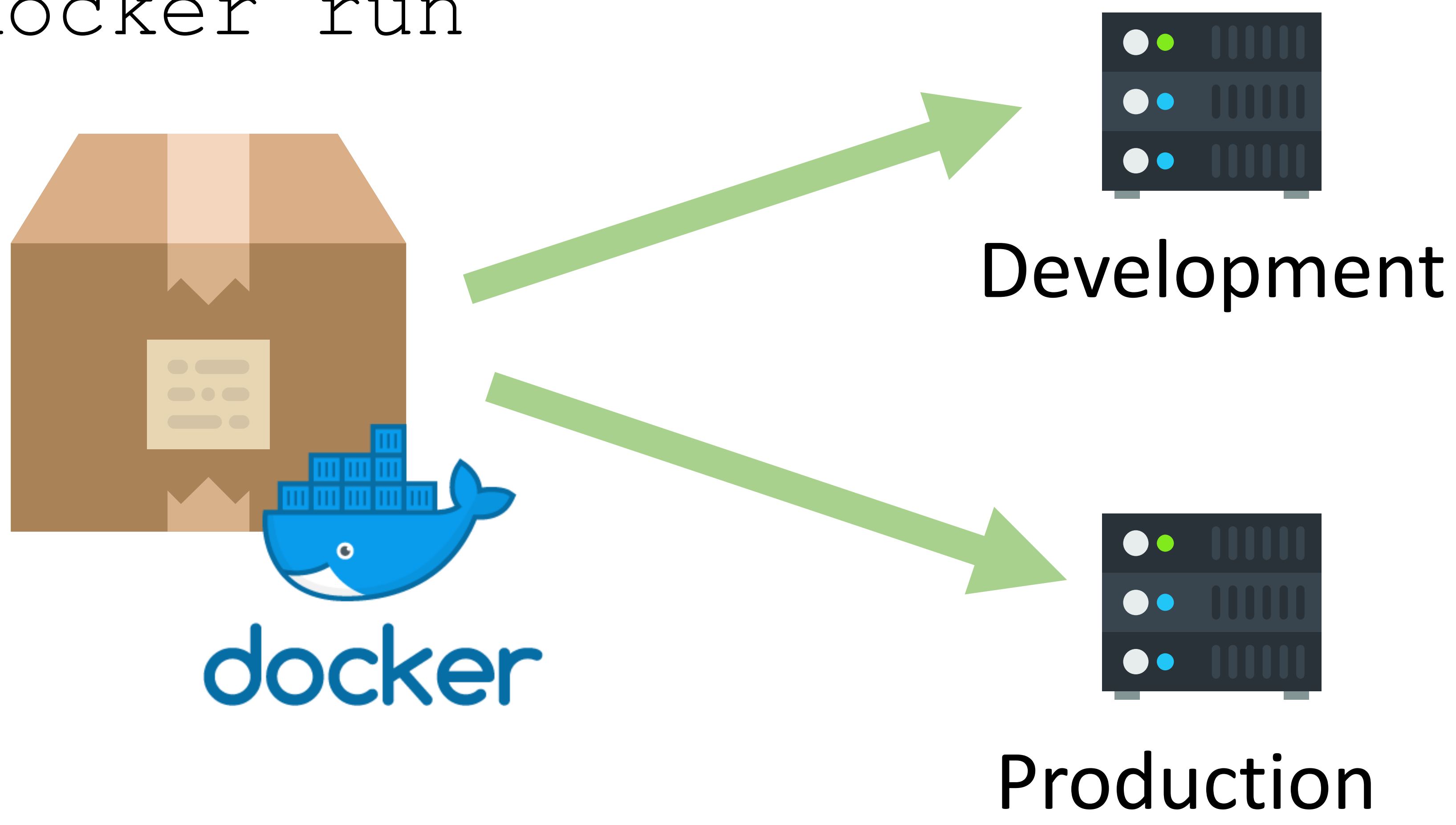
`Dockerfile`



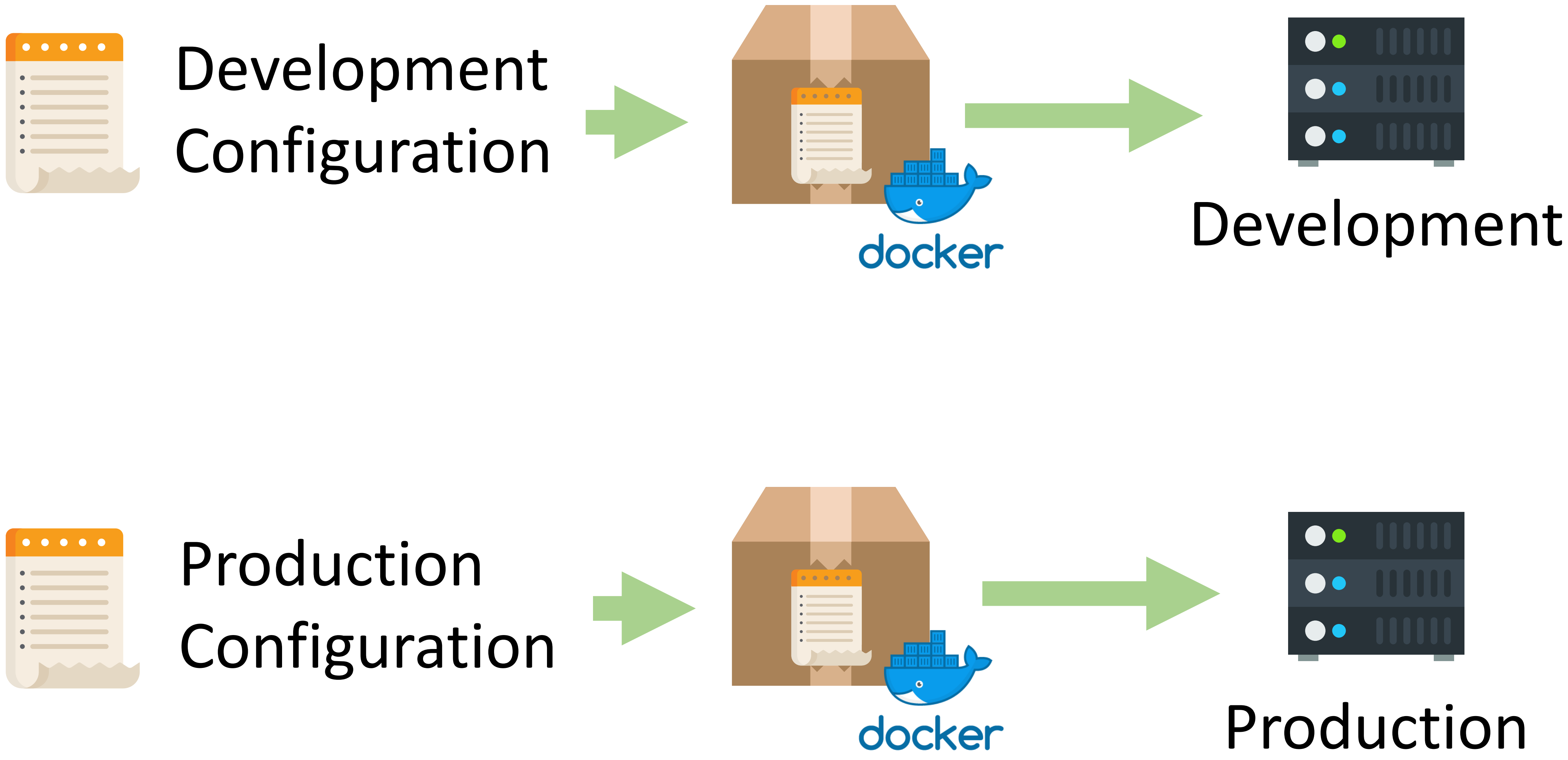
```
FROM mhart/alpine-node:8

RUN apk add --no-cache make gcc g++ python

WORKDIR /srv
ADD . .
RUN npm install

EXPOSE 3000
CMD ["node", "index.js"]
```

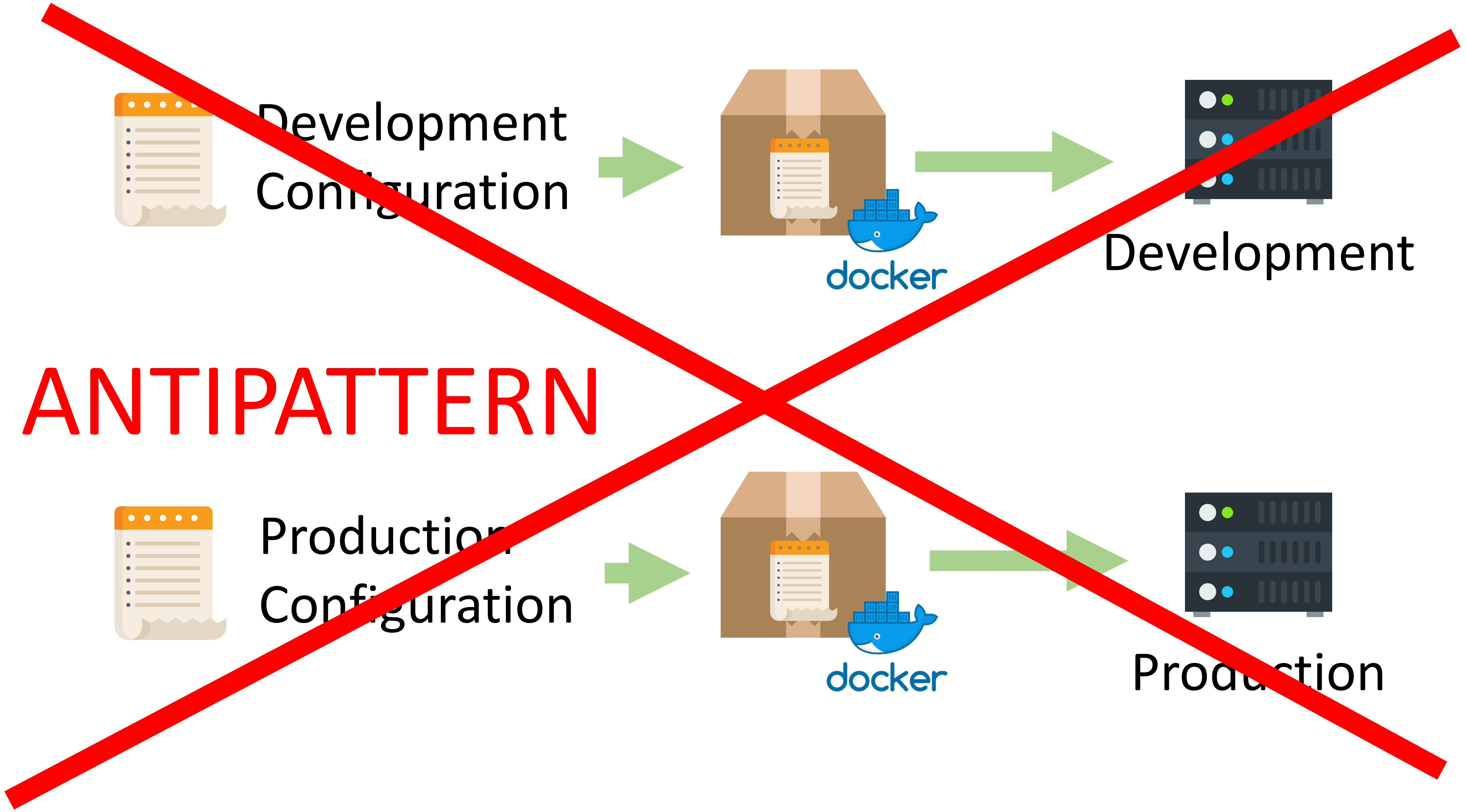
`docker build`

```
docker run
```

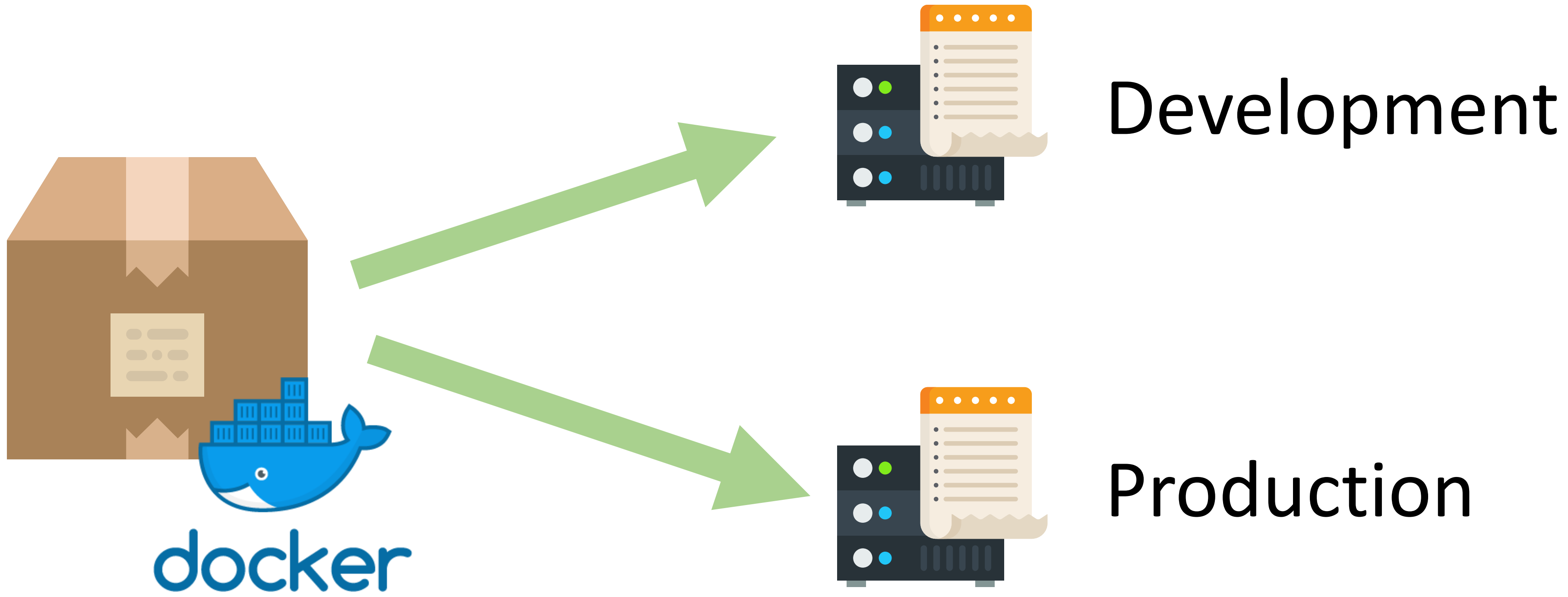


12 Factor Application: Config

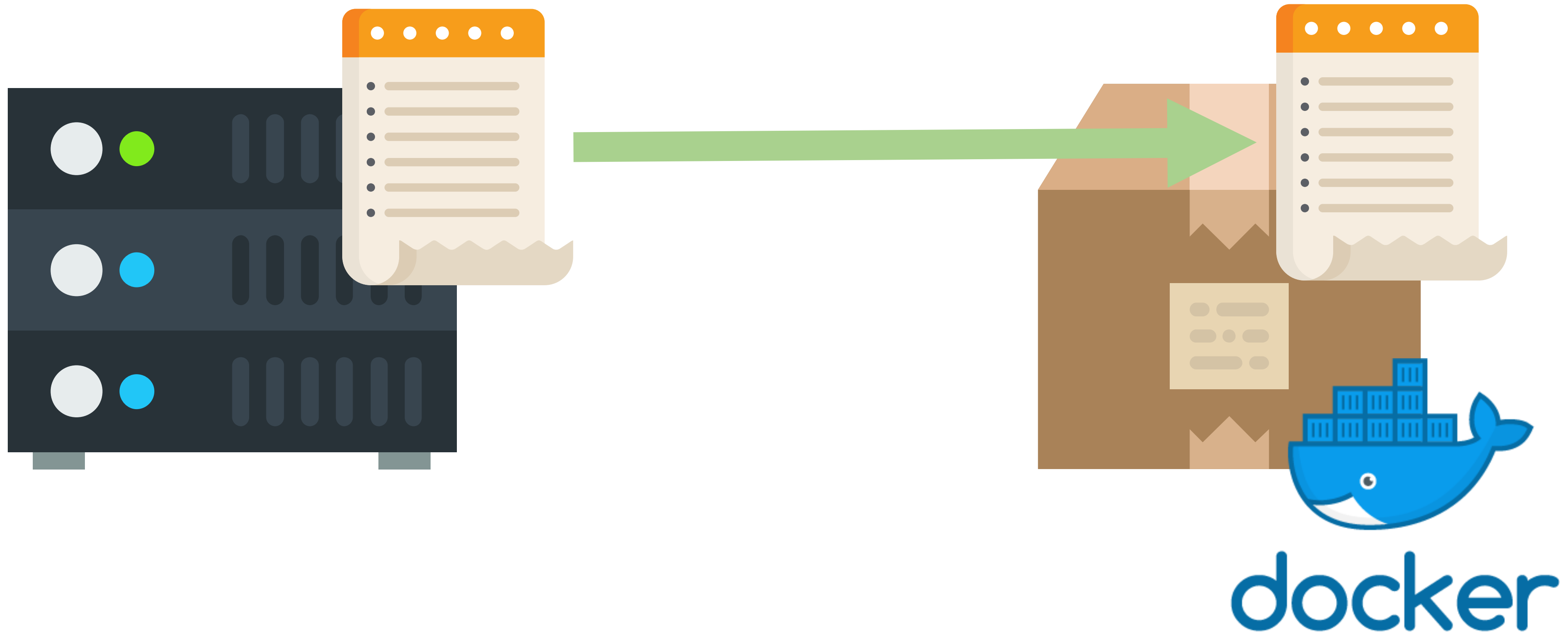

Development Configuration
docker
Development
Production Configuration
docker
Production


Development Configuration
docker
Development
ANTIPATTERN
Production Configuration
docker
Production

Same container deployed to both environments.
Configuration is part of the environment on the host.



At runtime the container gets config from the environment.



Application code pulls from the environment

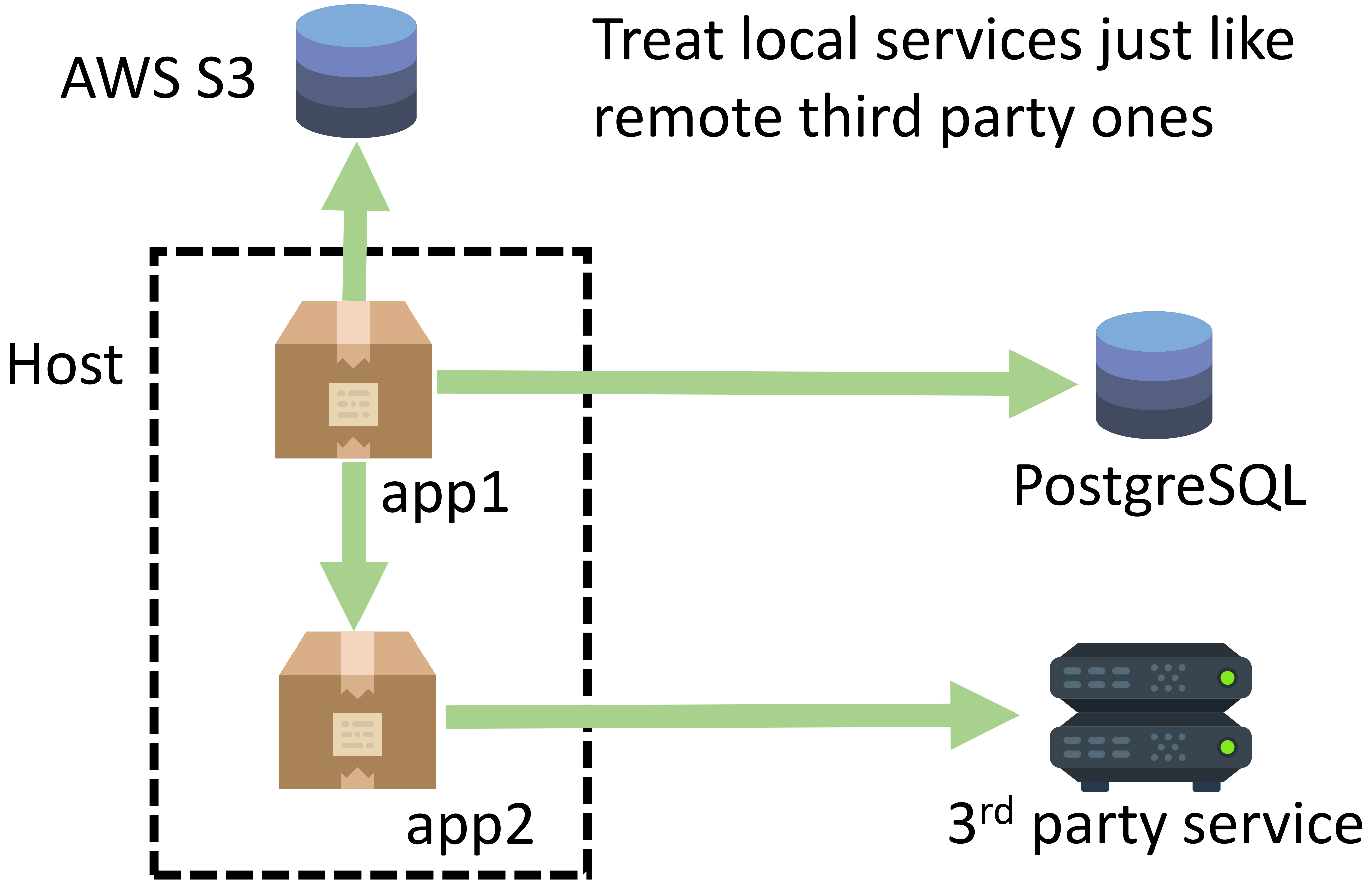
```
module.exports = {
  DATABASE: process.env.DATABASE,
  SECRET: process.env.SECRET
};
```

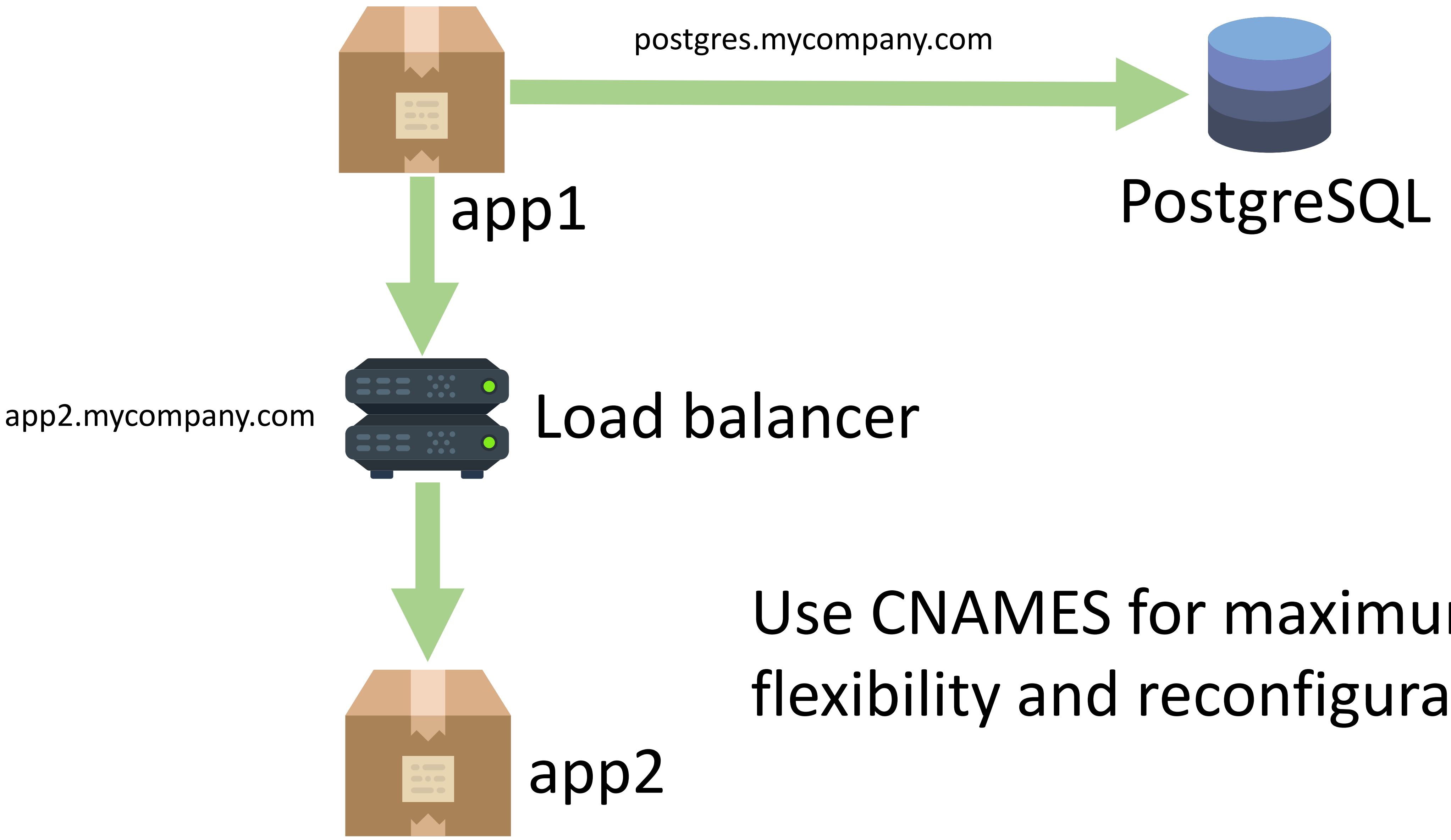
Environment is customized when docker runs a container

```
docker run -e "DATABASE=mongodb://localhost:27017" -e "SECRET=default" myapp

docker run -e "DATABASE=mongodb://db1.mycompany.com,db2.mycompany.com/
production?replicaSet=production" -e "SECRET=hunter2" myapp
```

