UNIVERSITÀ CATTOLICA del Sacro Cuore

Cloud Native Concepts

Evolution of application architectures

Late 90's	**Enterprise Application (EAI) Services and Models** Addressed integration and transactional challenges primarily by using message oriented middleware. Mostly proprietary systems needing a proliferation of custom interfaces.
Mid 00's	**Service Oriented Architectures** Based on open protocols like SOAP and WSDL (web service description language, XML based) making integration and adoption easier. Usually deployed on an Enterprise ESB which is hard to manage and scale.
Early 10's	**API Platforms and API Management** REST and JSON become the defacto standard for consuming backend data. Mobile apps become major consumers of backend data. New Open protocols like OAuth become available further simplifying API development .
2015 and beyond	**Cloud Native and Microservice Architecture** Applications are composed of small, independently deployable processes communicating with each other using language-agnostic APIs and protocols.

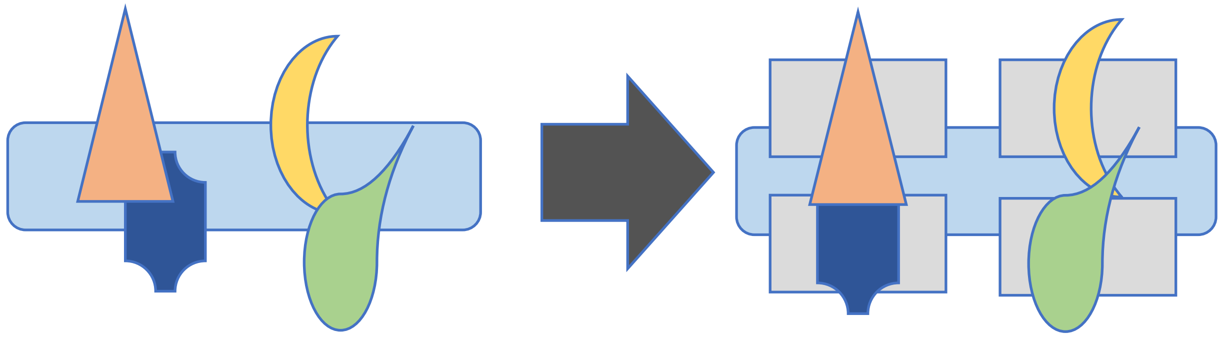
Key tenets of a cloud native application

1. Packaged as light weight **containers**
2. Developed with best-of-breed languages and frameworks
3. Designed as loosely coupled **microservices**
4. Centered around **APIs** for interaction and collaboration
5. Architected with a clean separation of stateless and stateful services
6. Isolated from server and operating system dependencies
7. Deployed on self-service, elastic, **cloud infrastructure**
8. Managed through agile **DevOps** processes
9. Automated capabilities
10. Defined, policy-driven resource allocation

https://thenewstack.io/10-key-attributes-of-cloud-native-applications/

Key tenets of a microservices architecture

1. Large monoliths are broken down into many small services
2. Services are optimized for a single function or business capability
3. Teams that write the code should also deploy the code
4. Smart endpoints, dumb pipes (message brokers)
5. Decentralized governance
6. Decentralized data management