12 Factor Application: Backing Services

Treat local services just like remote third party ones

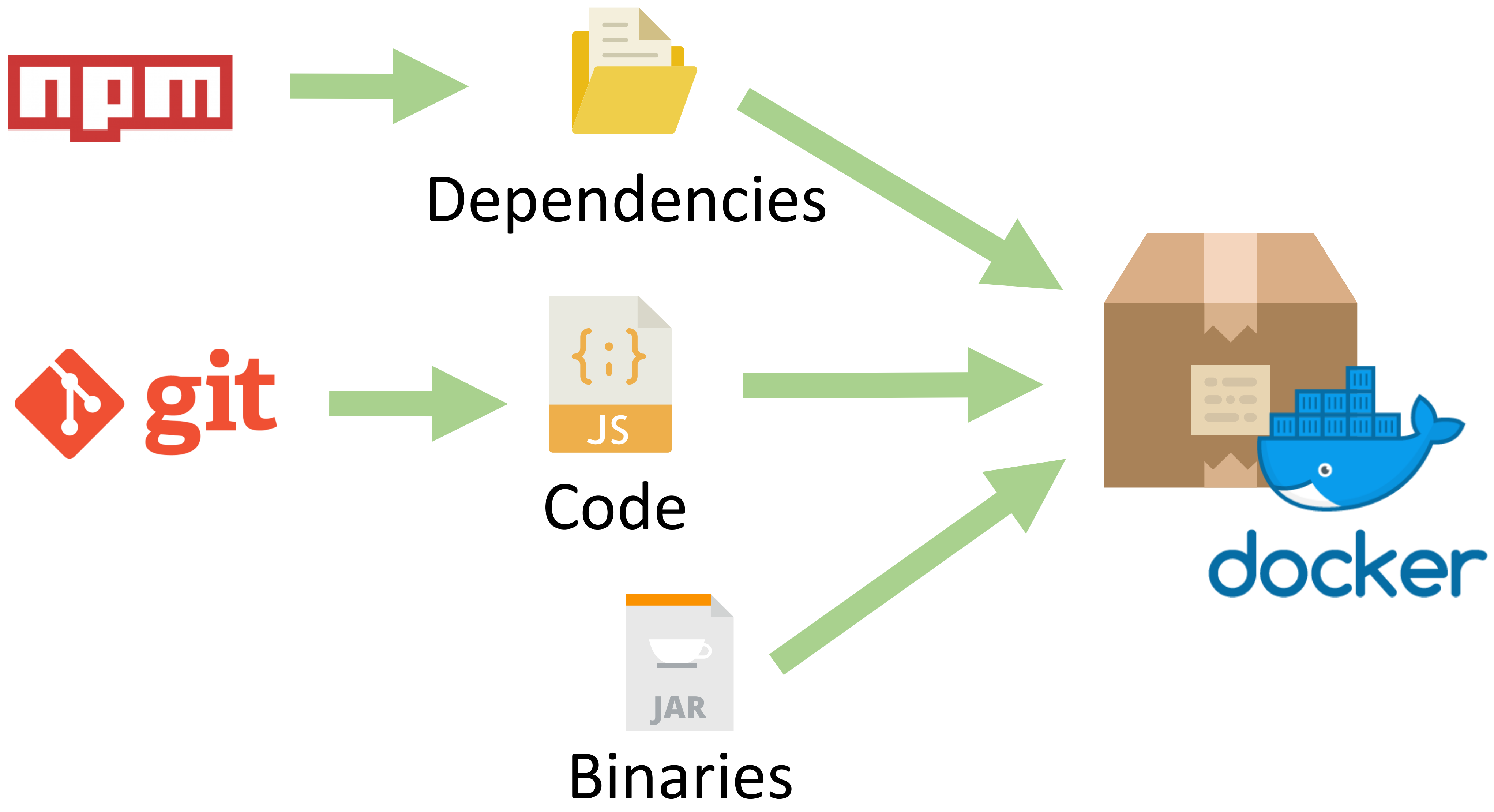




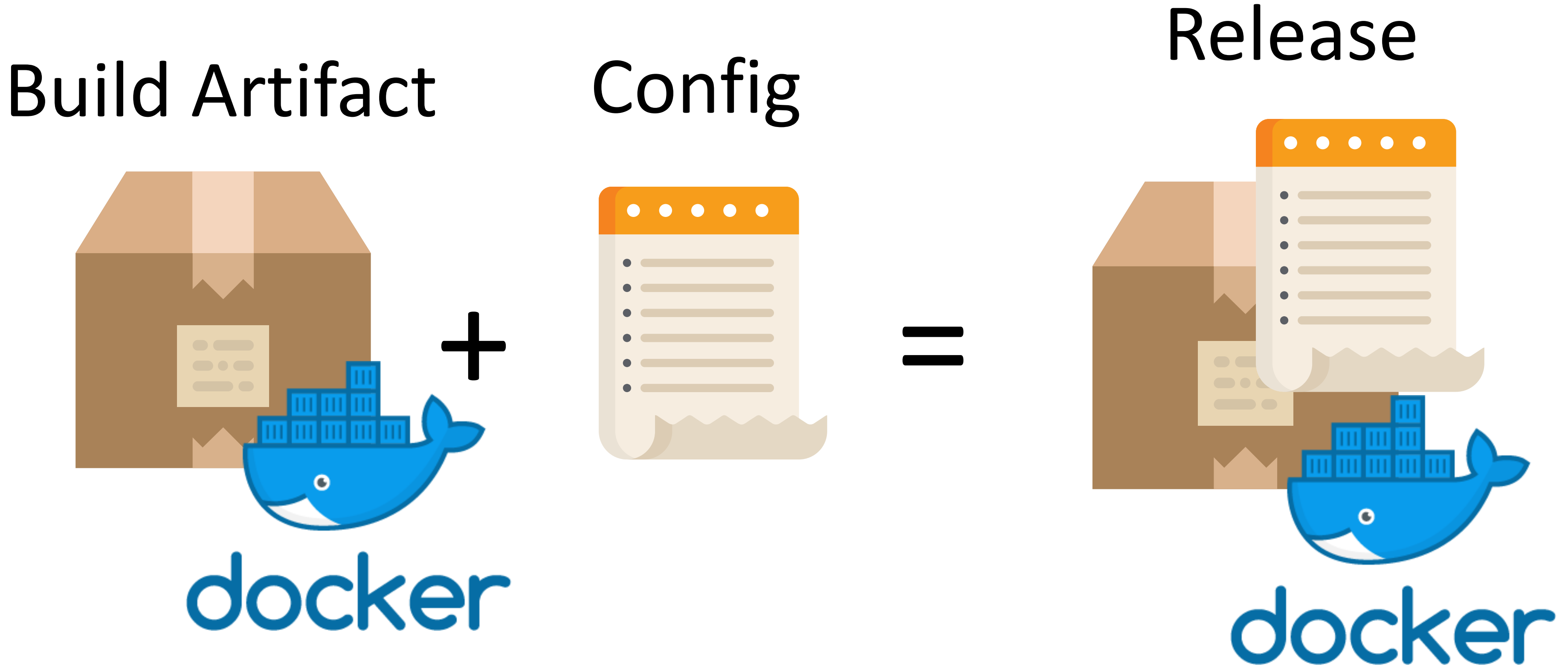
Use CNAMES for maximum flexibility and reconfigurability

12 Factor Application: Build, Release, Run

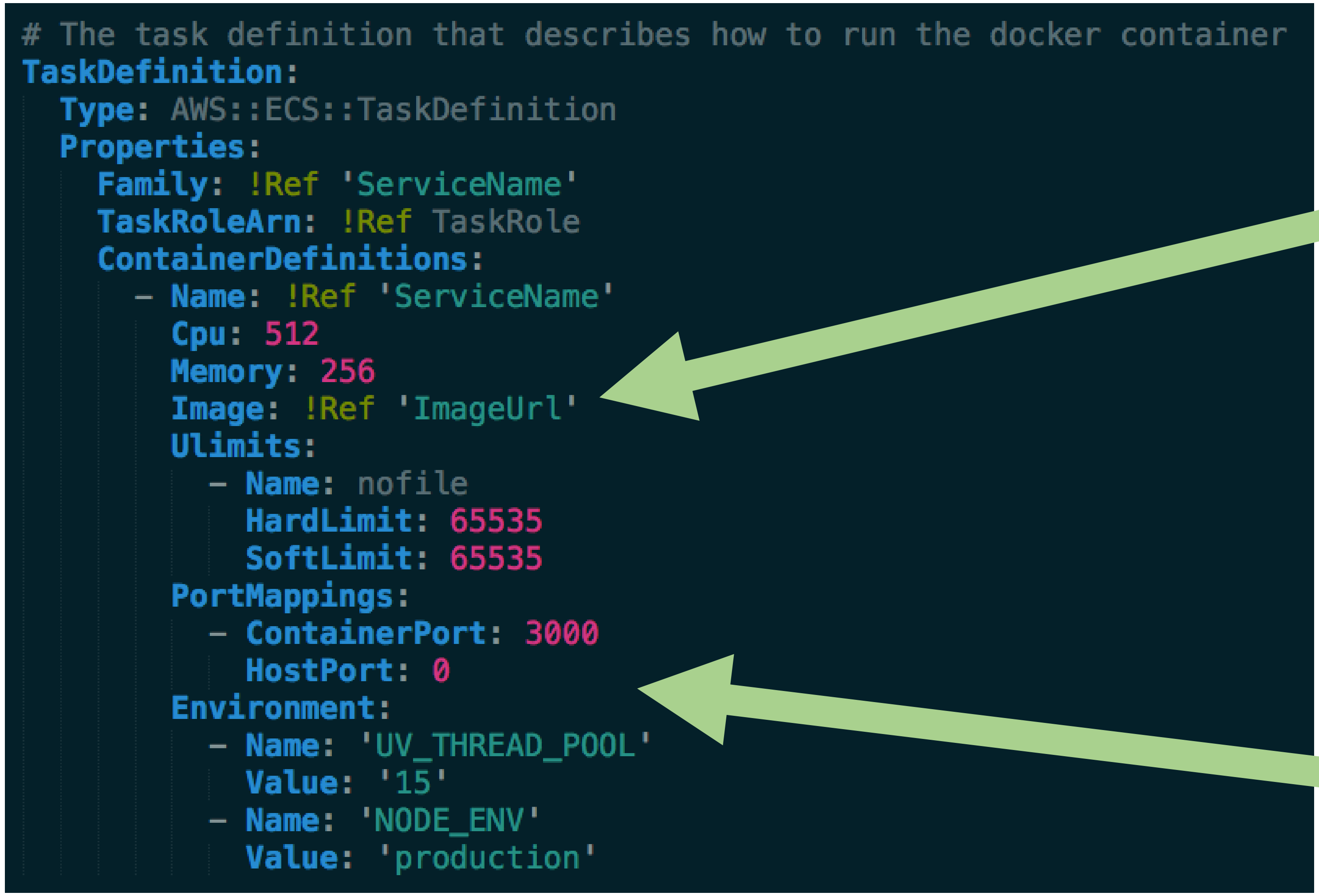
Build



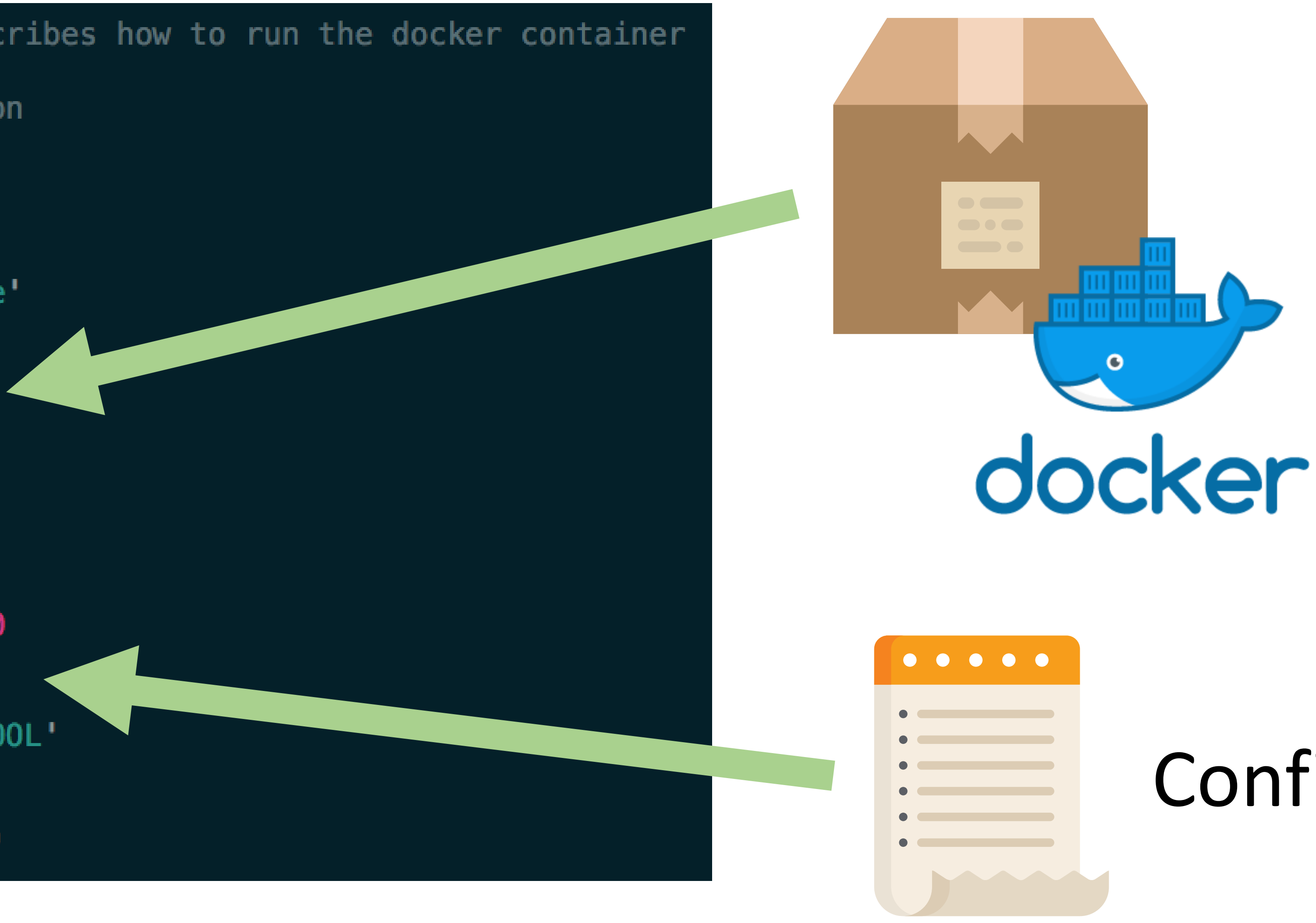
Release



Amazon Elastic Container Service



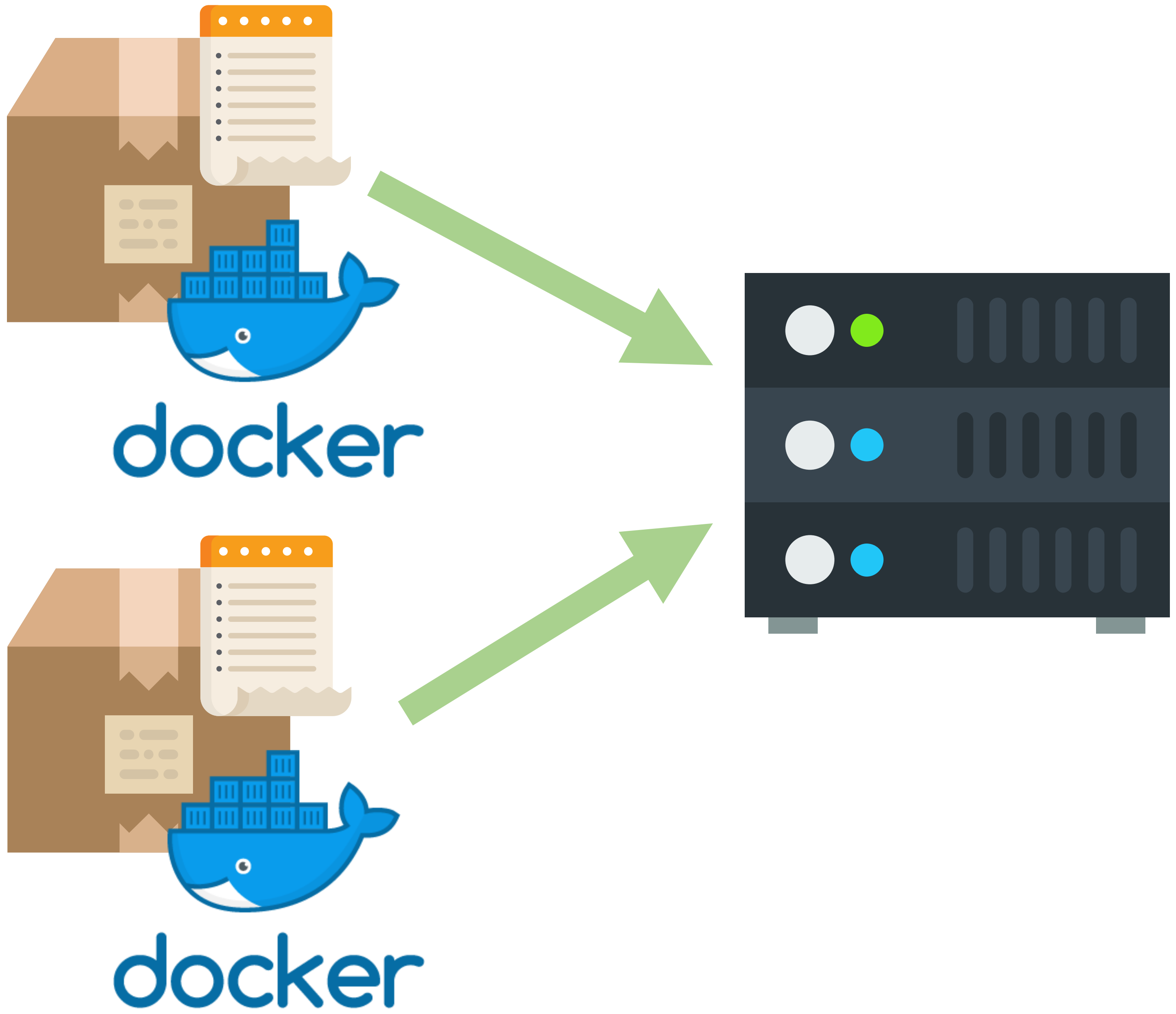
```yaml
# The task definition that describes how to run the docker container
TaskDefinition:
  Type: AWS::ECS::TaskDefinition
  Properties:
    Family: !Ref 'ServiceName'
    TaskRoleArn: !Ref TaskRole
    ContainerDefinitions:
      - Name: !Ref 'ServiceName'
        Cpu: 512
        Memory: 256
        Image: !Ref 'ImageUrl'
        Ulimits:
          - Name: nofile
            HardLimit: 65535
            SoftLimit: 65535
        PortMappings:
          - ContainerPort: 3000
            HostPort: 0
        Environment:
          - Name: 'UV_THREAD_POOL'
            Value: '15'
          - Name: 'NODE_ENV'
            Value: 'production'
```