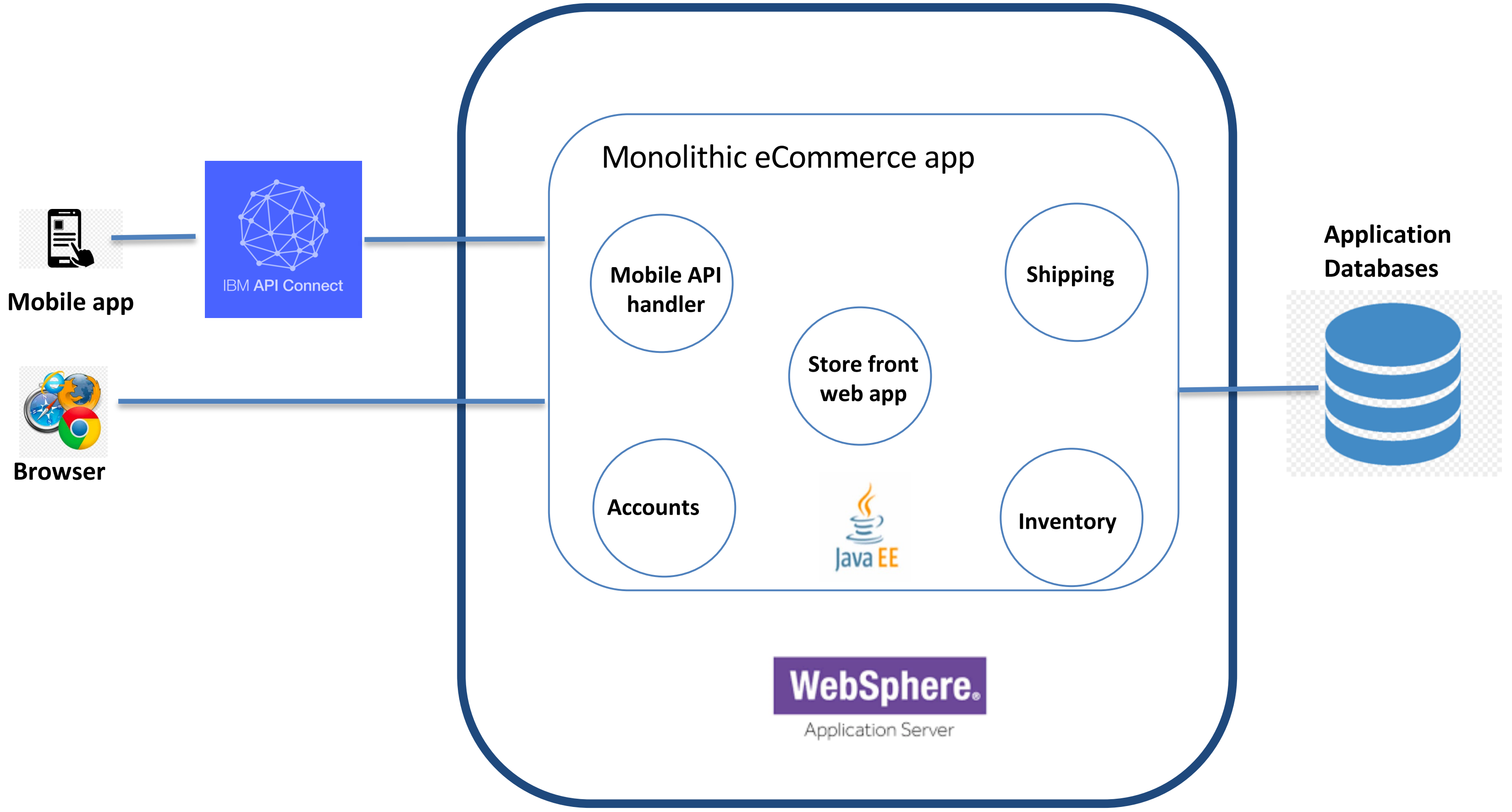


https://martinfowler.com/articles/microservices.html

Example monolithic application

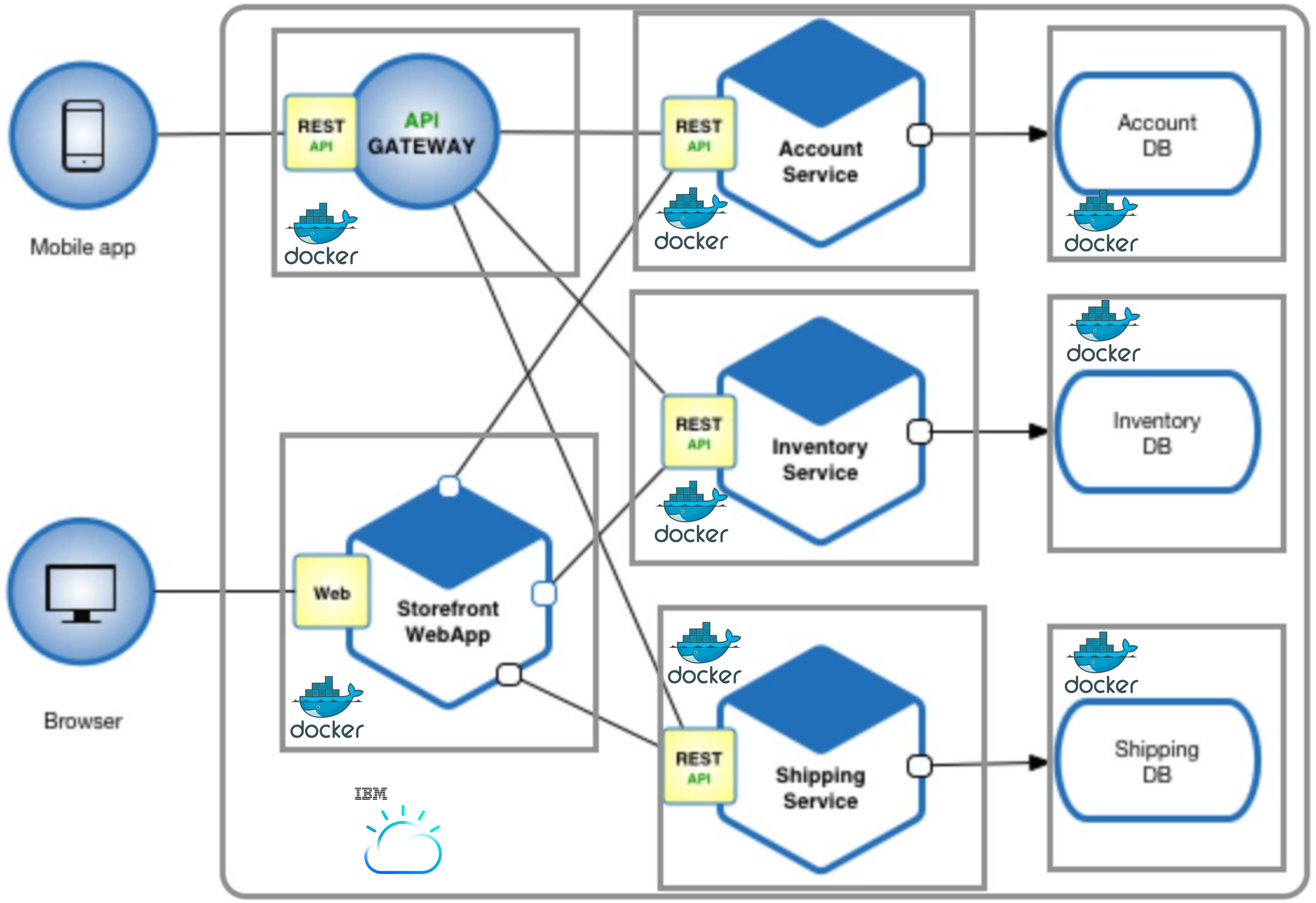
eCommerce app

- Store front web interface
- Customer Accounts
- Inventory
- Shipping
- Back end for mobile app



Transformed application

- Key technologies
  - Containers (Docker)
  - Container orchestration (Kubernetes)
  - 12-Factor Best Practices
  - CI/CD tools (e.g Jenkins)



Why microservices and cloud native?