Run

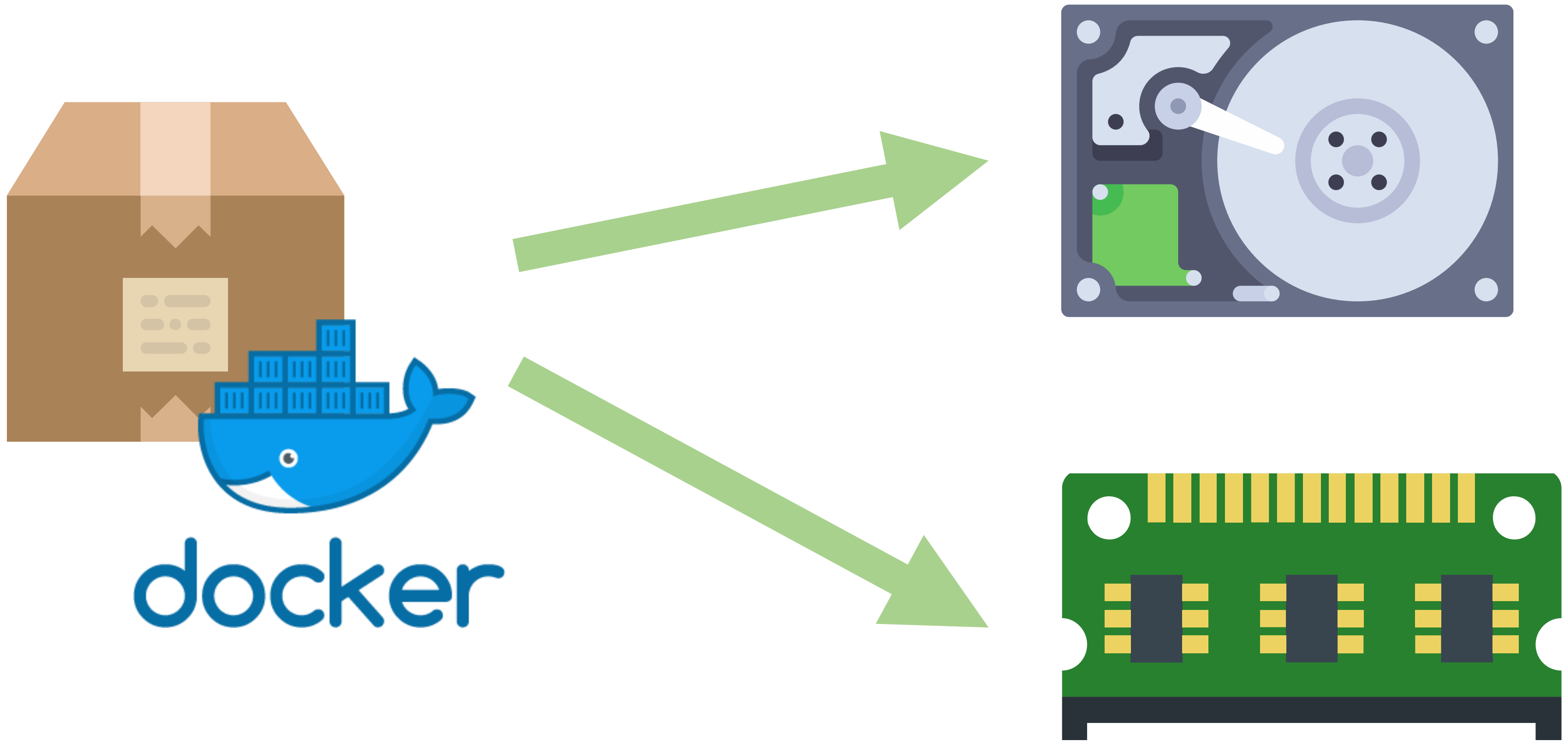
Task Definition
Release v1.0.0

Task Definition
Release v1.0.1

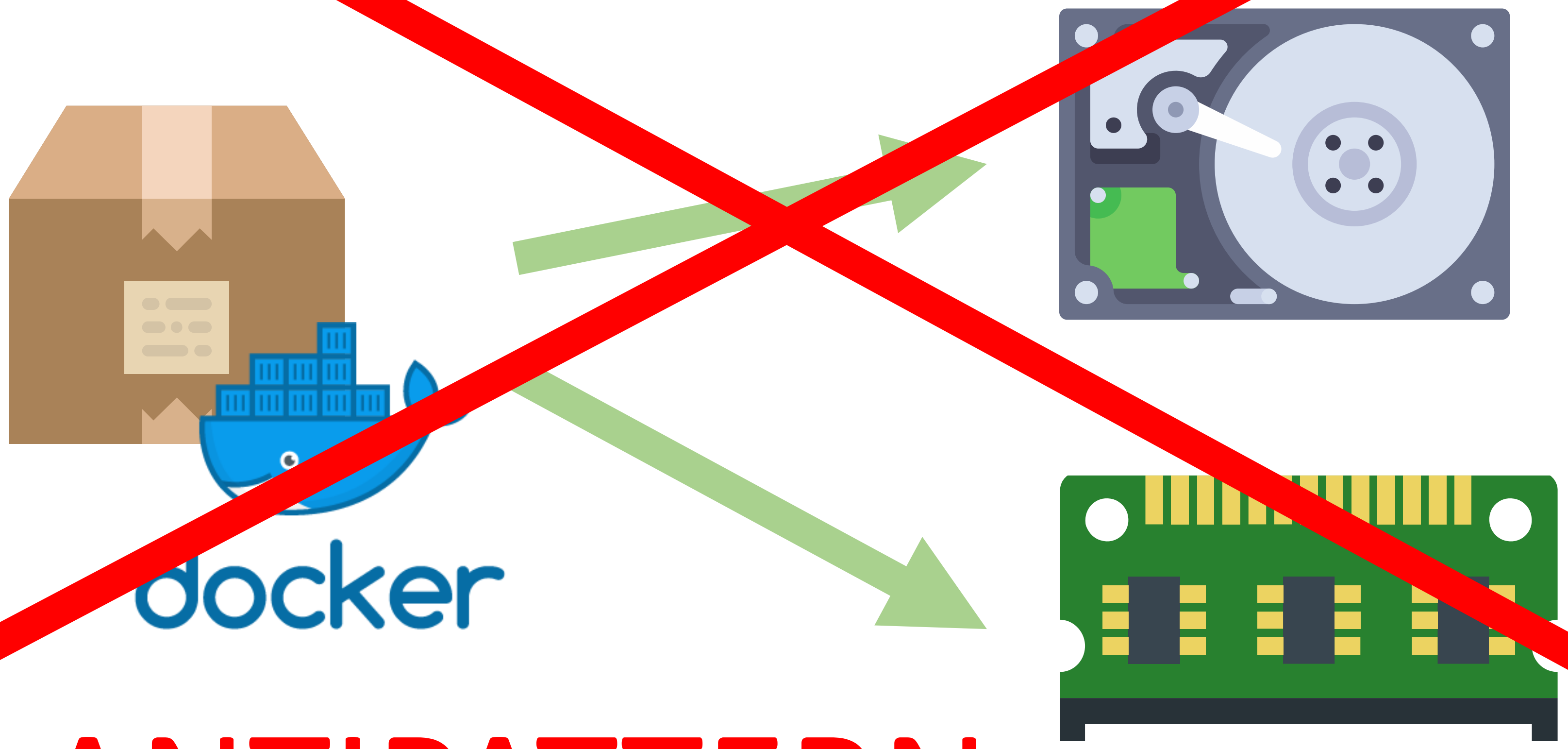


12 Factor Application: Stateless Processes

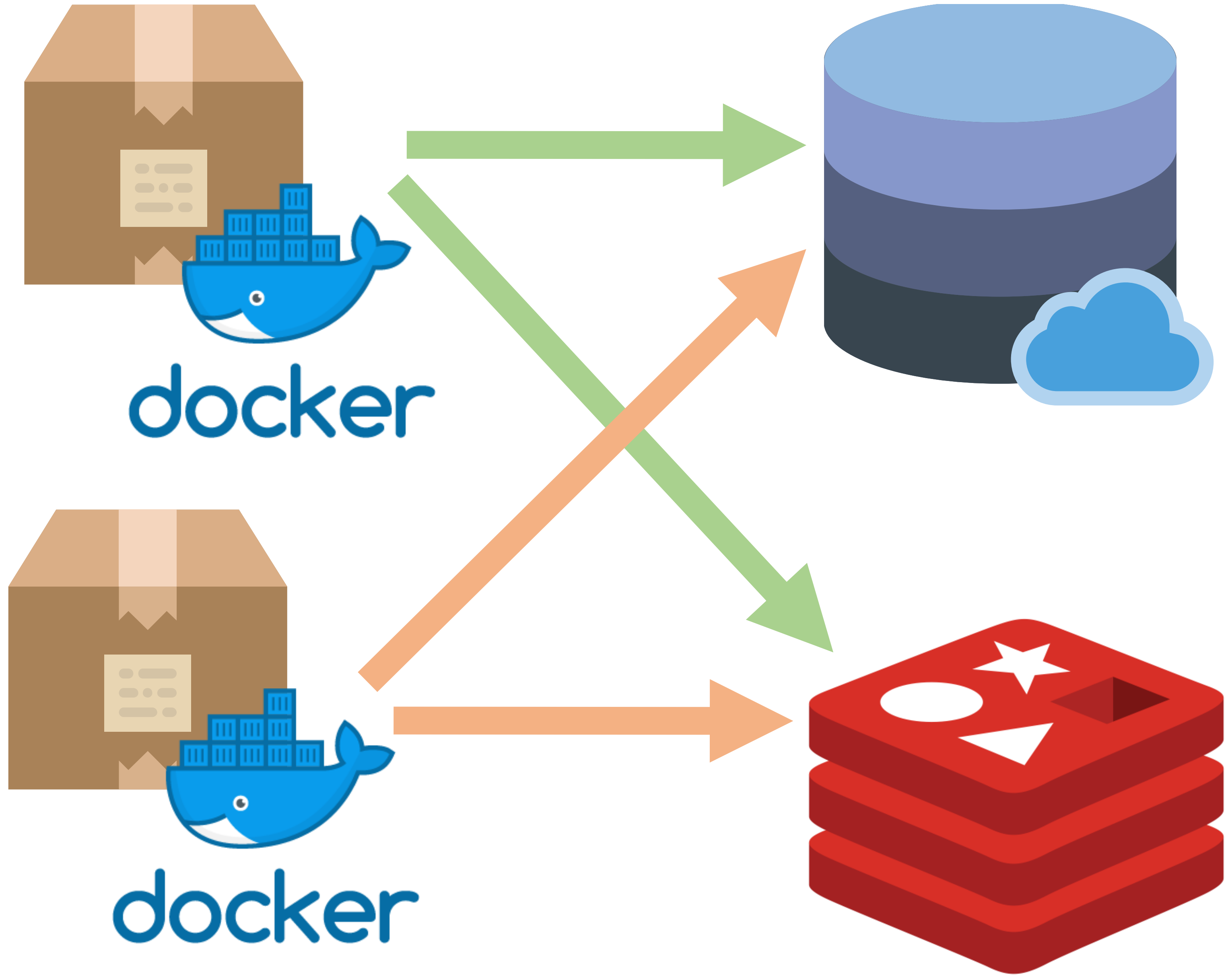
Stateful container stores state in local disk or local memory. Workload ends up tied to a specific host that has state data.



Stateful container stores state in local disk or local memory.
Workload ends up tied to a specific host that has state data.



ANTIPATTERN



Database

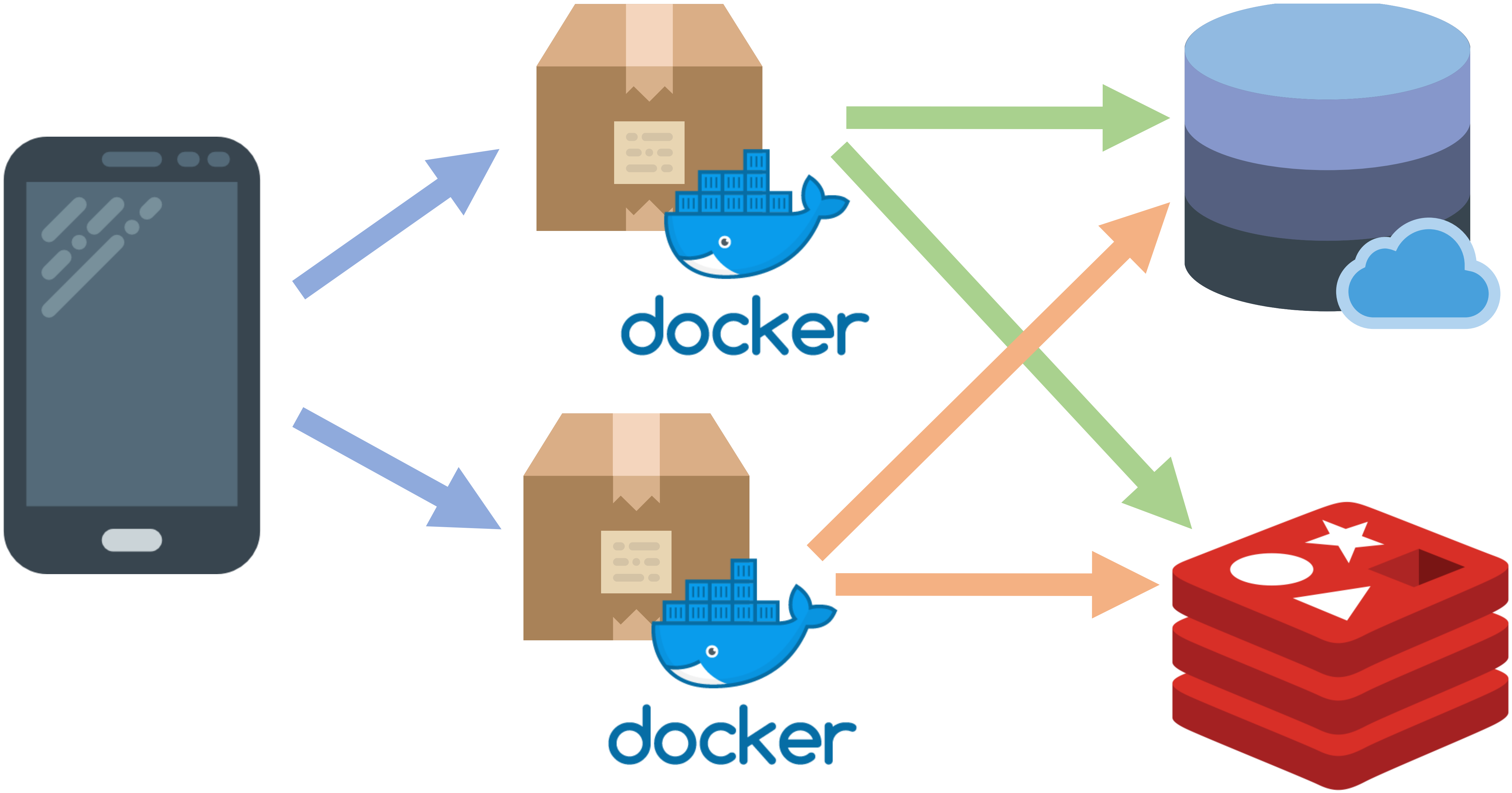
Durable store of truth.

MySQL, PostgreSQL, MongoDB, DynamoDB

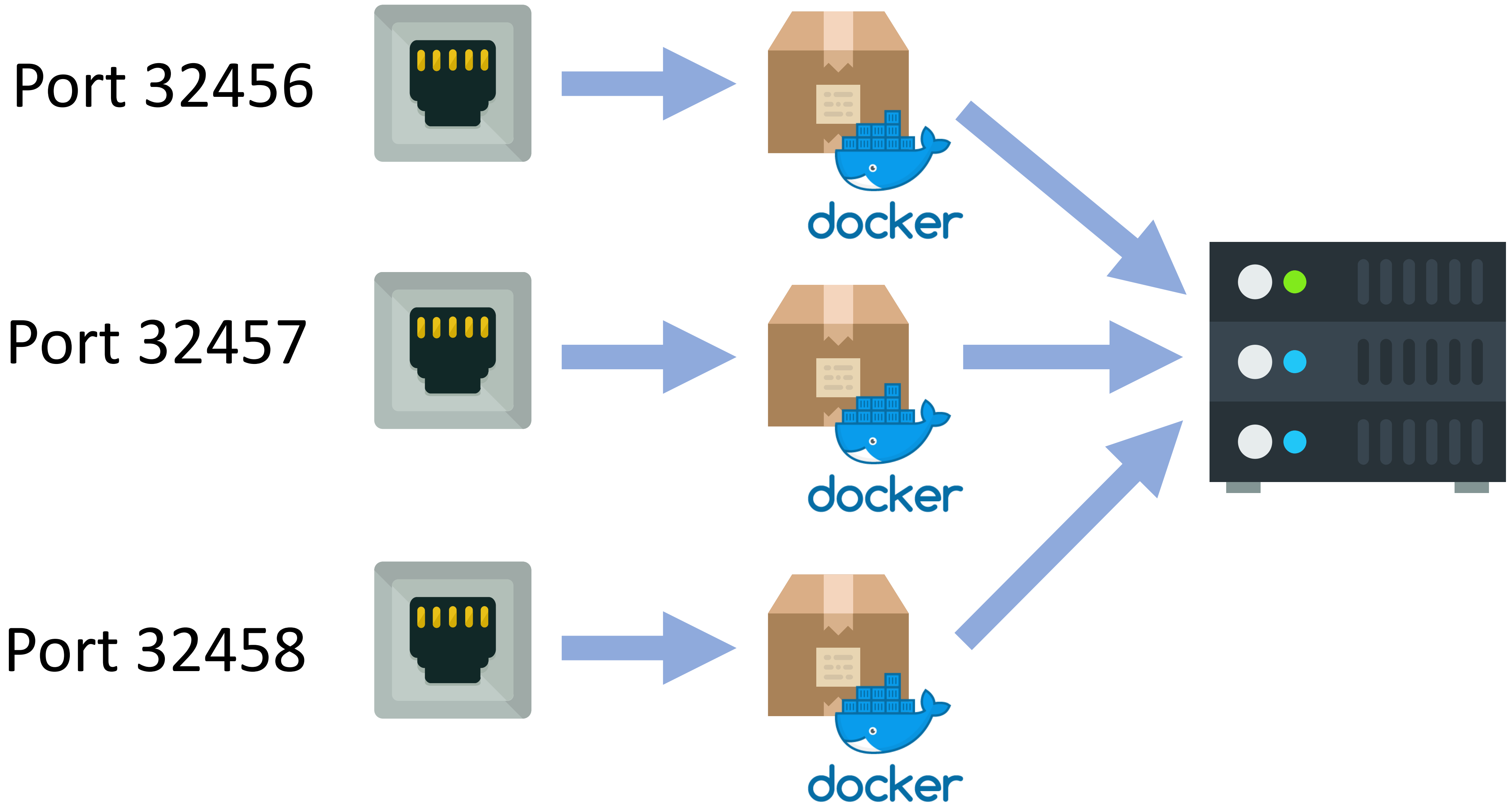
Cache

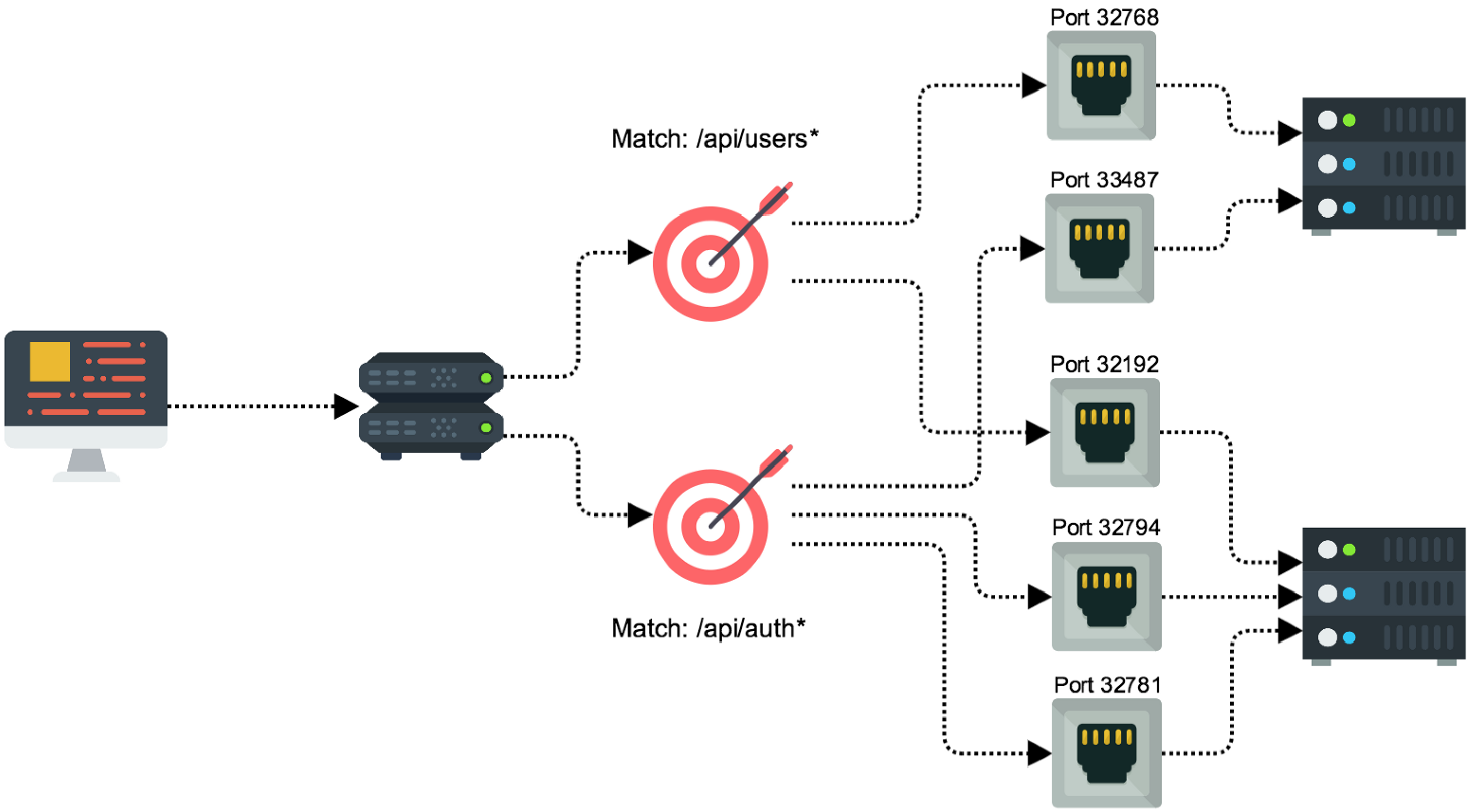
Fast, temporary state store.

redis, memcached


docker
docker

12 Factor Application: Port Binding


Port 32456
Port 32457
Port 32458
docker
docker
docker

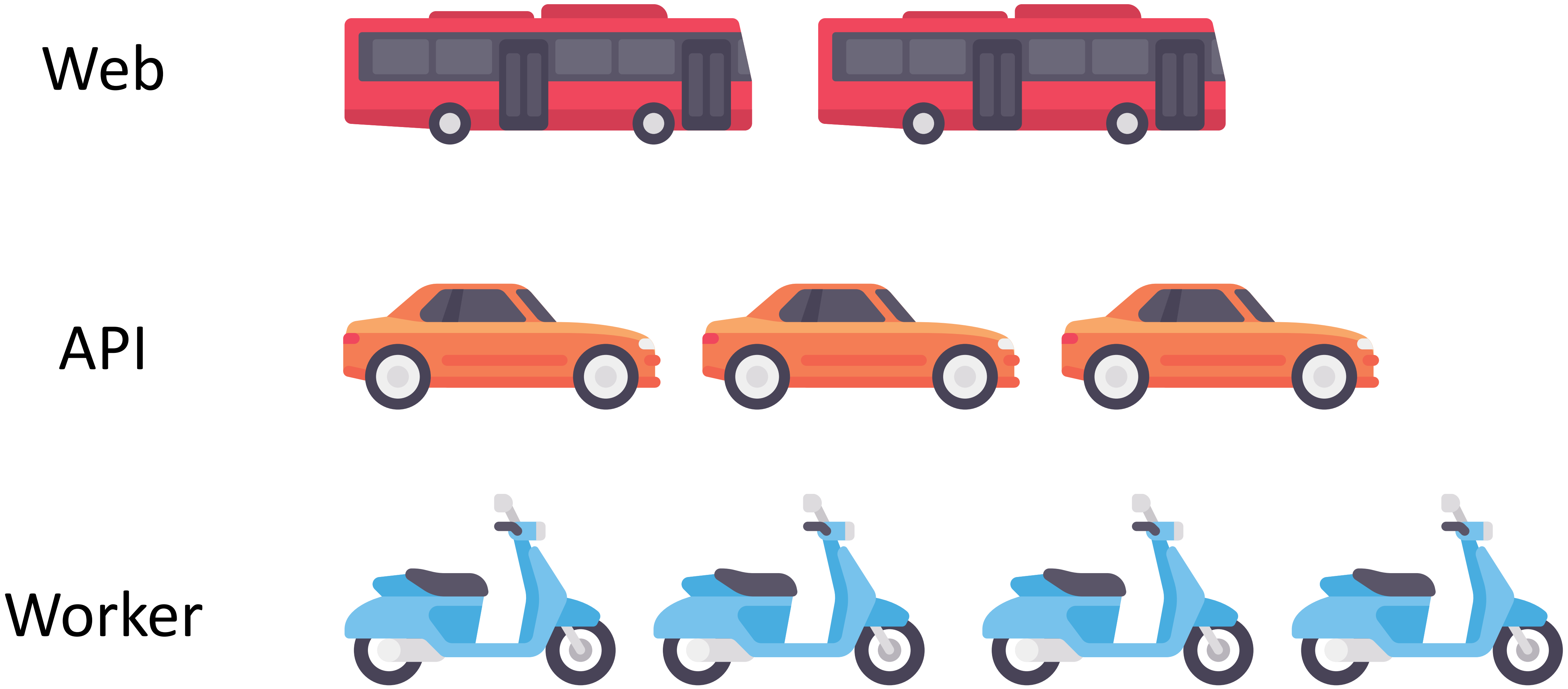

Match: /api/users*
Match: /api/auth*
Port 32768
Port 33487
Port 32192
Port 32794
Port 32781

12 Factor Application: Concurrency

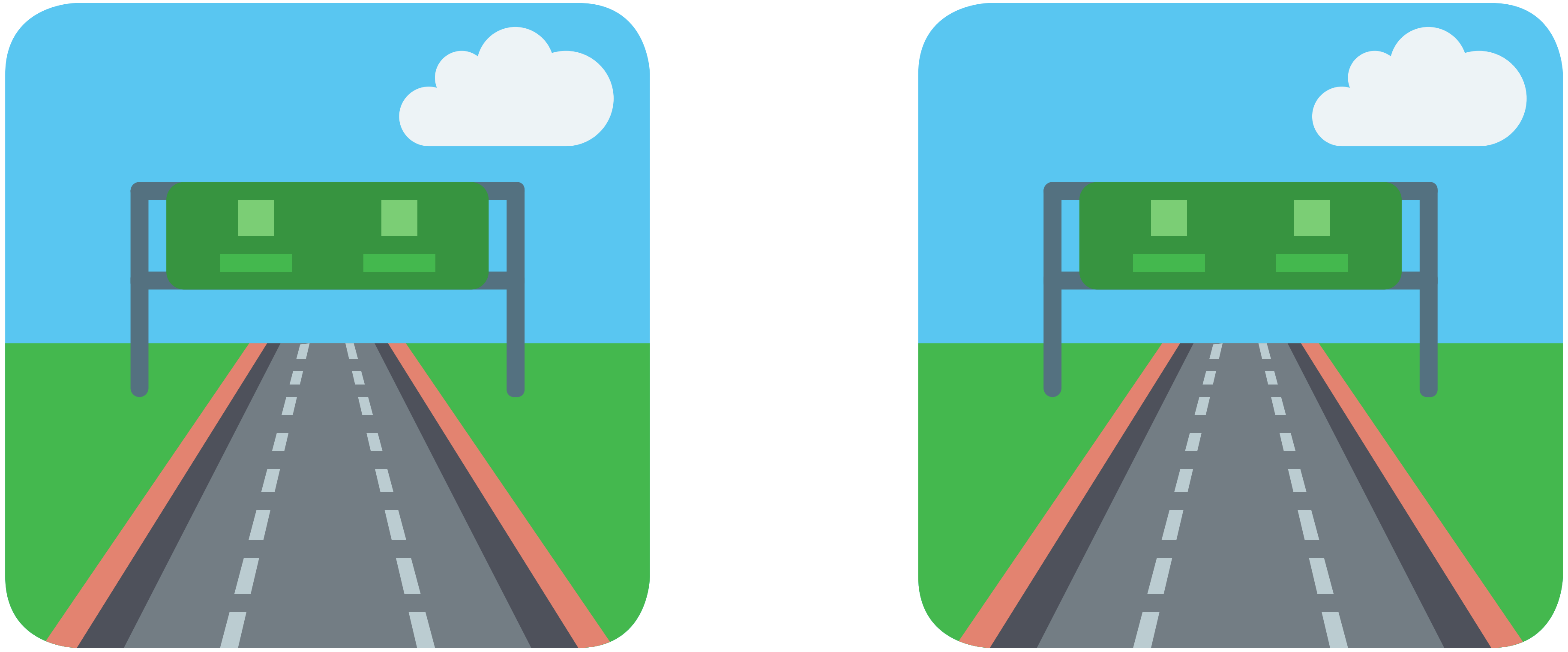
API

Web

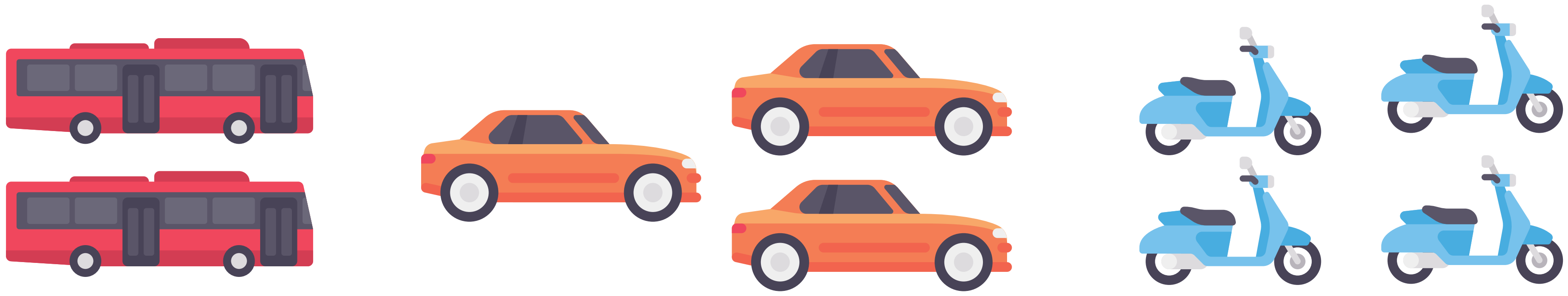
Worker


Web
API
Worker

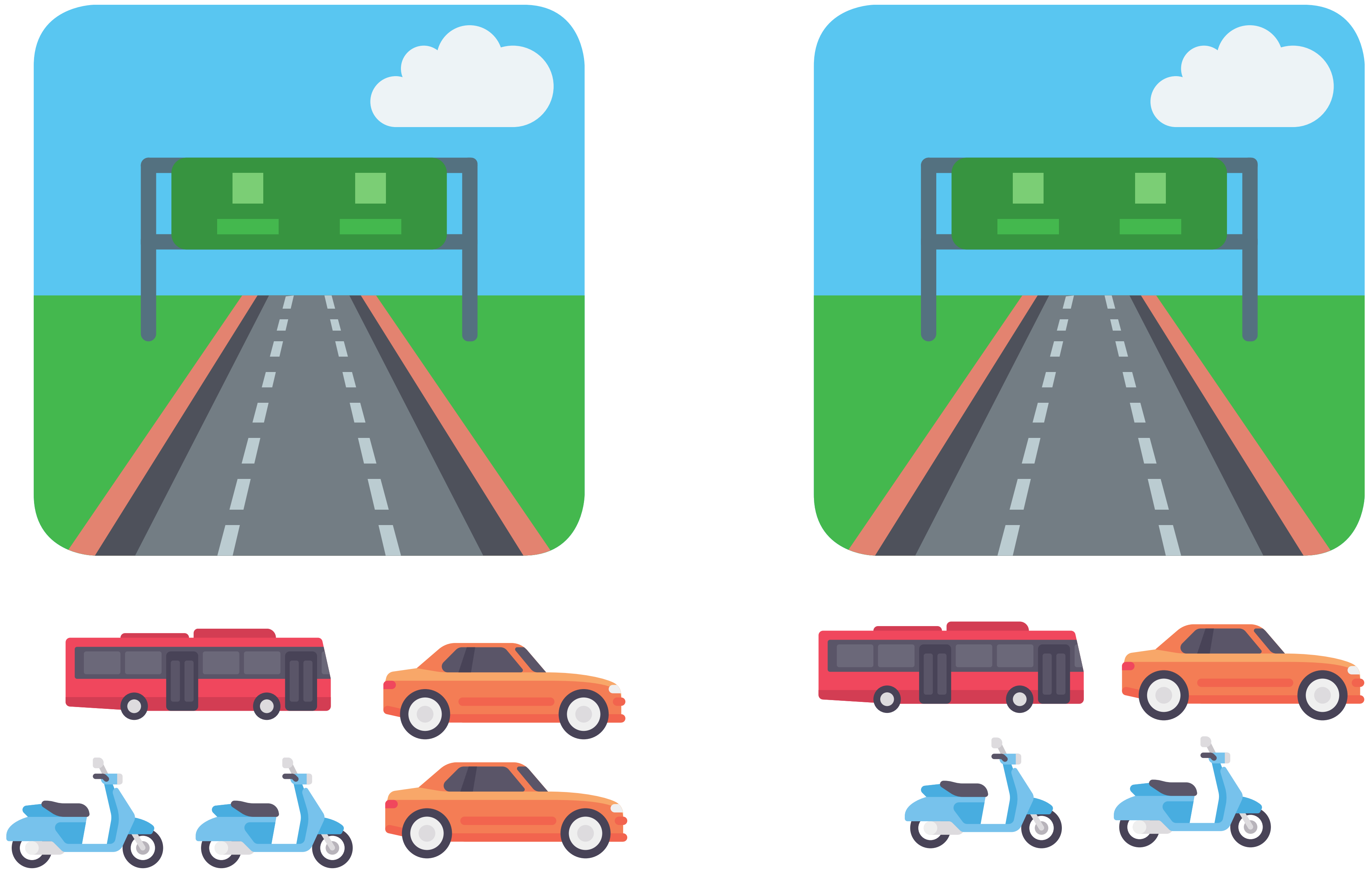
Hosts



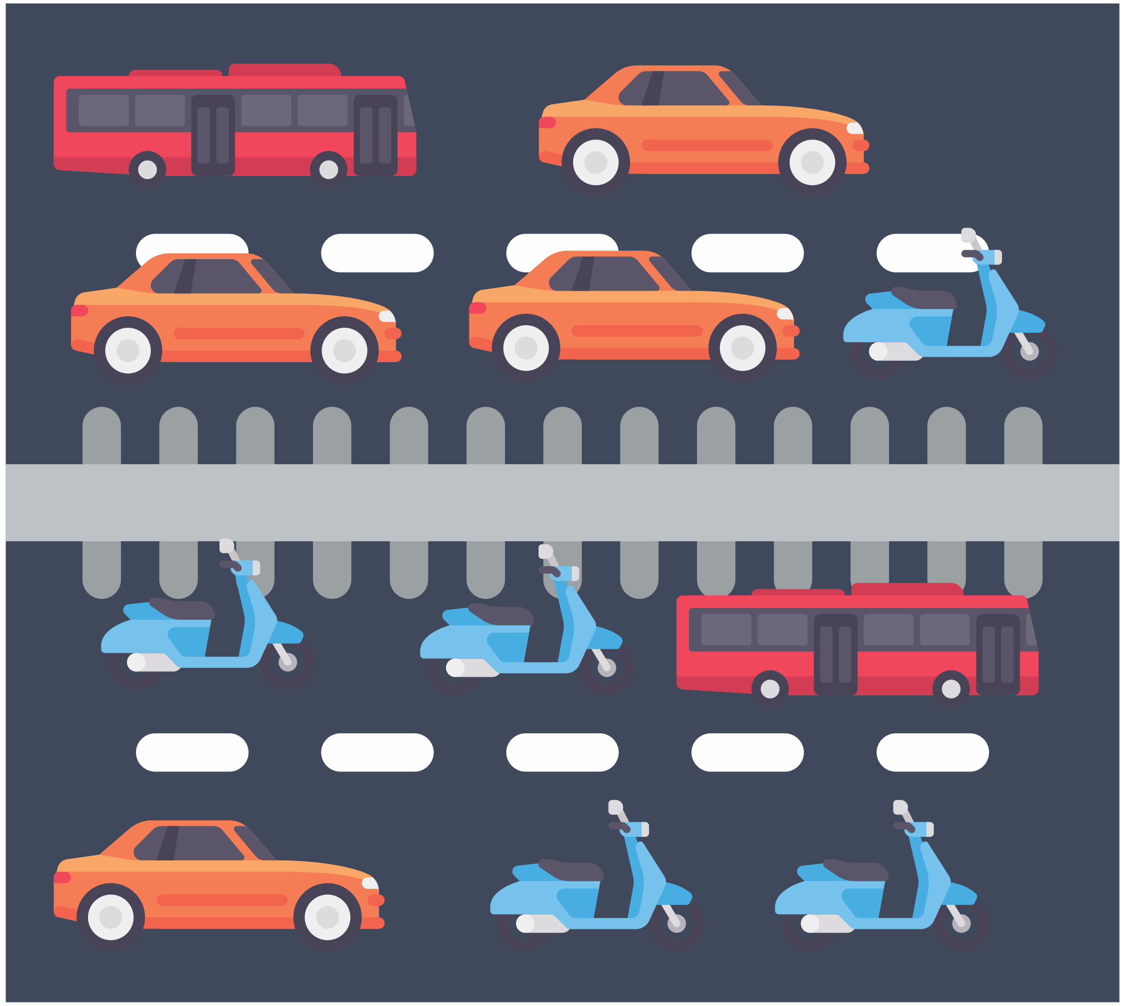
Processes

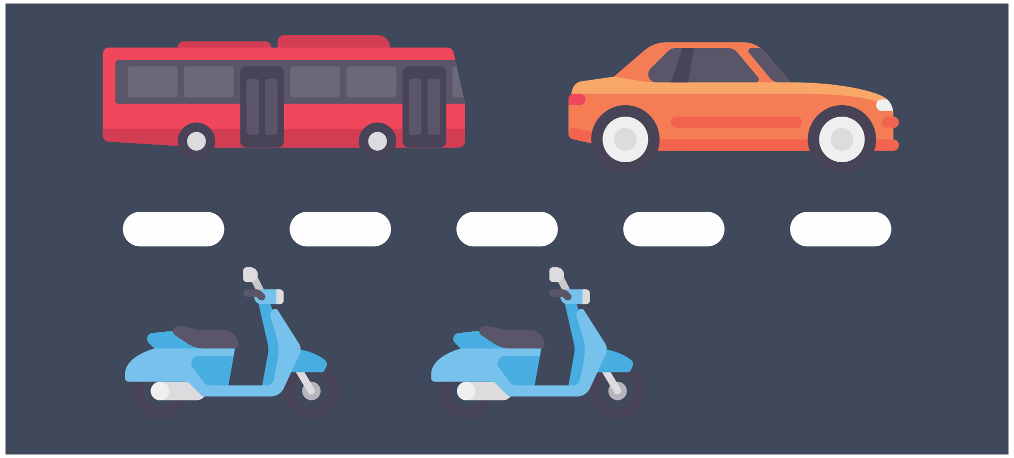


Hosts

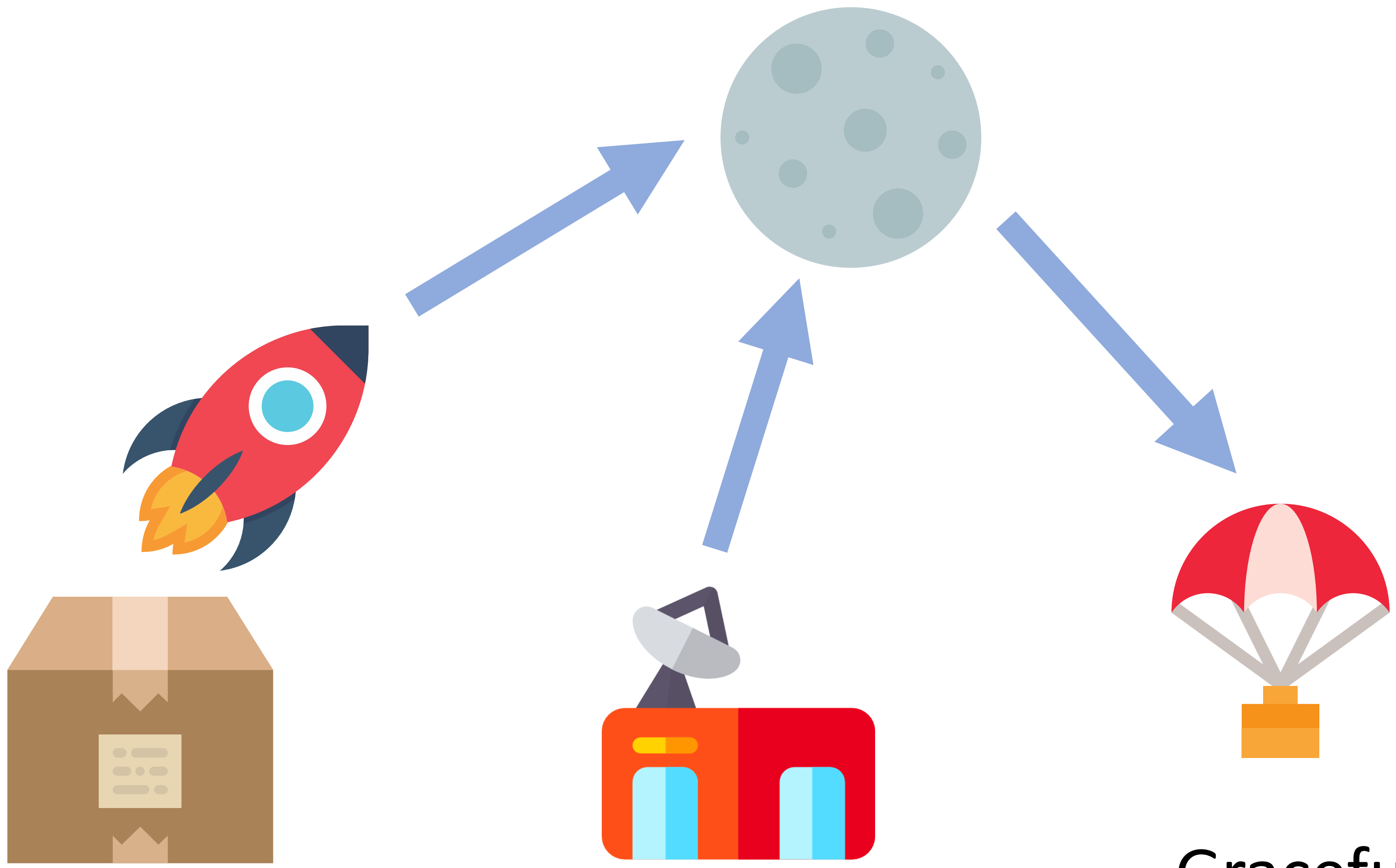


Processes


Large Host = More Concurrent Processes


Small Host = Fewer Concurrent Processes

12 Factor Application: Disposability


Fast Launch
Responsive
Graceful Shutdown

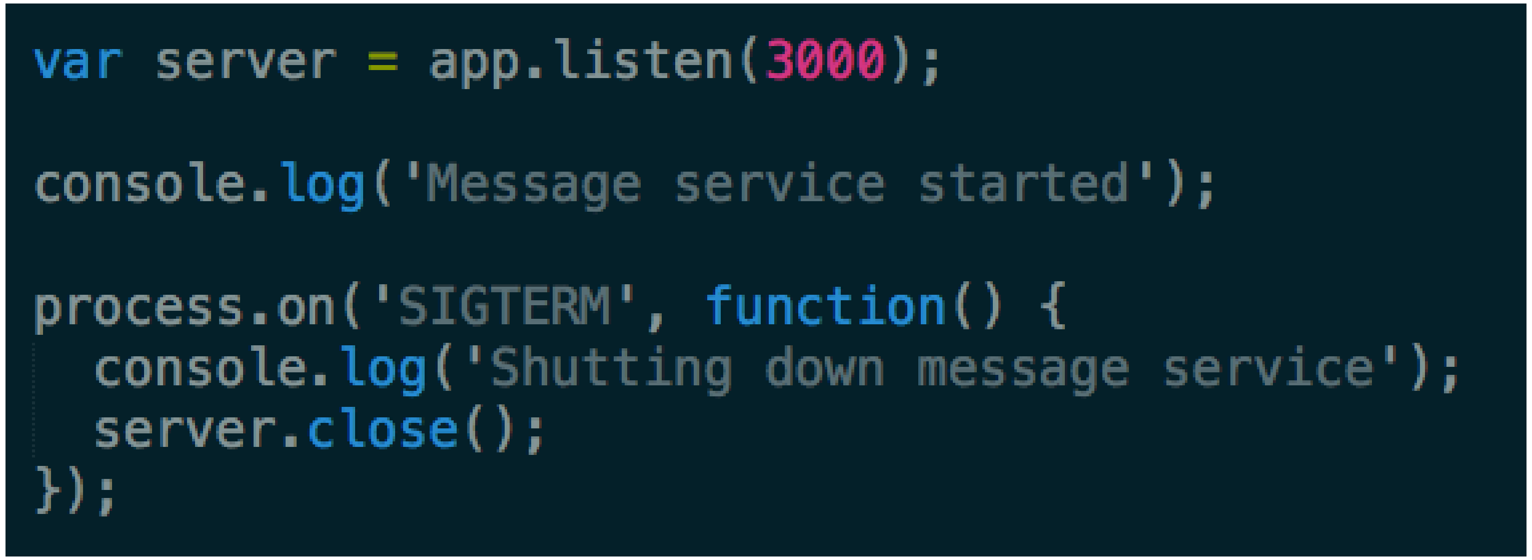
Fast Launch

Minimize the startup time of processes:

- Scale up faster in response to spikes
- Ability to move processes to another host as needed
- Replace crashed processes faster

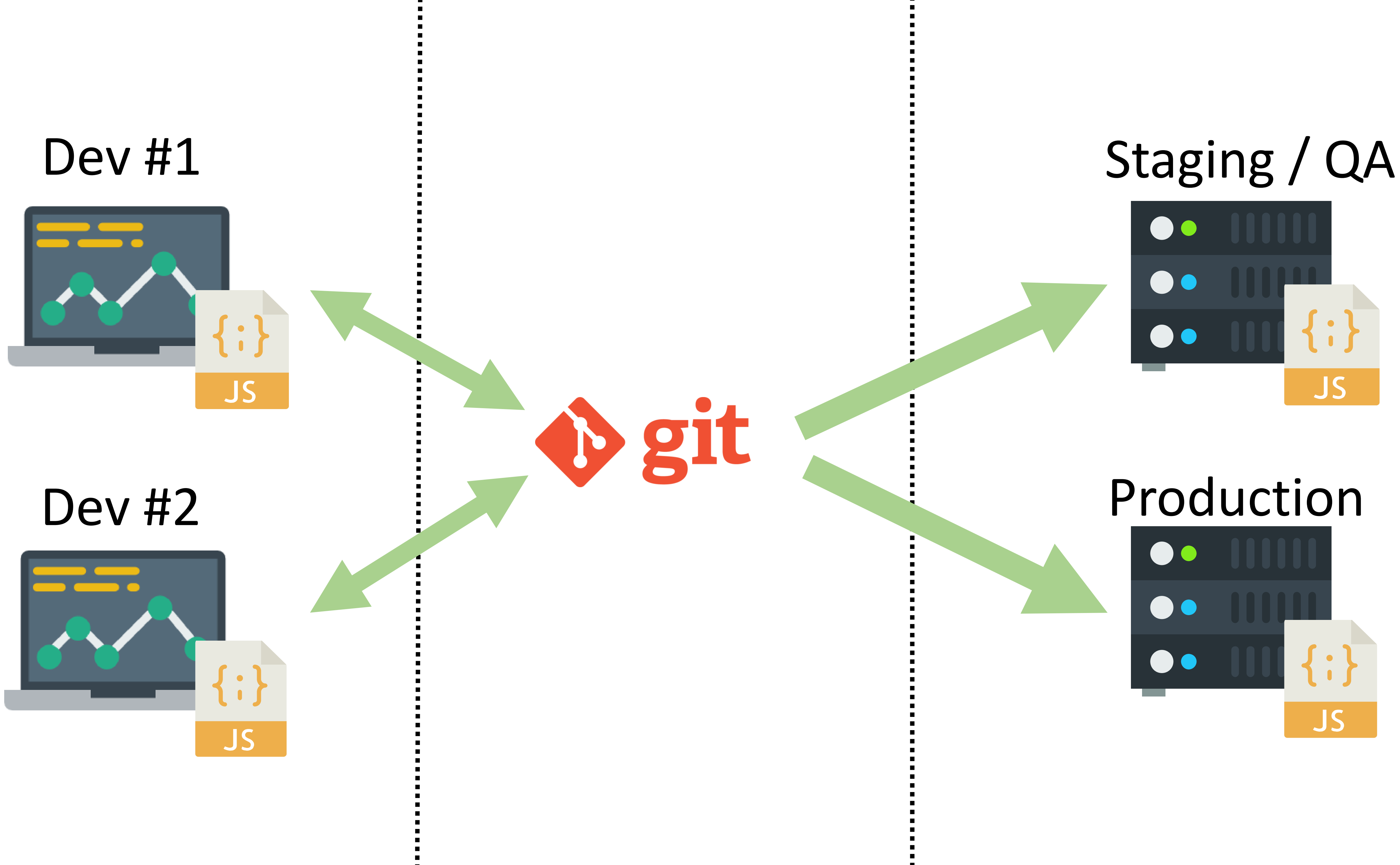
Responsive, Graceful Shutdown

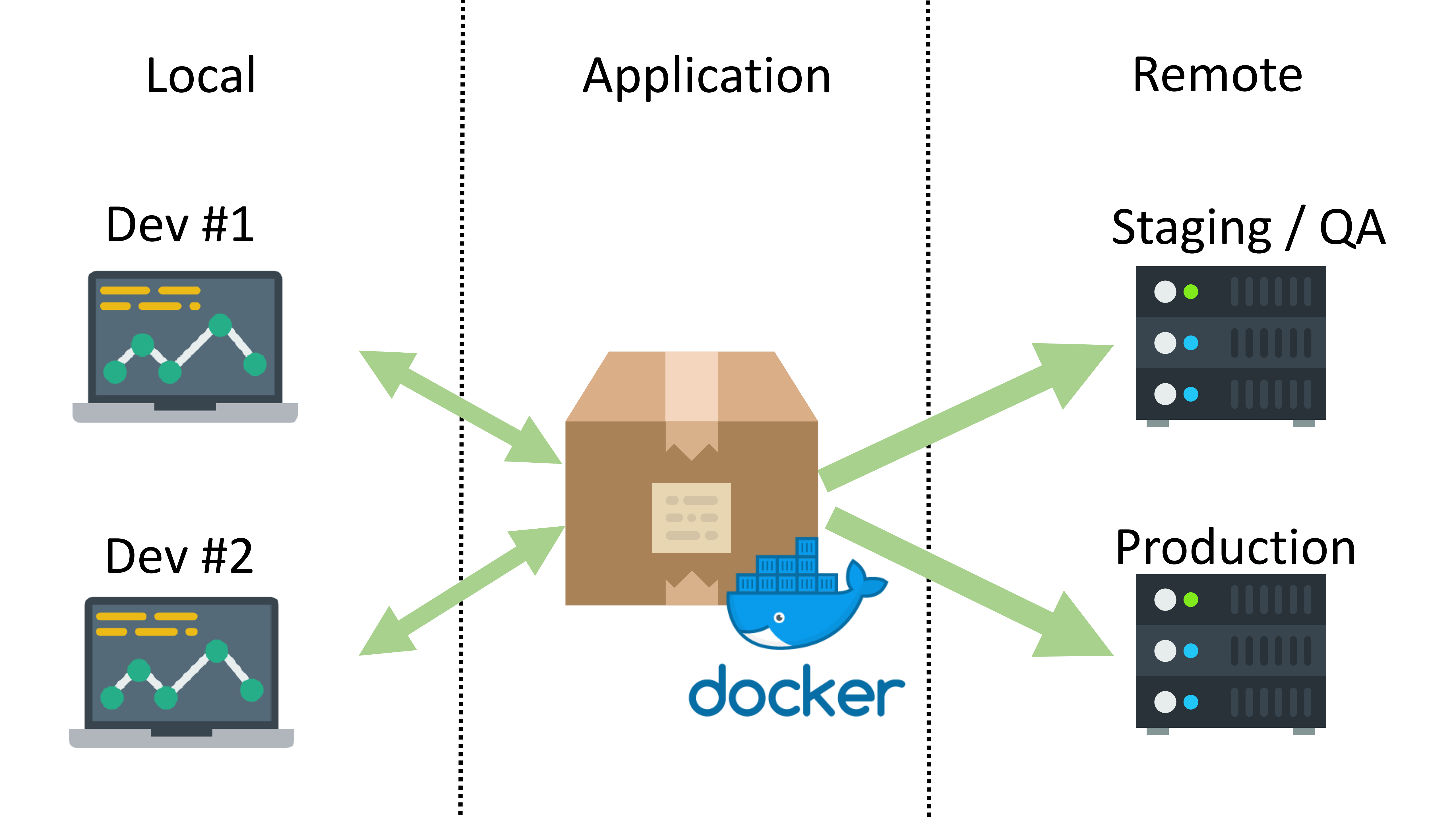
Should respond to SIGTERM by shutting down gracefully



```
var server = app.listen(3000);

console.log('Message service started');

process.on('SIGTERM', function() {
  console.log('Shutting down message service');
  server.close();
});
```

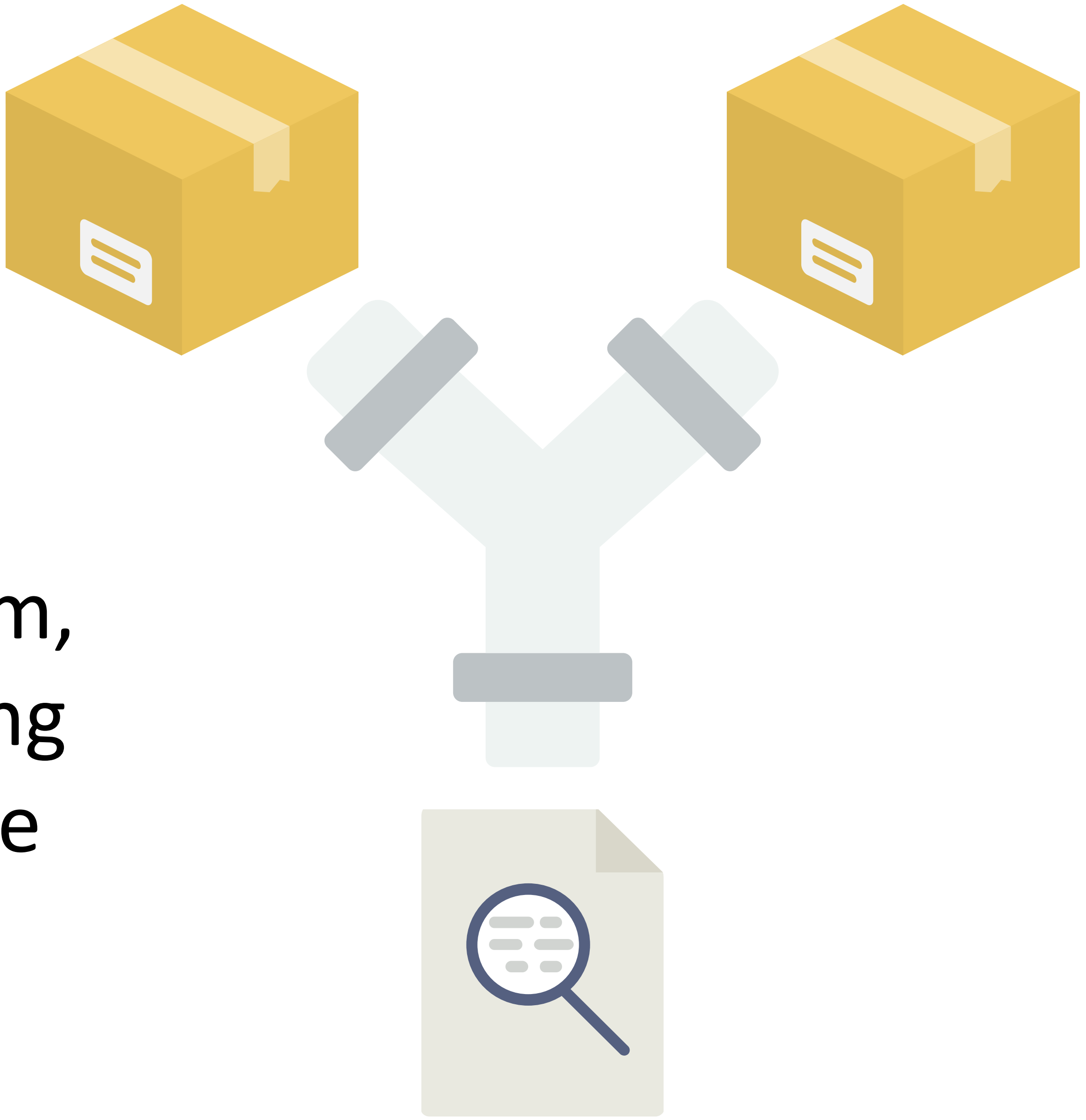
12 Factor Application: Dev/Prod Parity

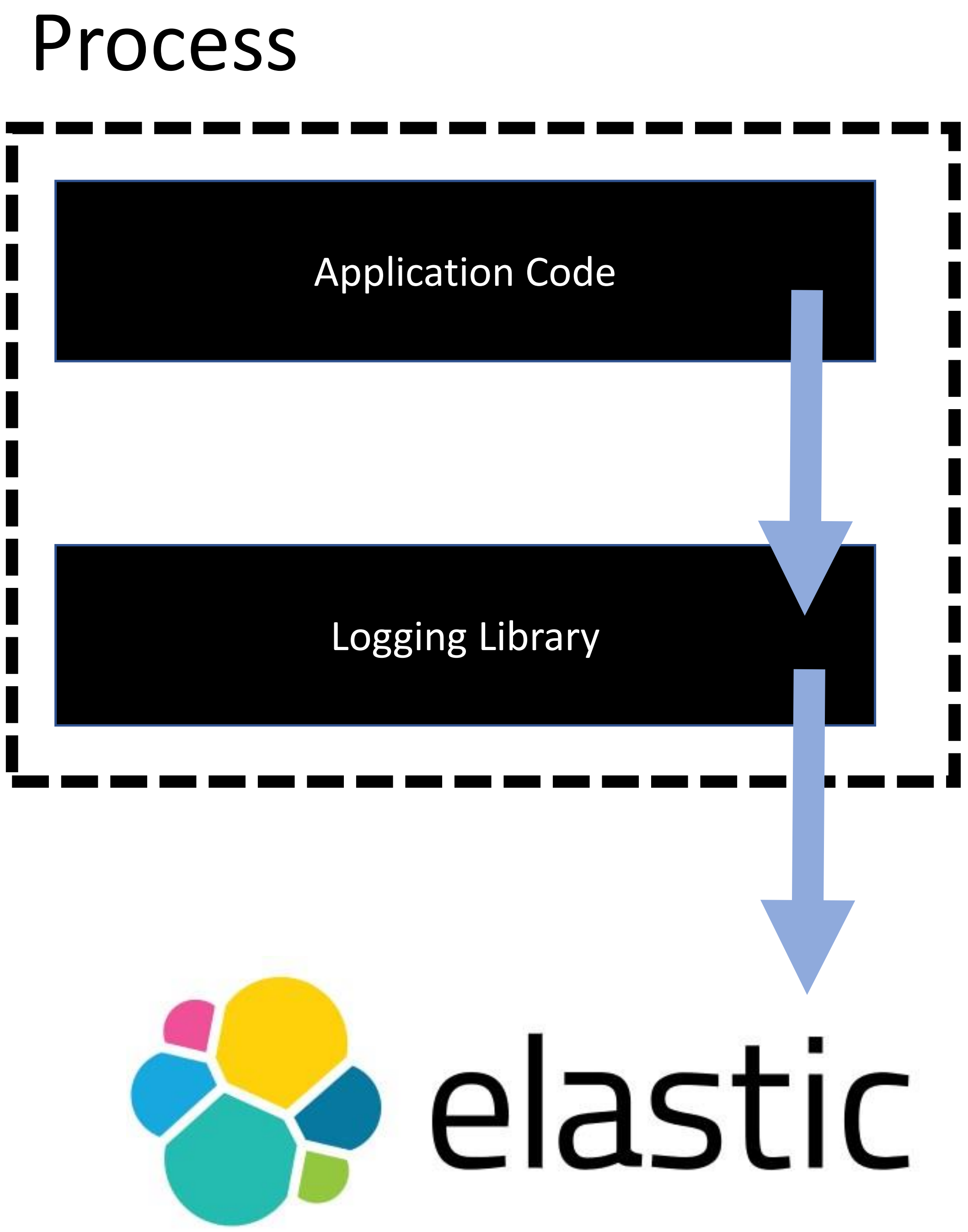

Dev #1
JS
Dev #2
JS
git
Staging / QA
JS
Production
JS

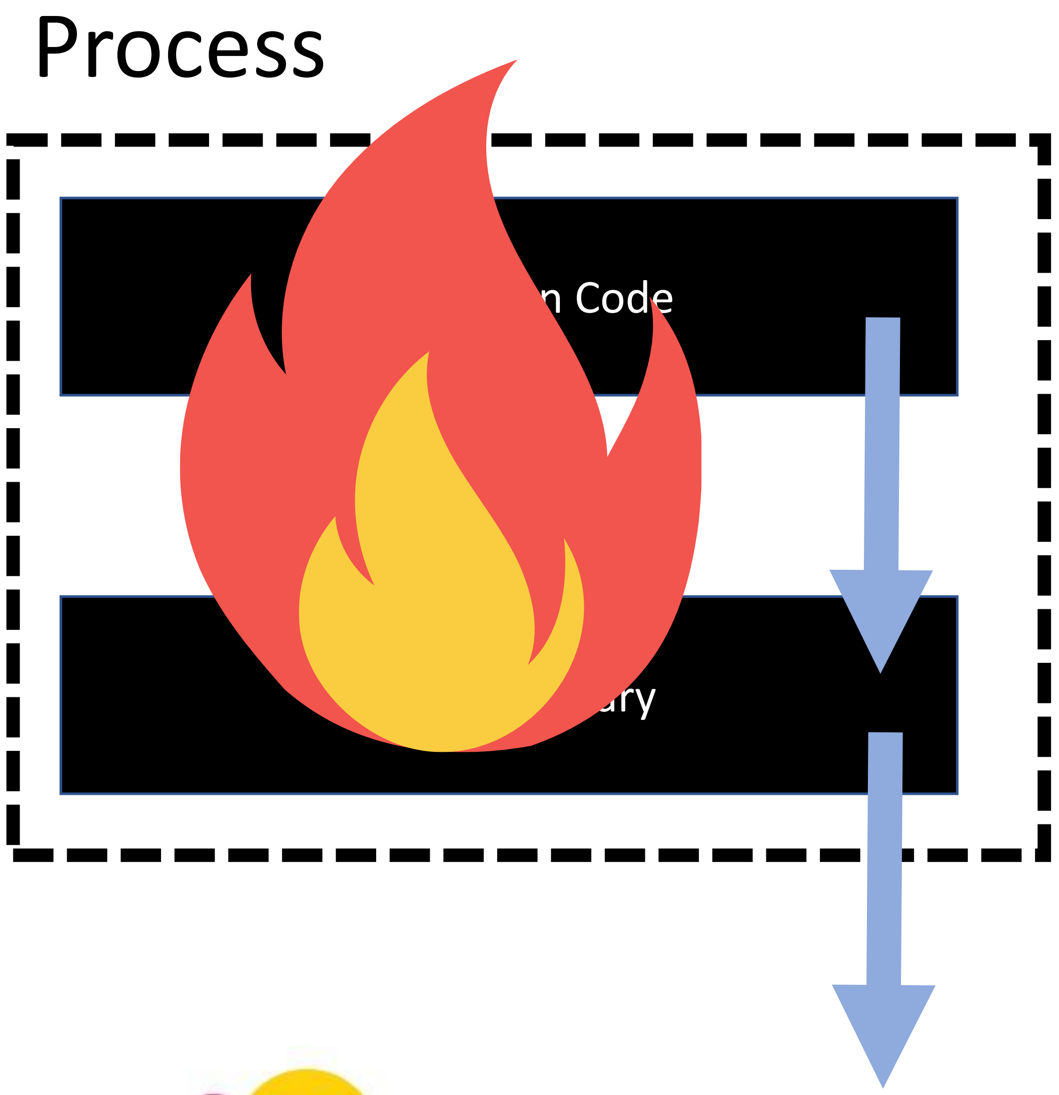

Local
Application
Remote
Dev #1
Dev #2
Staging / QA
Production
docker

12 Factor Application: Logs

Treat logs as an event stream, and keep the logic for routing and processing logs separate from the application itself.



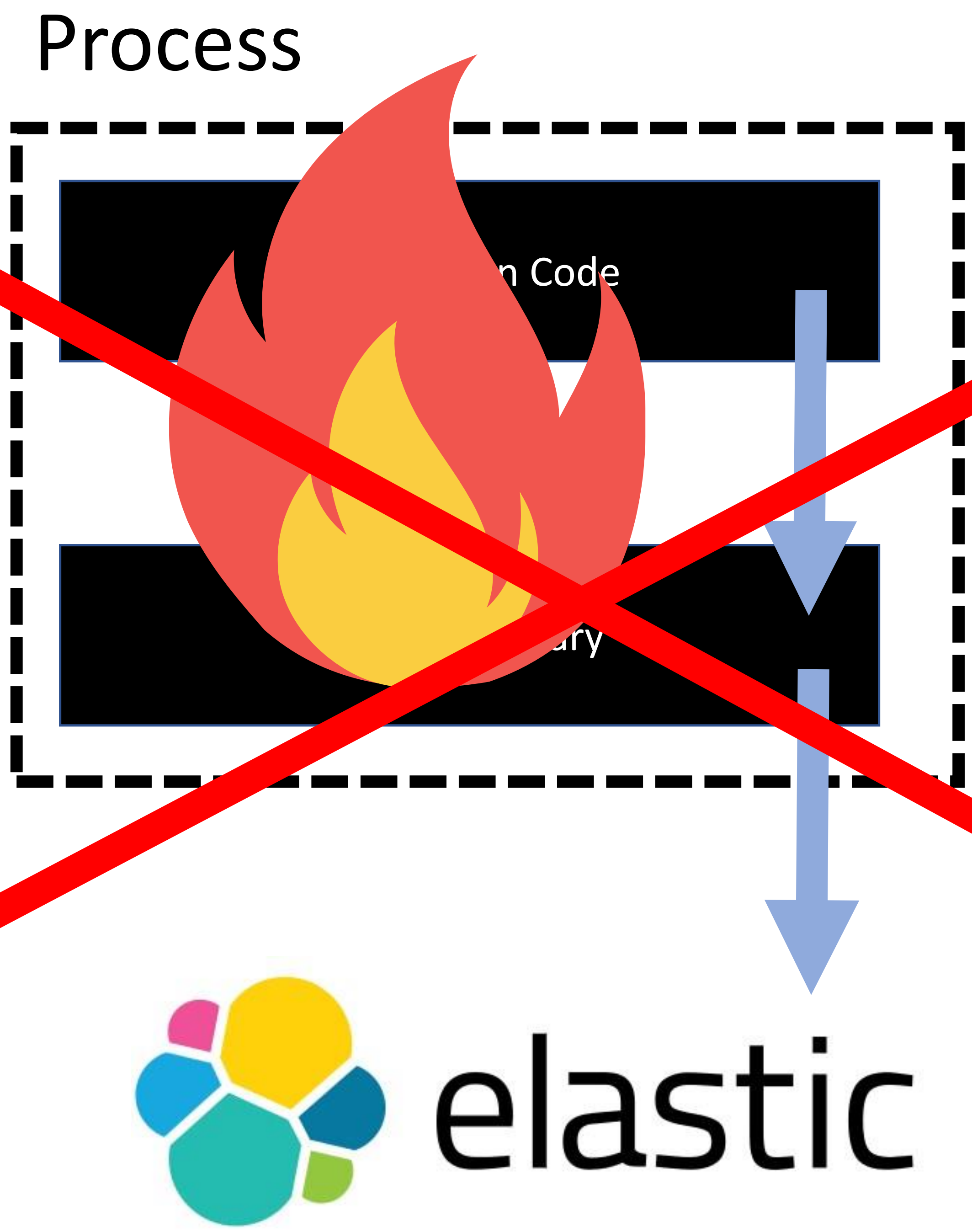

Process
Application Code
Logging Library
elastic



Some logs get lost if they haven’t fully flushed

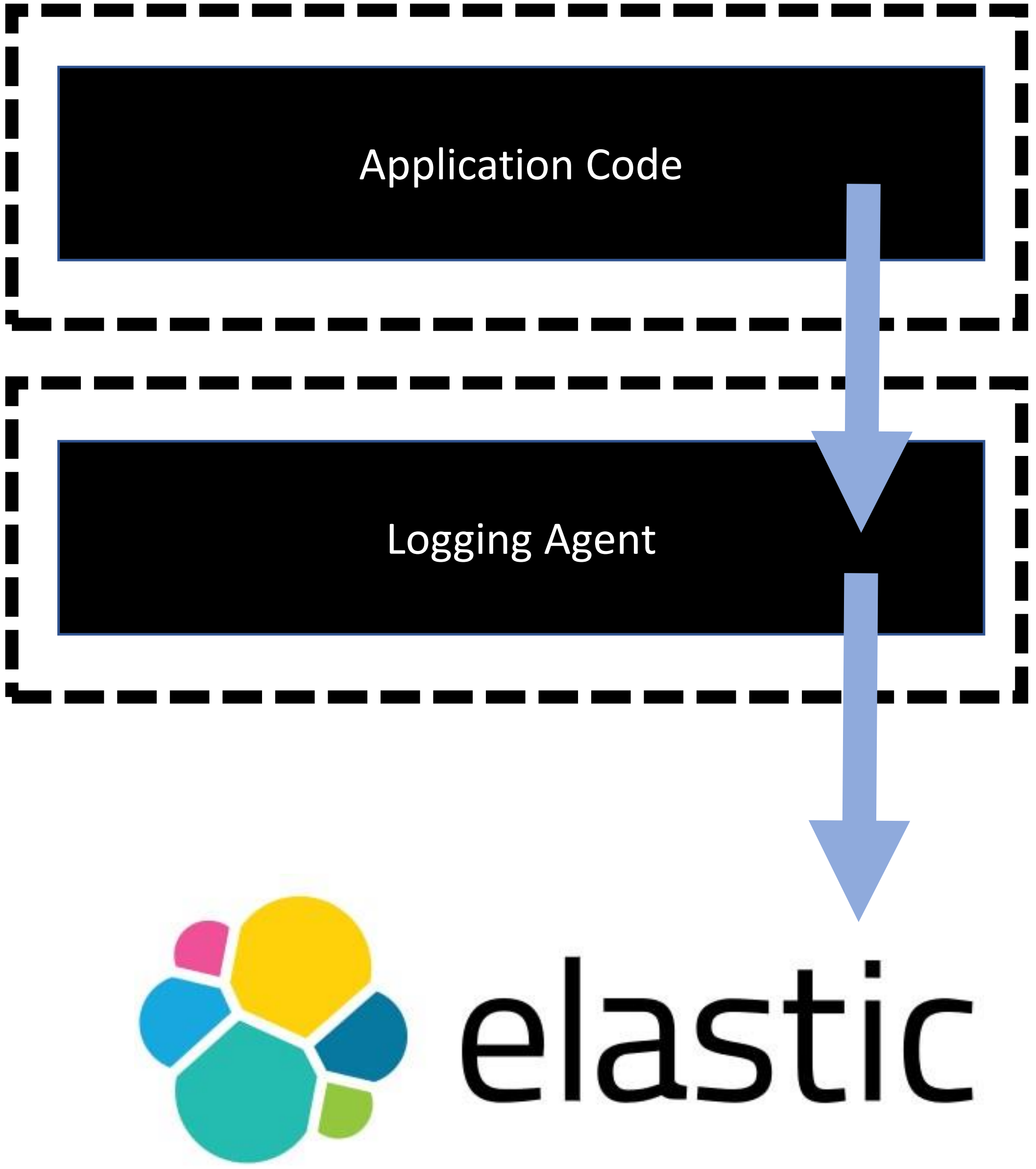
elastic

ANTIPATTERN

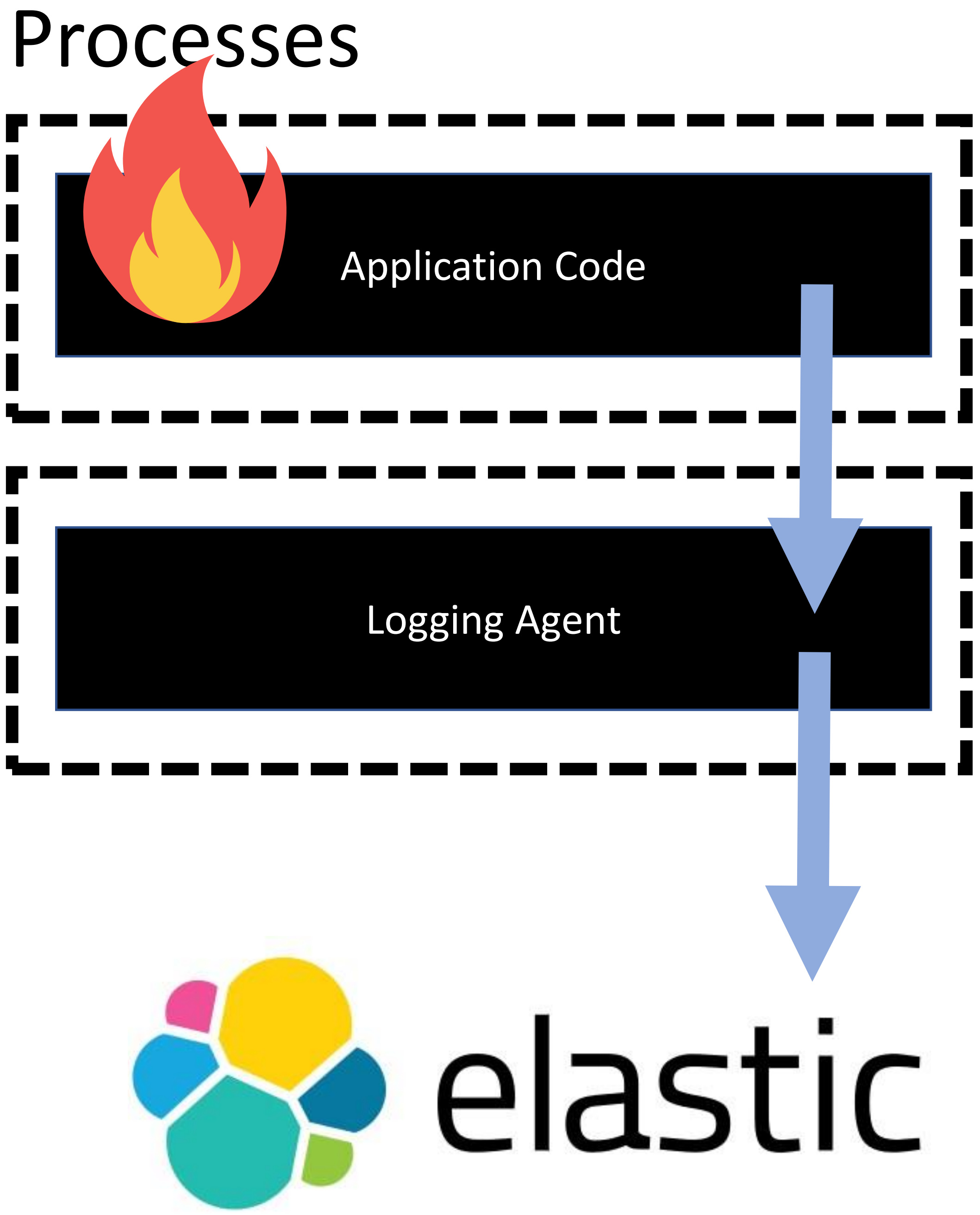


Some logs get lost if they haven’t fully flushed

Processes

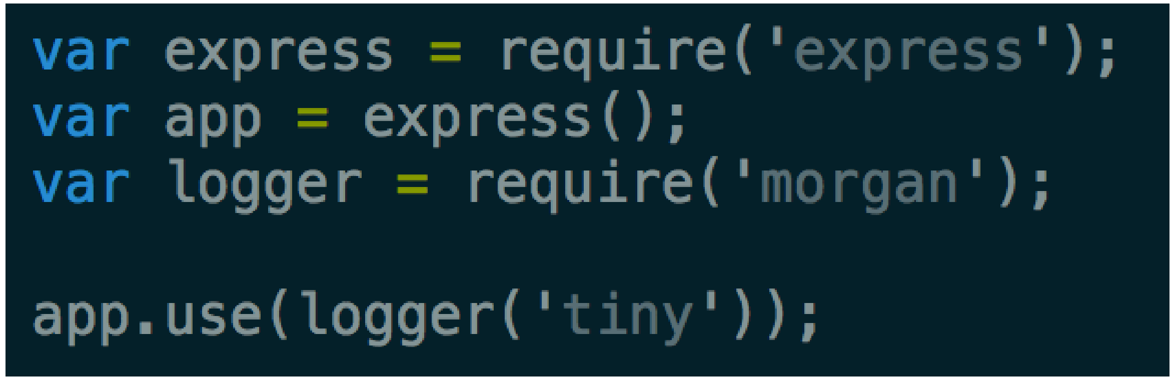


Logs go to an agent which handles exporting them off the host



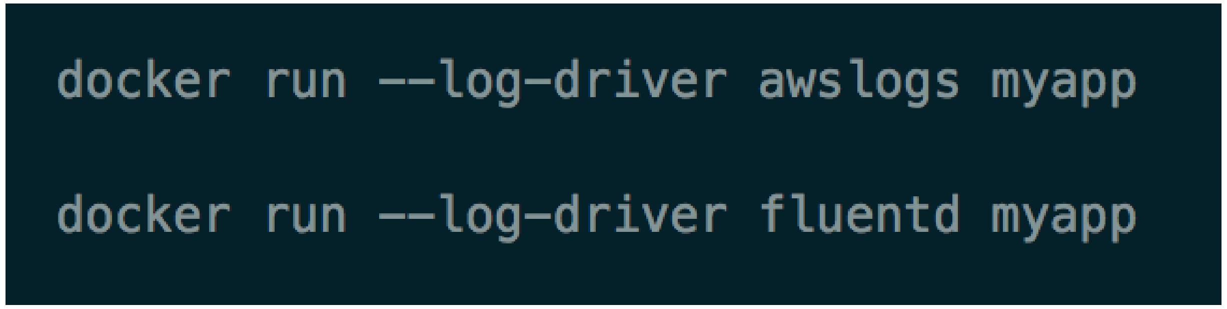
Logs still reach agent, and still make it into ELK stack

Containerized code writes to stdout



```
var express = require('express');
var app = express();
var logger = require('morgan');

app.use(logger('tiny'));
```

Docker connects container’s stdout to a log driver

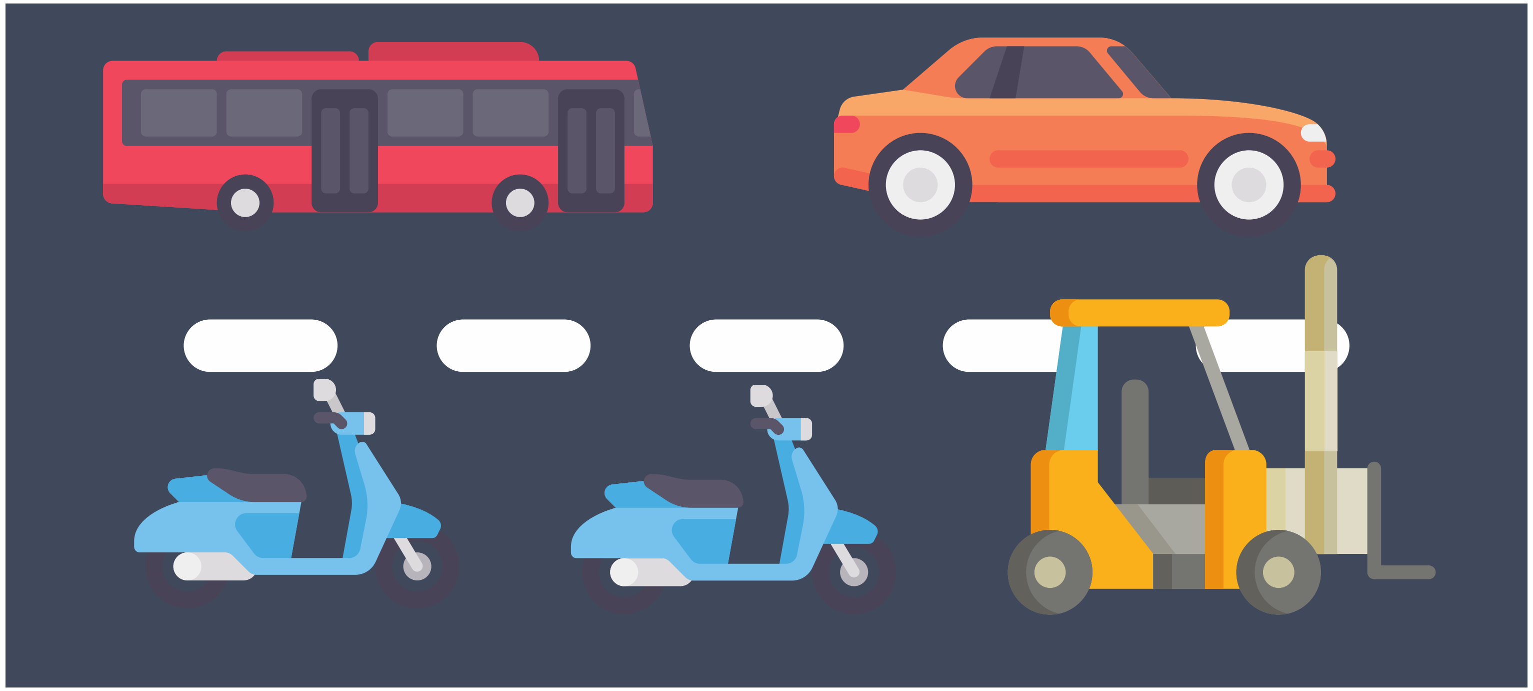


```
docker run --log-driver awslogs myapp

docker run --log-driver fluentd myapp
```

12 Factor Application: Admin Processes

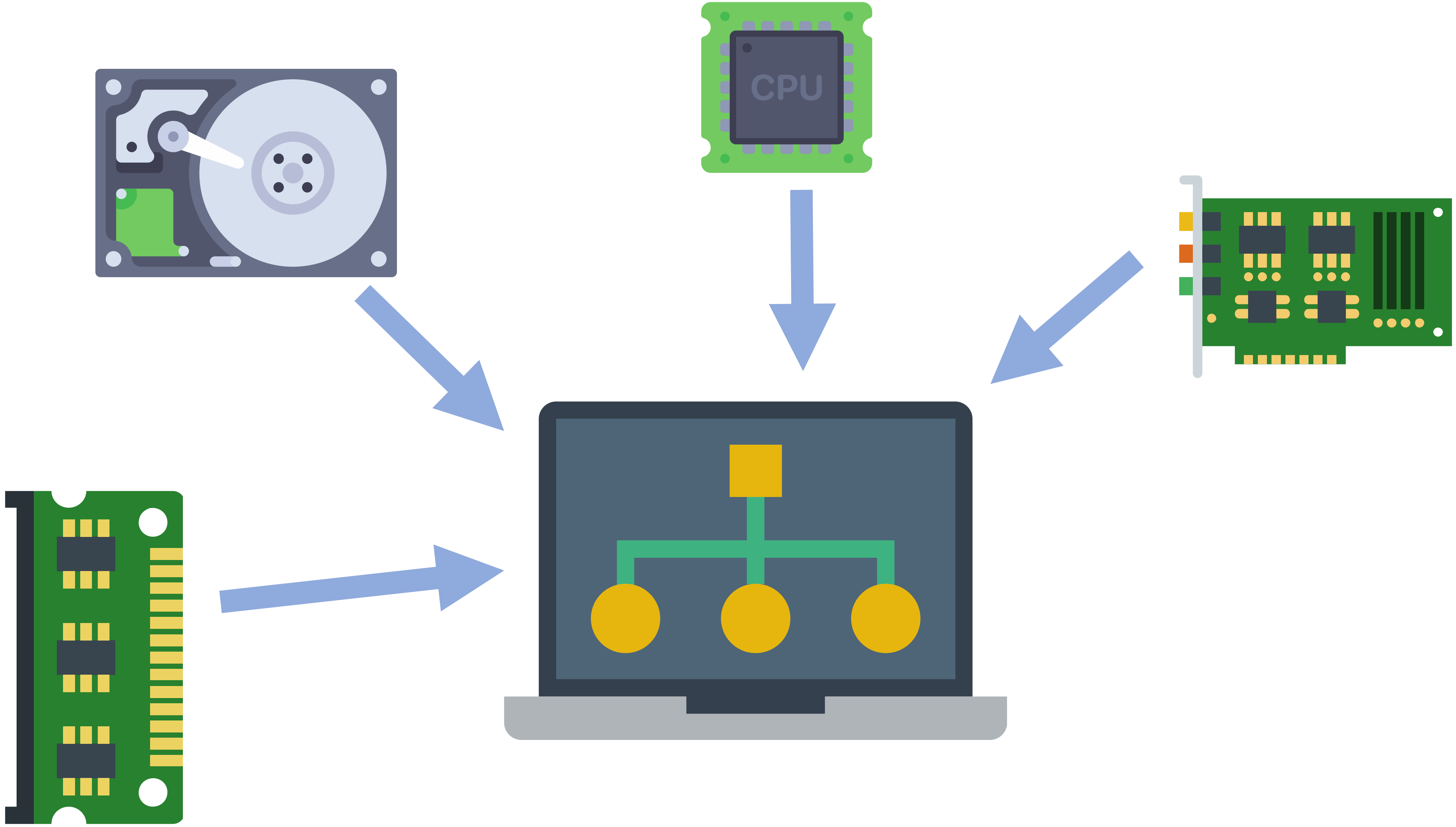
Admin / management processes are inevitable:

- Migrate database
- Repair some broken data
- Once a week move database records older than X to cold storage
- Every day email a report to this person



Run admin processes just like other processes.

Microservices: Componentization


CPU

Each component is a 12 factor application.

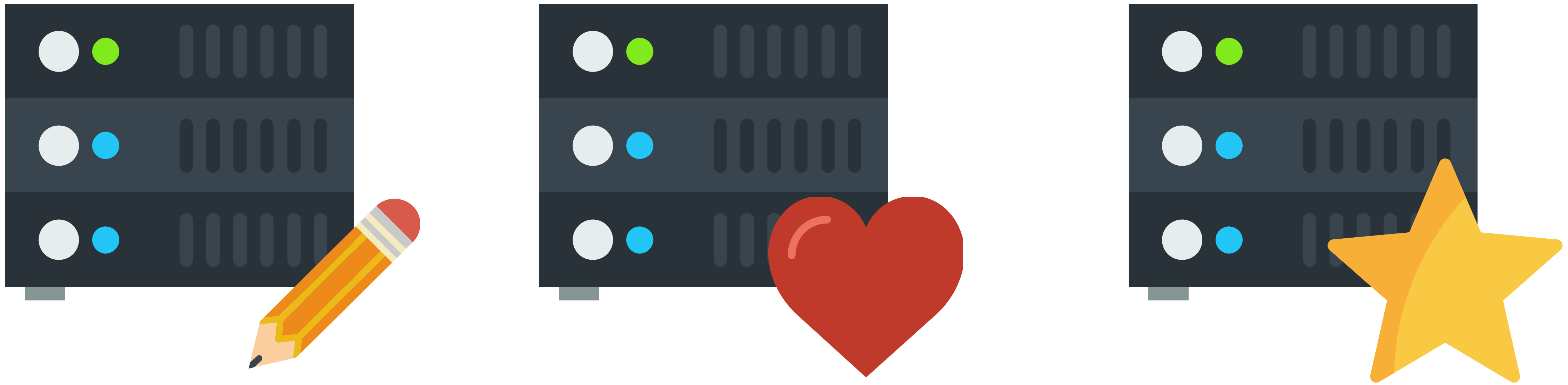
Codebase Dependencies Configuration Port Binding Stateless

Fast Launch Graceful stop Log stream Concurrent

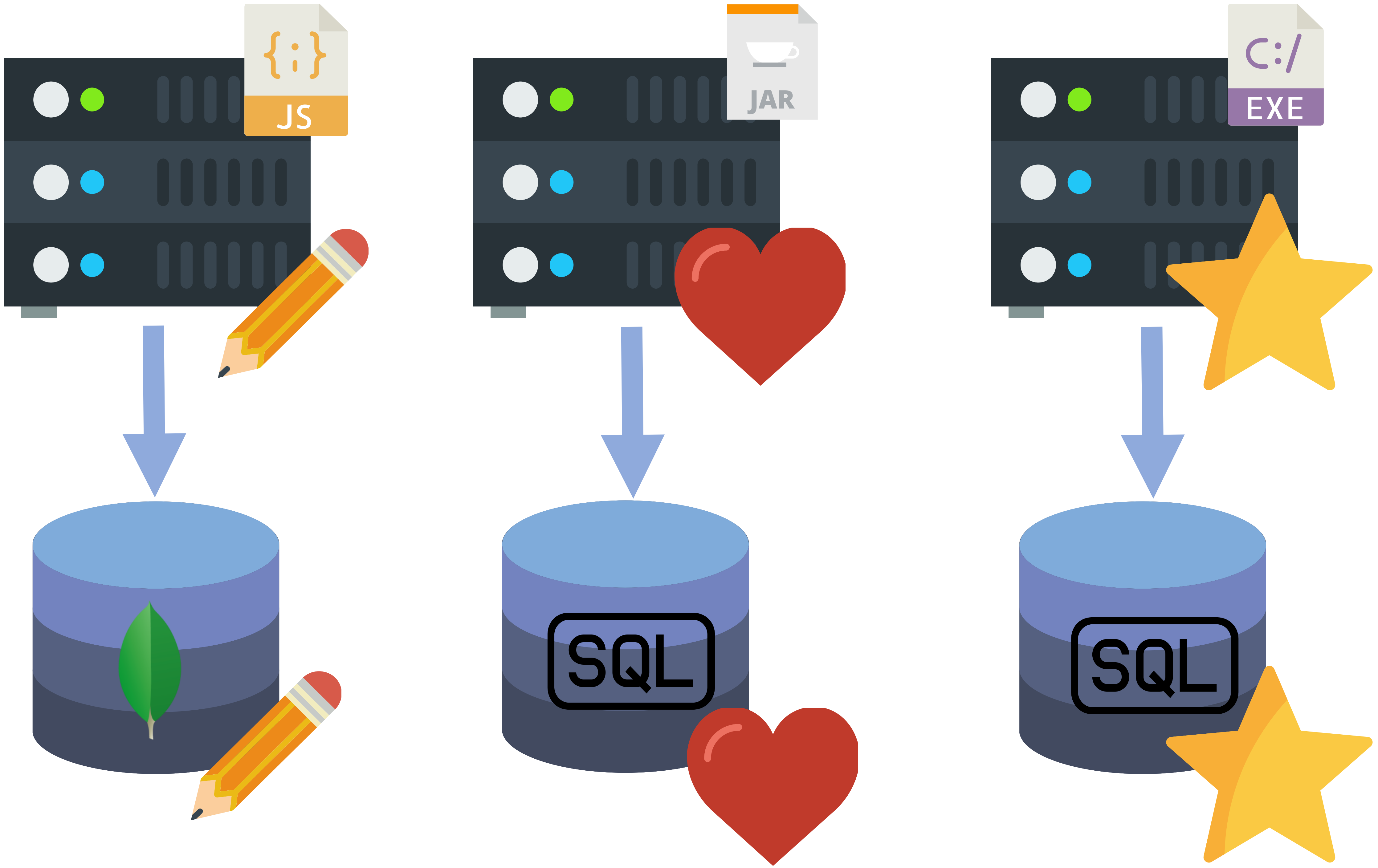
Microservices:
Organized around capabilities

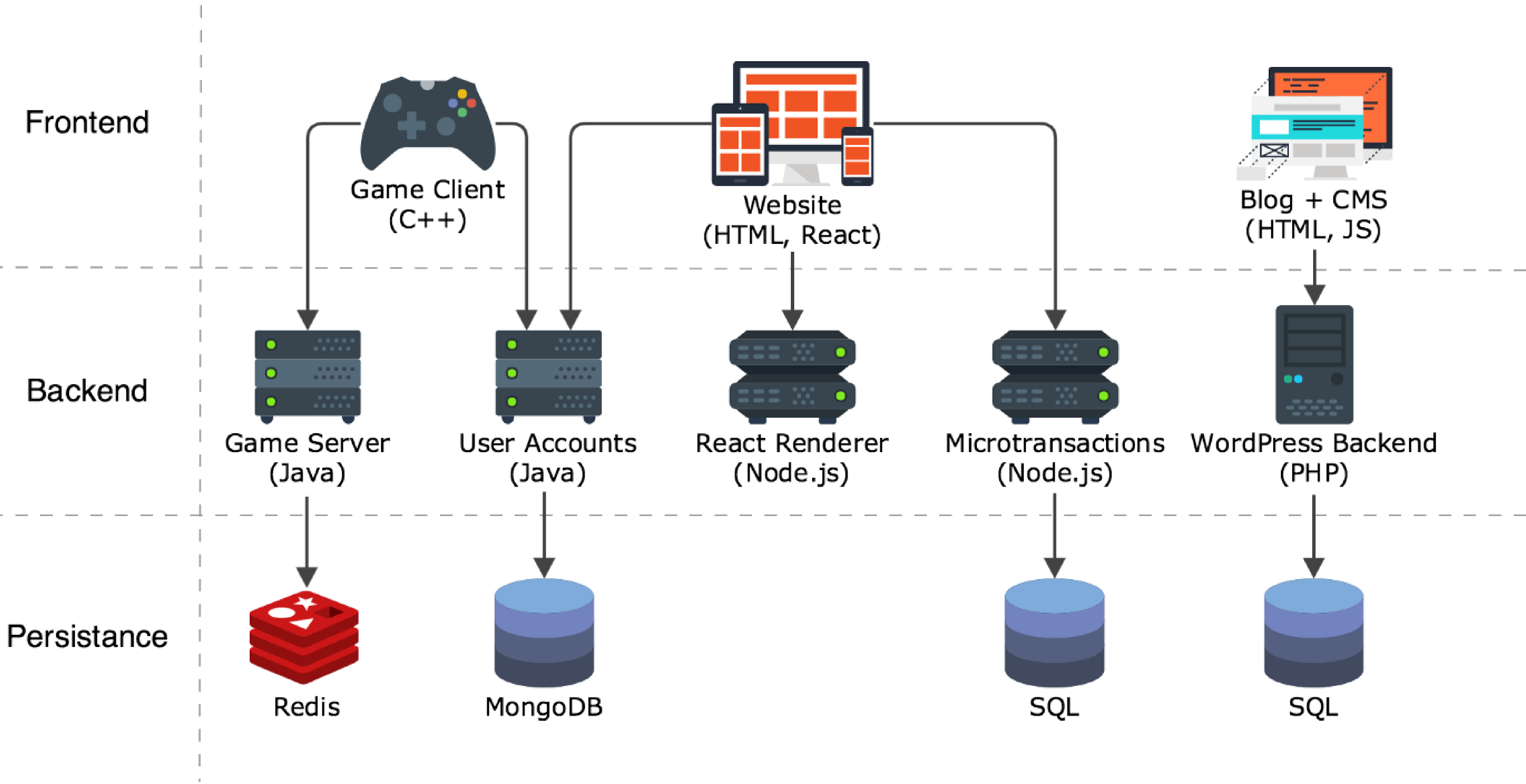
Identify the capabilities of the platform

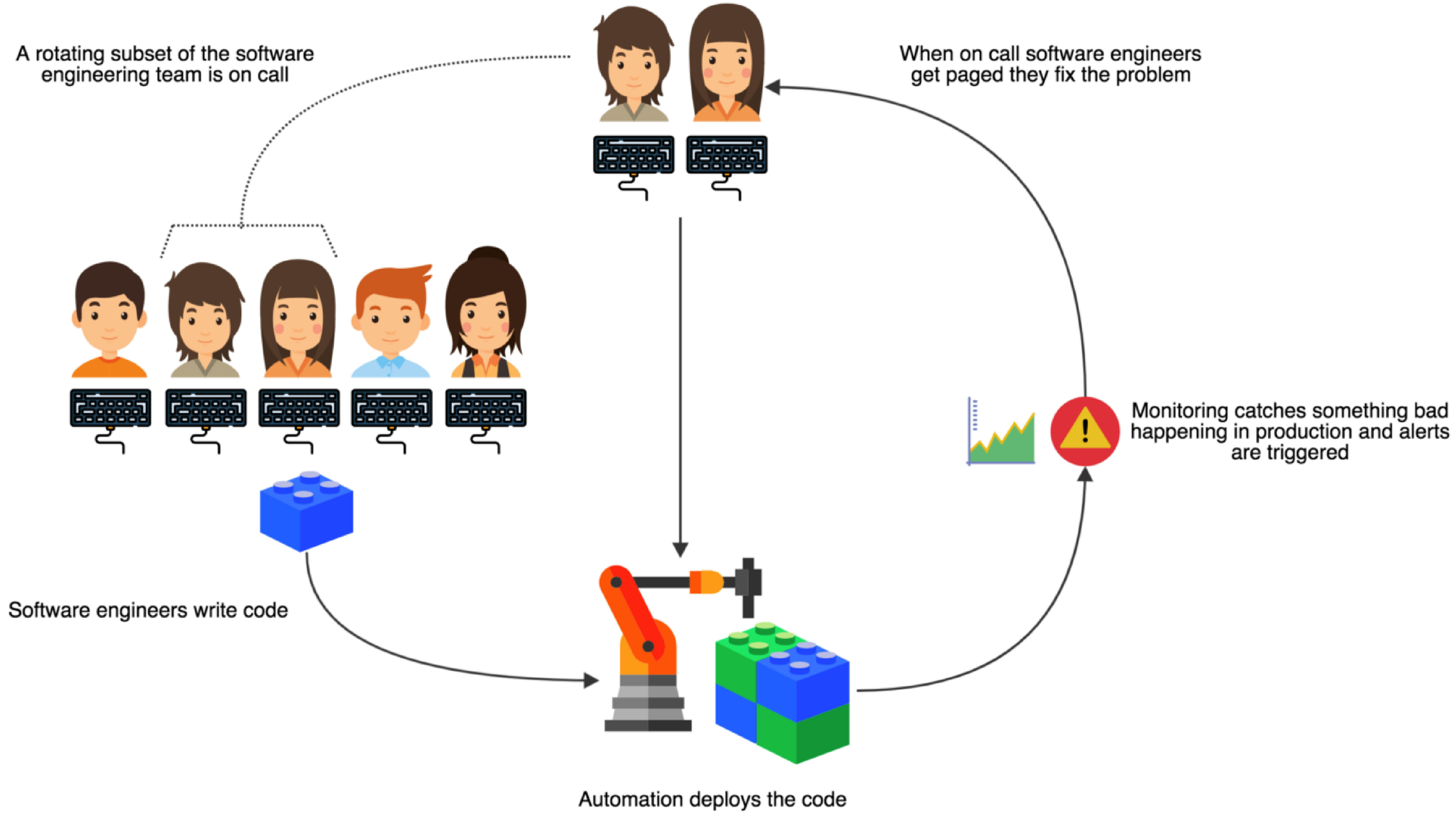
Each major capability of the platform becomes a component that is its own 12 factor app



Microservices:
Decentralized Governance

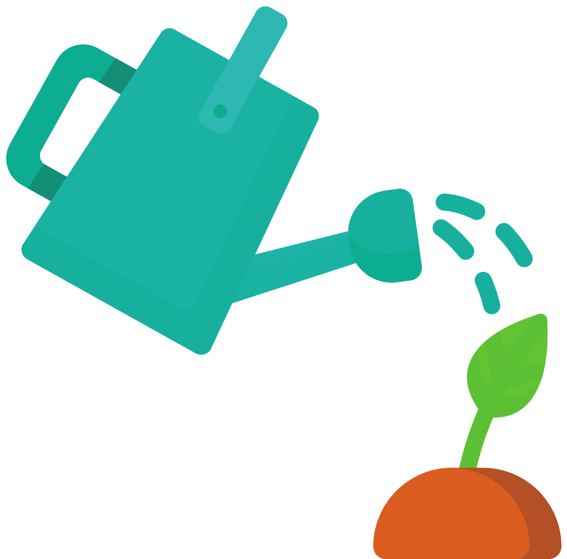

{ : }
JS
JAR
C:/
EXE
SQL
SQL

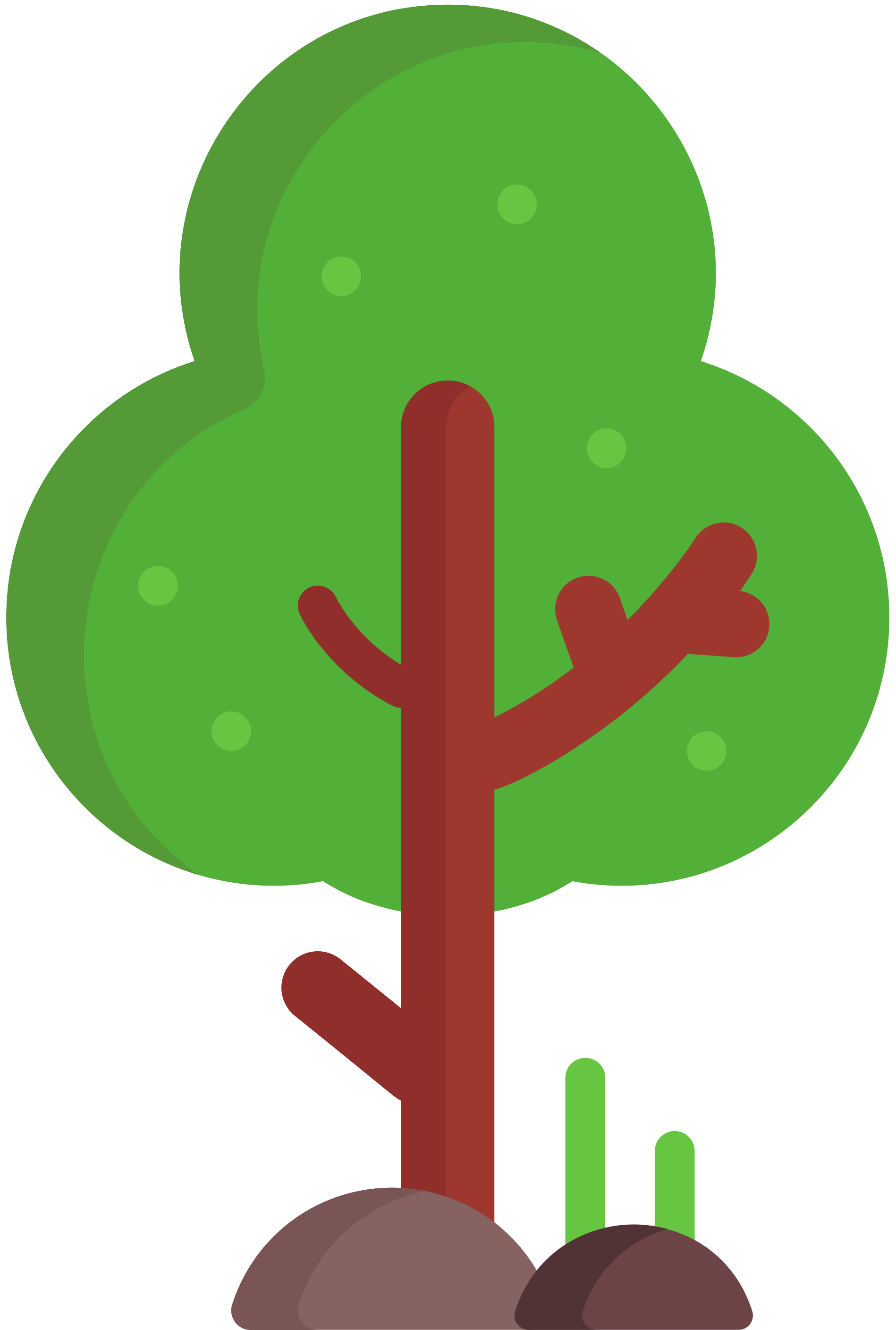

Frontend
Backend
Persistance
Game Client (C++)
Website (HTML, React)
Blog + CMS (HTML, JS)
Game Server (Java)
User Accounts (Java)
React Renderer (Node.js)
Microtransactions (Node.js)
WordPress Backend (PHP)
Redis
MongoDB
SQL
SQL


A rotating subset of the software engineering team is on call
When on call software engineers get paged they fix the problem
Monitoring catches something bad happening in production and alerts are triggered
Software engineers write code
Automation deploys the code

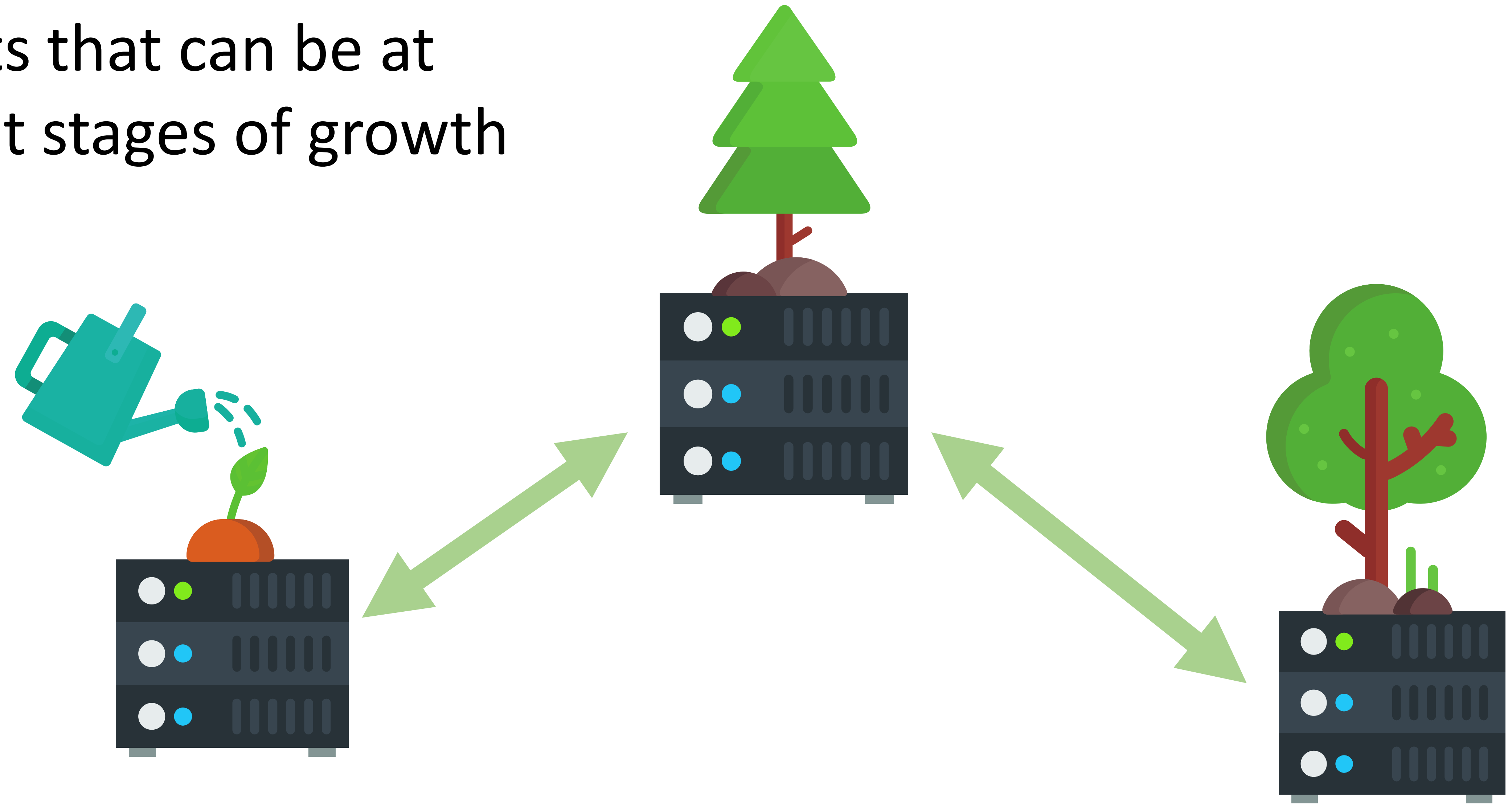
Microservices:
Products Not Projects

Products grow over time

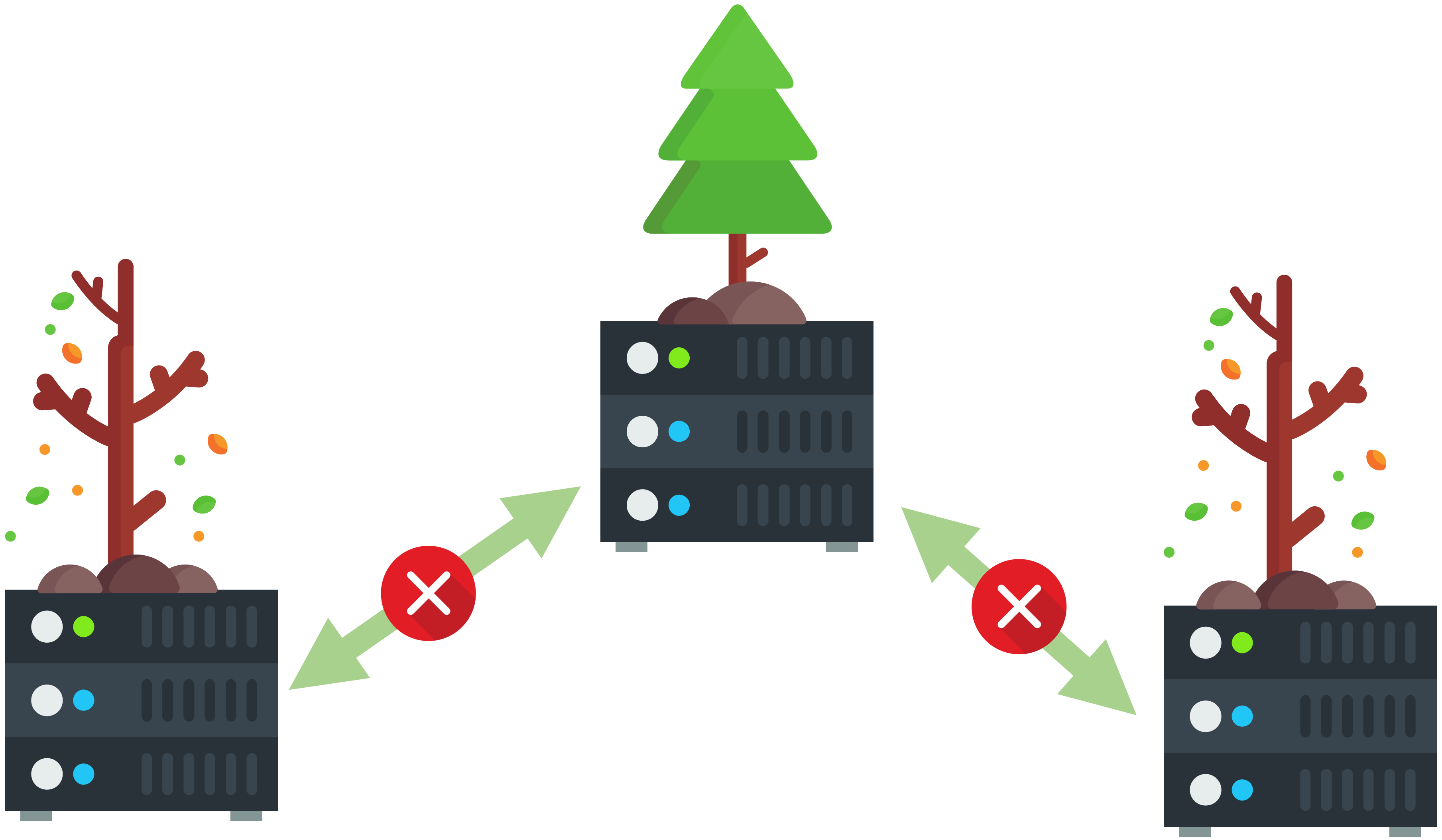




Microservices are an ecosystem of connected products that can be at different stages of growth

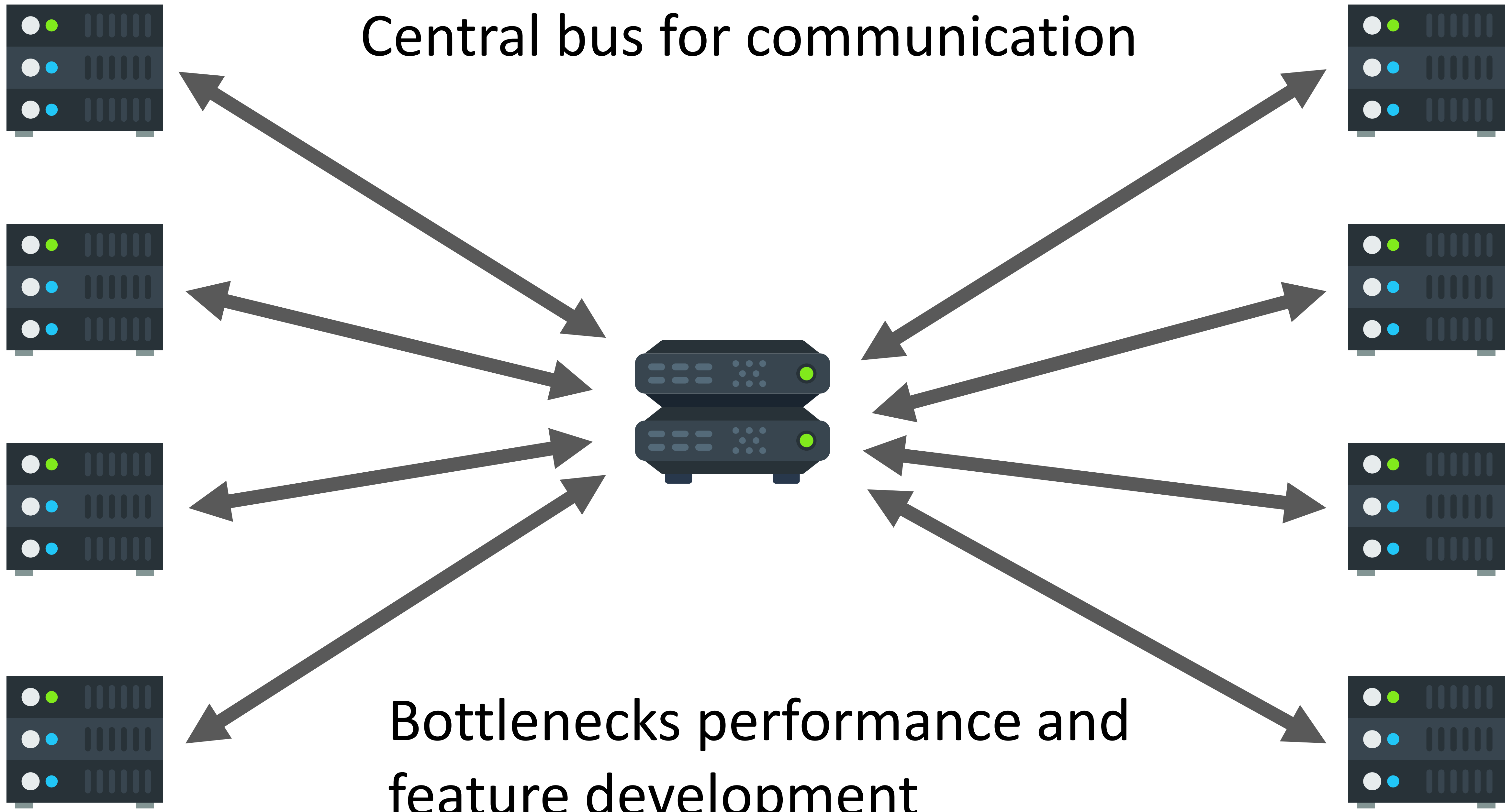


Don’t create throwaway microservices that become unmaintained and break the ecosystem.



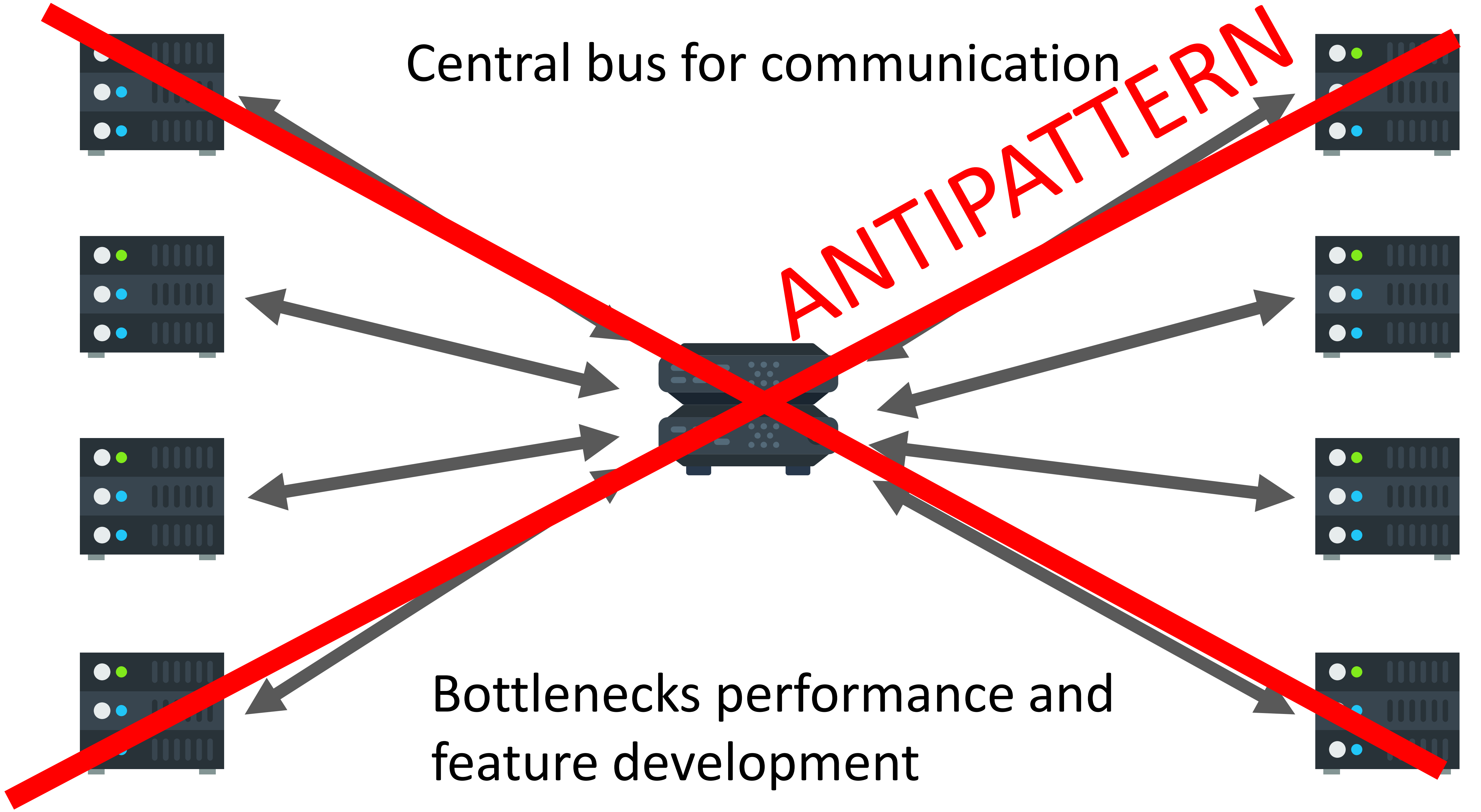
Microservices:
Smart endpoints, Dumb pipes

Central bus for communication



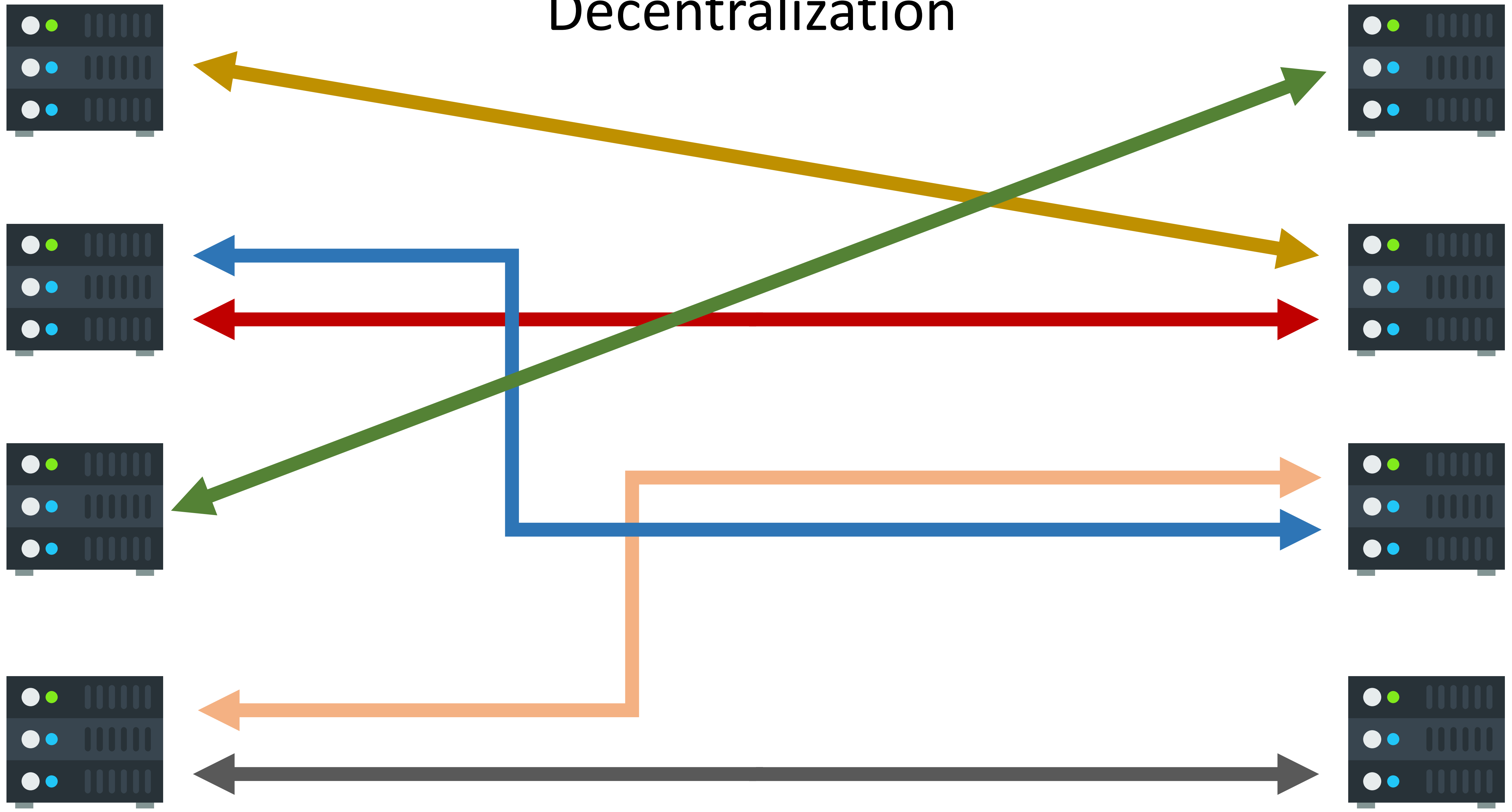
Bottlenecks performance and feature development

Central bus for communication

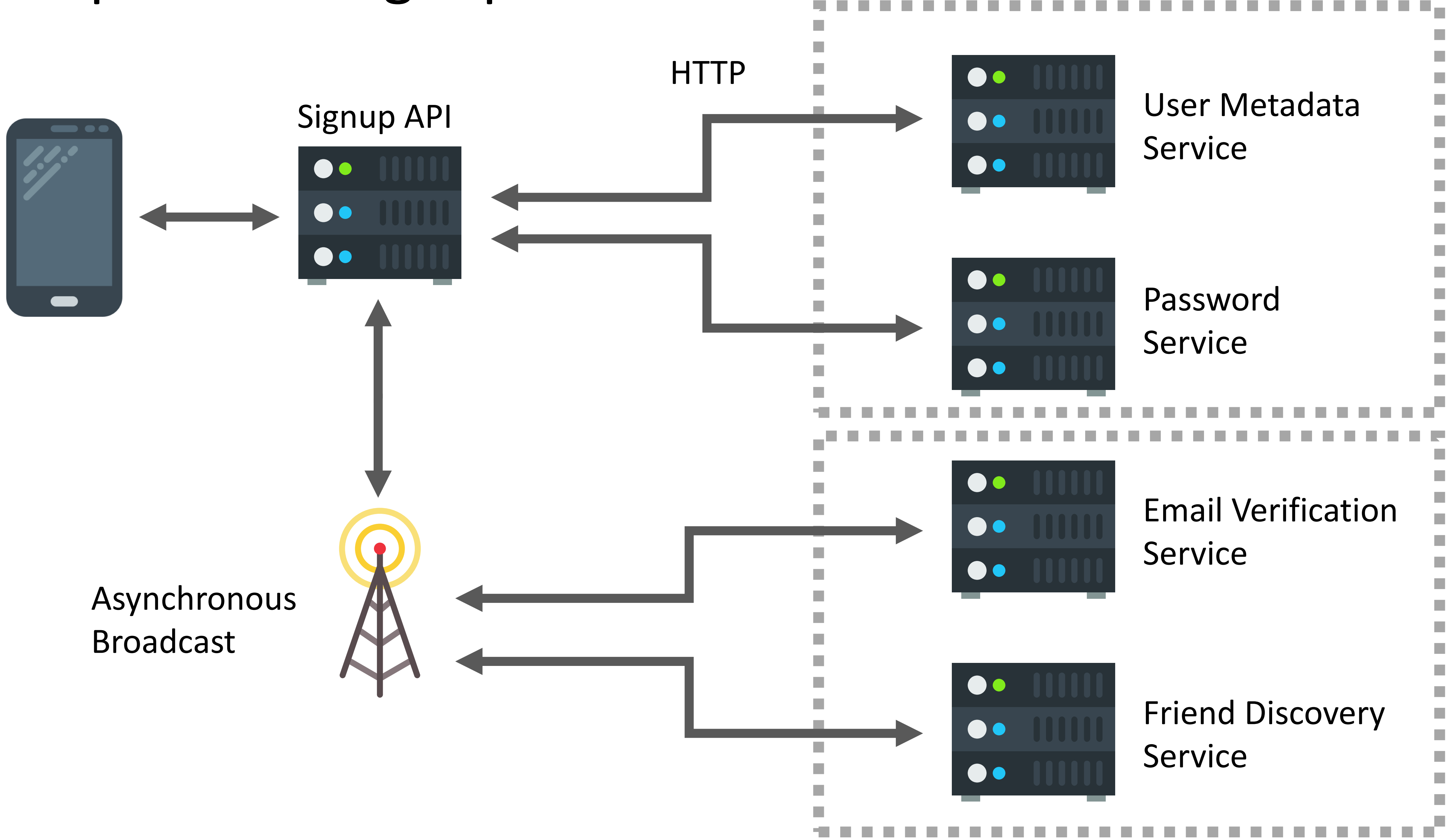


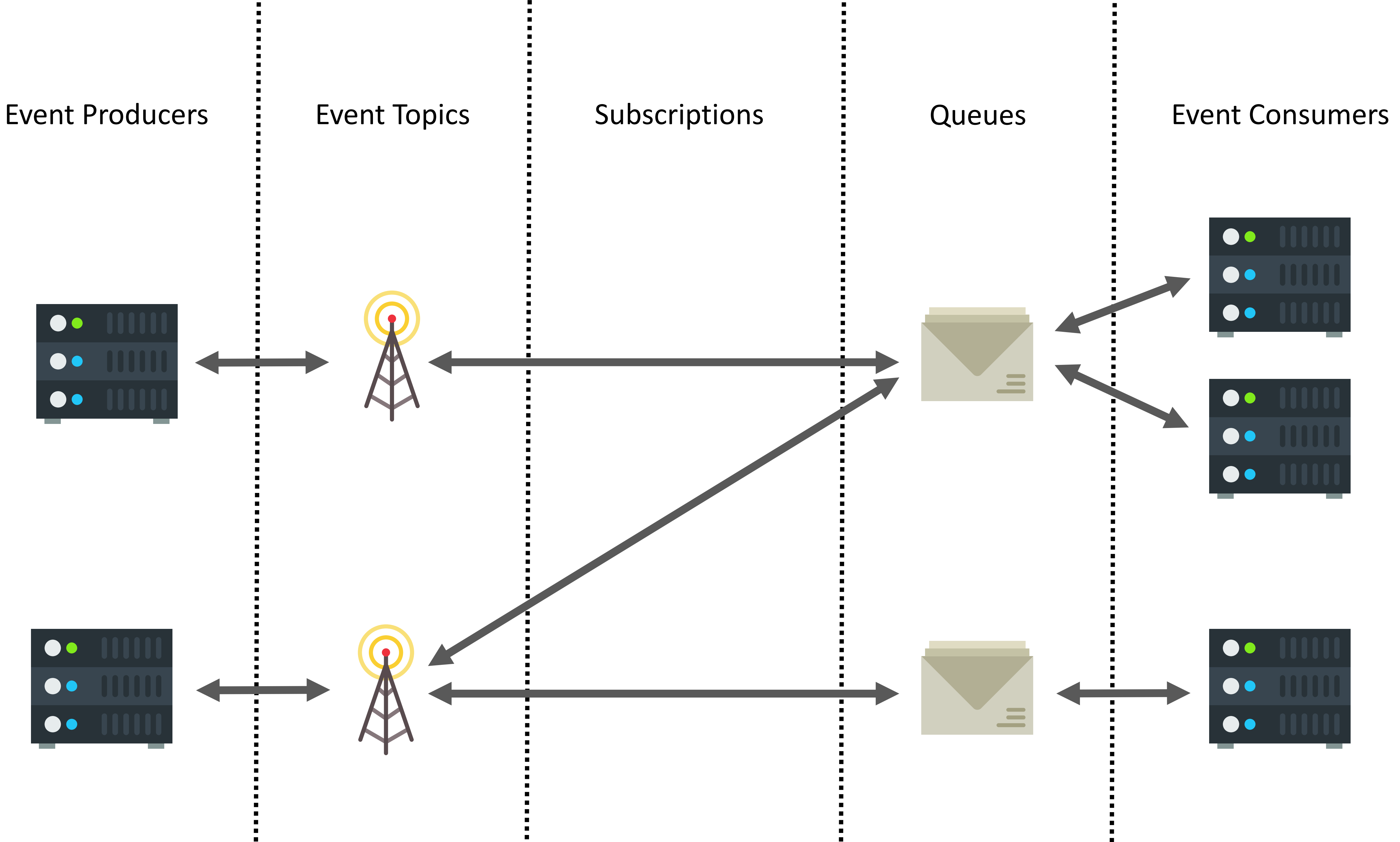
Bottlenecks performance and feature development

Decentralization



Example: User Signup




Event Producers
Event Topics
Subscriptions
Queues
Event Consumers

Amazon Managed Service for Microservice Communication

Amazon Simple Notification Service (SNS)

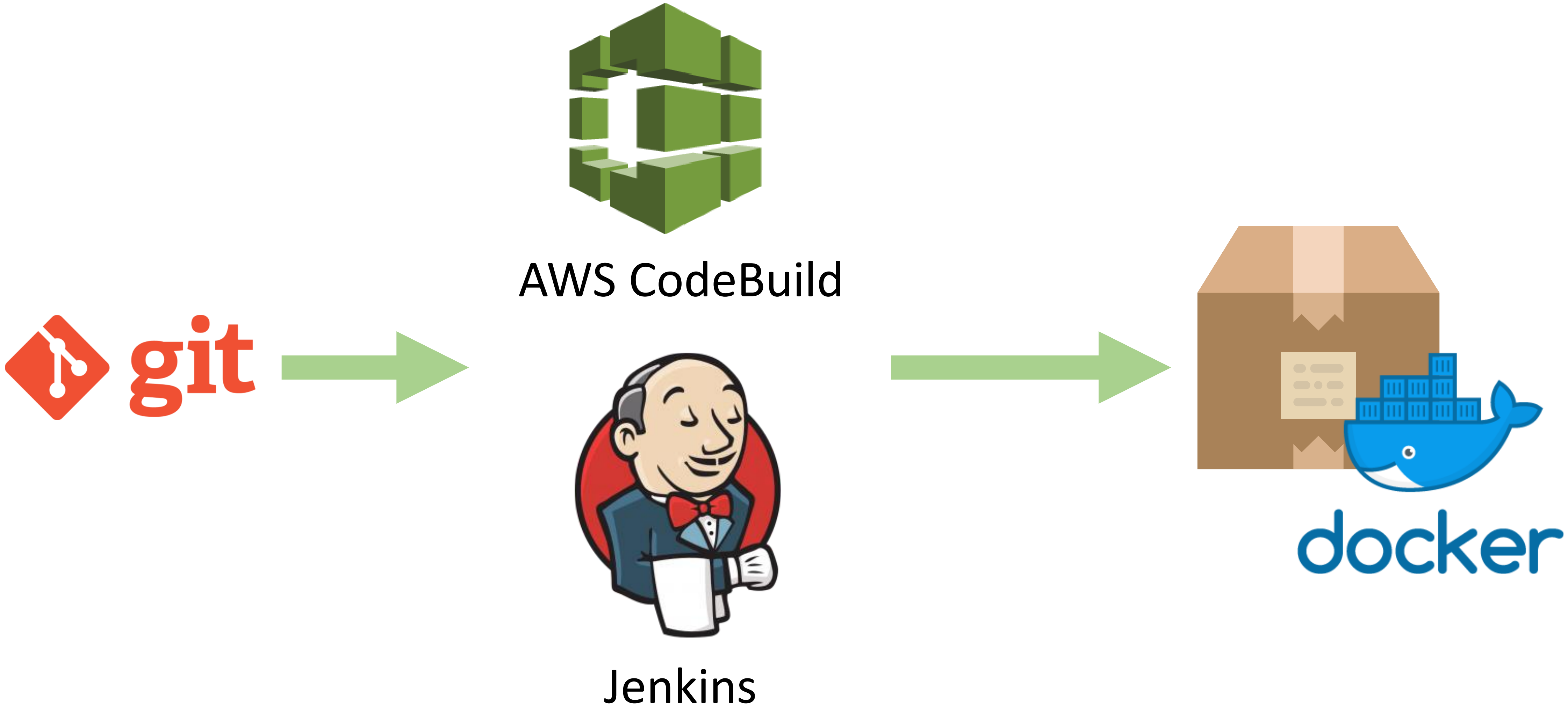
Amazon Simple Queue Service (SQS)

Amazon MQ

Managed message broker service for Apache ActiveMQ

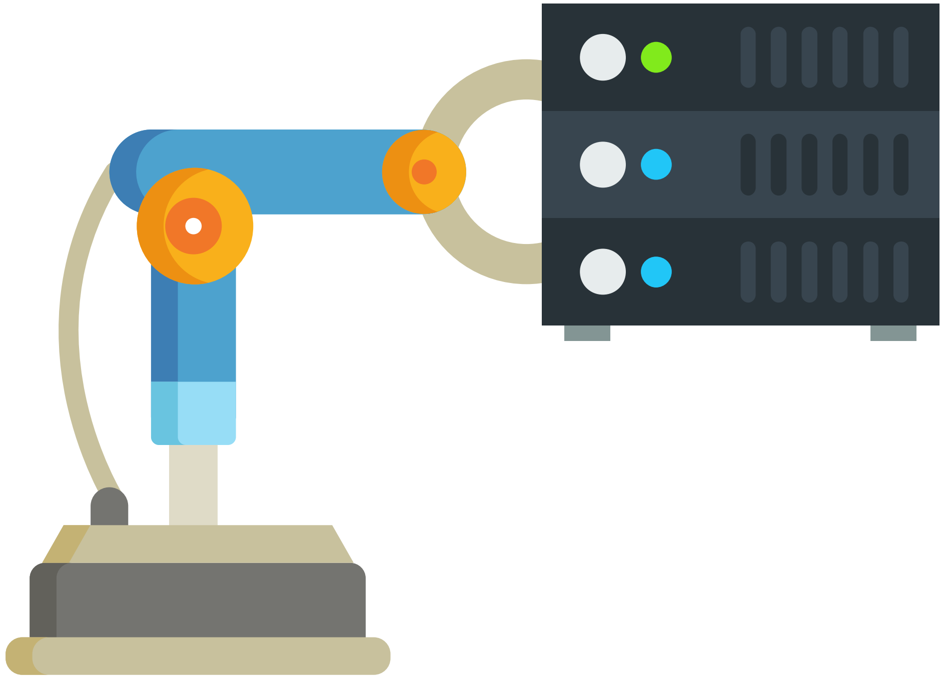
Microservices: Infrastructure Automation

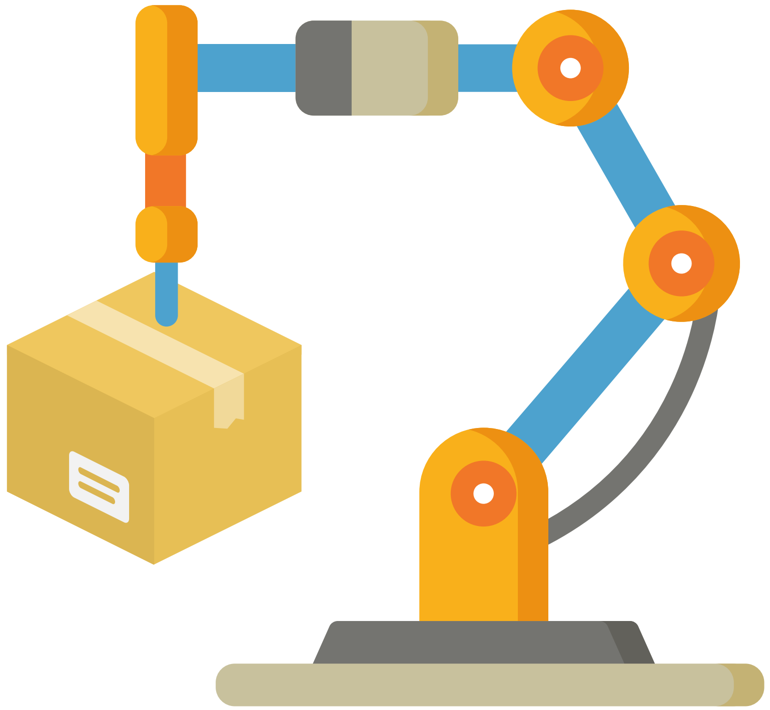
Automate the container build process:



Automate the provisioning of the servers that host microservice containers:

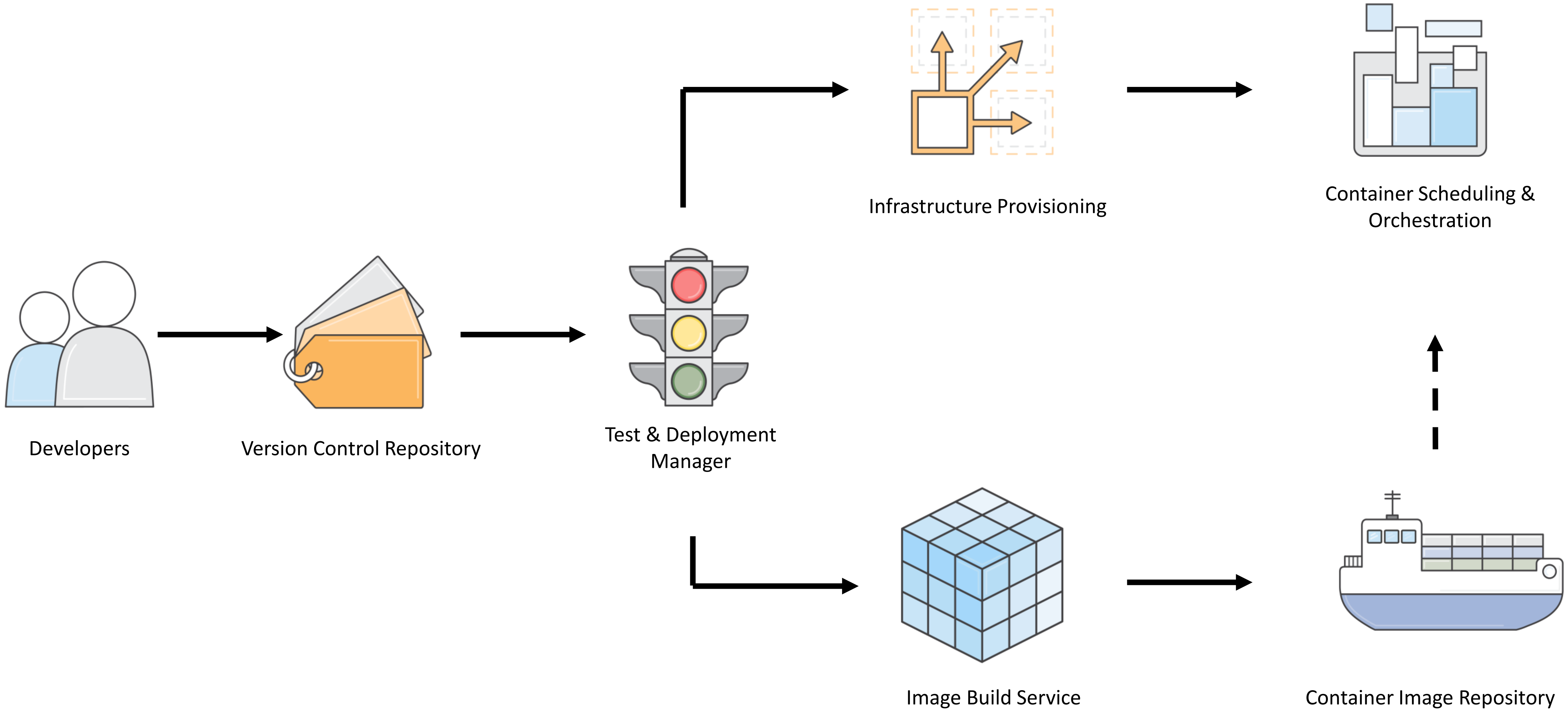
Terraform, Ansible, Amazon CloudFormation

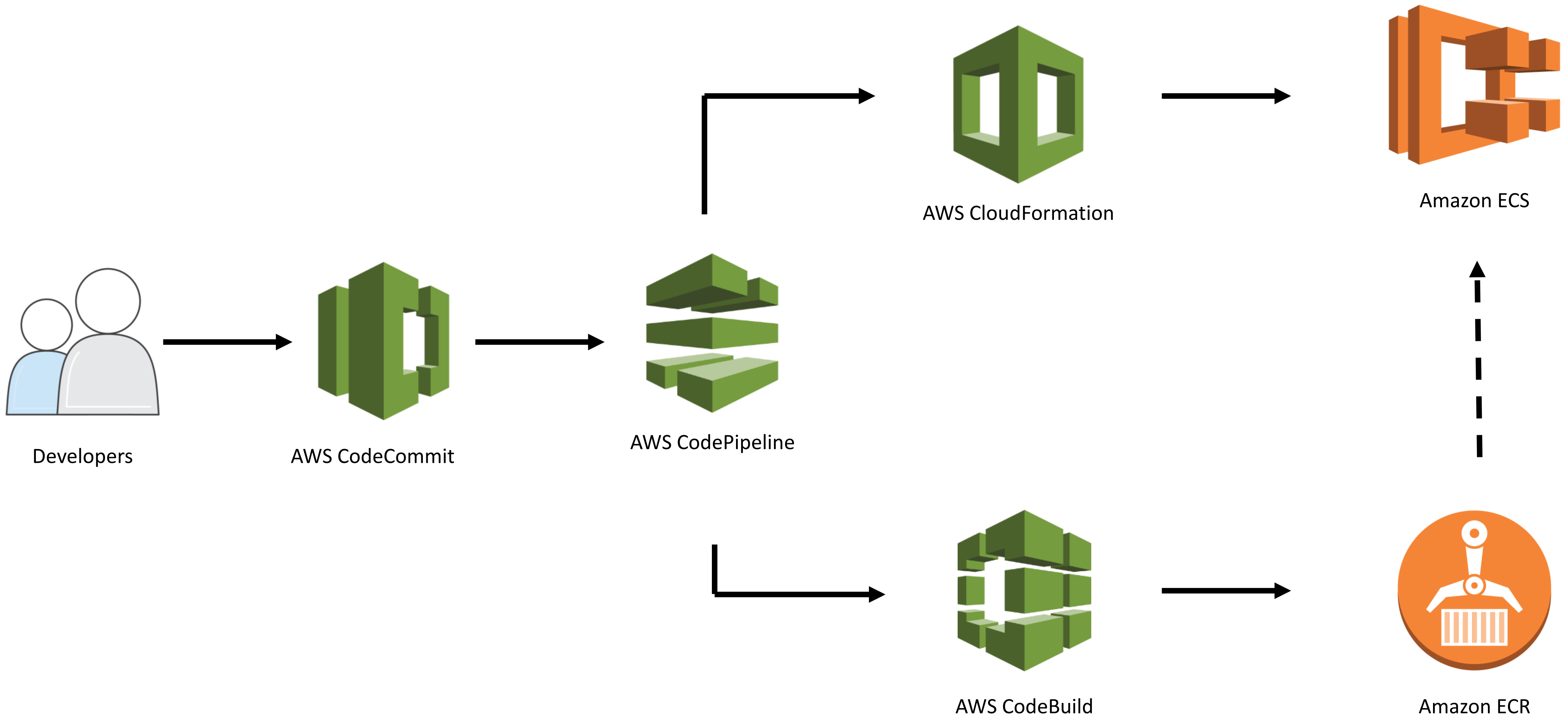


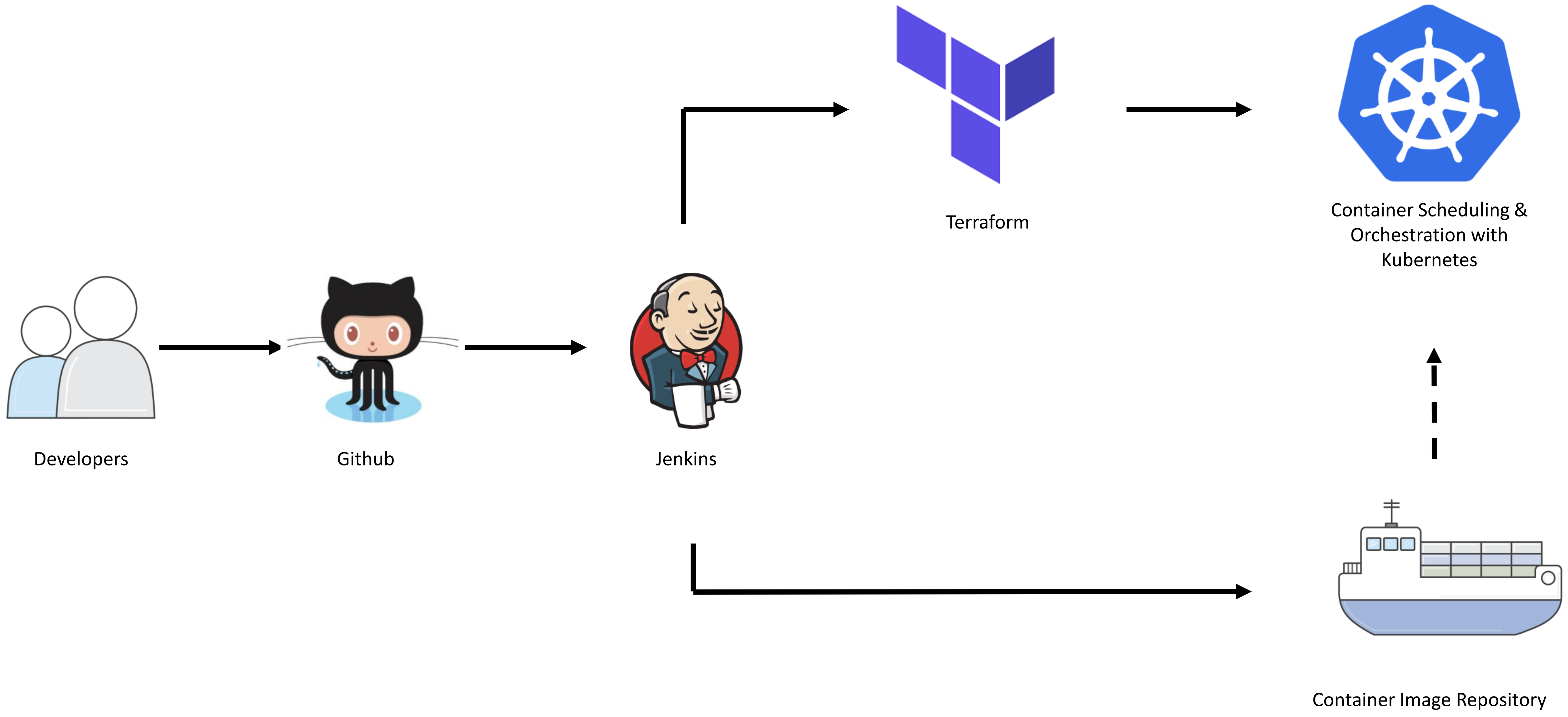


Automate the placement of containerized service processes onto hosts:

Amazon Elastic Container Service, Kubernetes, Docker Swarm


Developers
Version Control Repository
Test & Deployment Manager
Infrastructure Provisioning
Container Scheduling & Orchestration
Image Build Service
Container Image Repository


Developers
AWS CodeCommit
AWS CodePipeline
AWS CloudFormation
Amazon ECS
AWS CodeBuild
Amazon ECR


Developers
Github
Jenkins
Terraform
Container Scheduling & Orchestration with Kubernetes
Container Image Repository

Summary

12 factor application principles

Codebase

Dependencies

Configuration

Port Binding

Stateless

Fast Launch

Graceful stop

Log stream

Concurrent

Microservice principles

Automation

Componentization

Product Focused

Decentralization

Thank you!

peckn@amazon.com

nathankpeck

nathanpeck