Efficient teams	Simplified deployment	Right tools for the job	Improved application quality	Scalability
• End to end team ownership of relatively small codebases	• Each service is individually changed, tested, and deployed without affecting other services	• Teams can use best of breed technologies, libraries, languages for the job at hand	• Services can be tested more thoroughly in isolation	• Services can be scaled independently at different rates as needed
➢ Teams can innovate faster and fix bugs more quickly	➢ Time to market is accelerated.	➢ Leads to faster innovation	➢ Better code coverage	➢ Leads to better overall performance at lower cost

Cultural change considerations

- **Smaller teams with broader scope**
  - Mini end to end development orgs in each team vs large silos across the entire development team
- **Top down support with bottom up execution**
  - Change can't happen effectively w/o executive sponsorship
  - Change needs to be executed at the smallest organizational unit to take hold
- **Teams own all metrics related to operations and development**
  - Have to minimize downtime + number of bugs while also maximizing the rate at which needed features are added and minimizing the time to market of those new features
- **Trust**
  - Teams need to build trust with other teams that they collaborate with rather than relying on one size fits all checklists and rules
- **Reward based on results not compliance**
  - Cultures only change when people are measured and rewarded for outcomes consistent with the changes
  - Smaller more autonomous teams work better with less central micromanagement and more focus on broad measurable goals

12 Factor Apps

12 Factor is a methodology for building software

The twelve-factor app

- A set of best practices for creating applications
  - Implementing, deploying, monitoring, and managing
- Typical modern applications
  - Deployed in the cloud
  - Accessible as web applications that deliver software-as-a-service (SaaS)
- Can be applied to any application
  - Implemented in any programming language
  - Using any backing services such as database, messaging, and caching
- Addresses common problems
  - The dynamics of the growth of an app over time
  - The dynamics of collaboration between developers
  - Avoiding the cost of software erosion
  - Systemic problems in modern application development
- Provides a shared vocabulary for addressing these problems

Tenets for a 12 Factor App

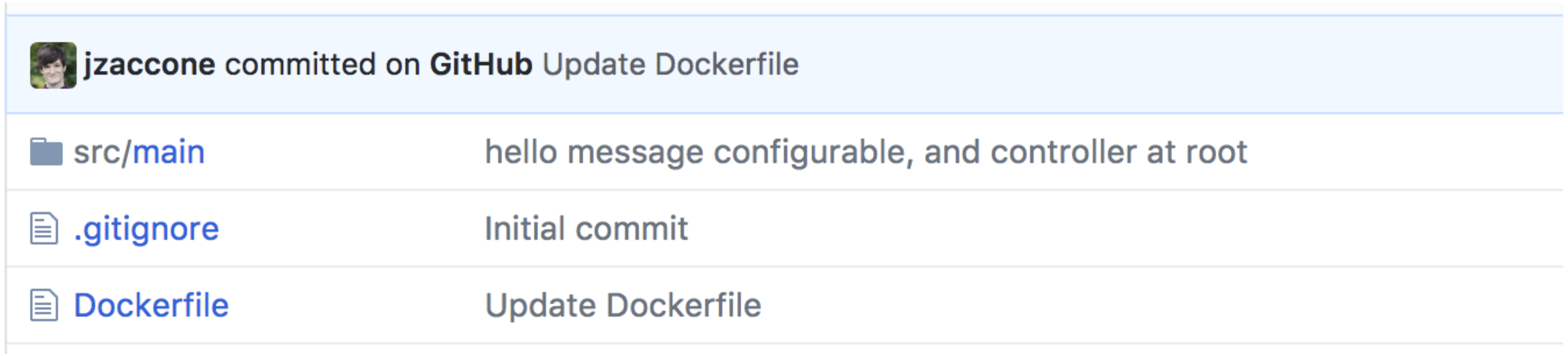
1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, Release, Run
6. Processes
7. Port Binding
8. Concurrency
9. Disposability
10. Dev/Prod Parity
11. Logs
12. Admin processes

https://12factor.net

I. Codebase

- Code for a single application should be in a single code base

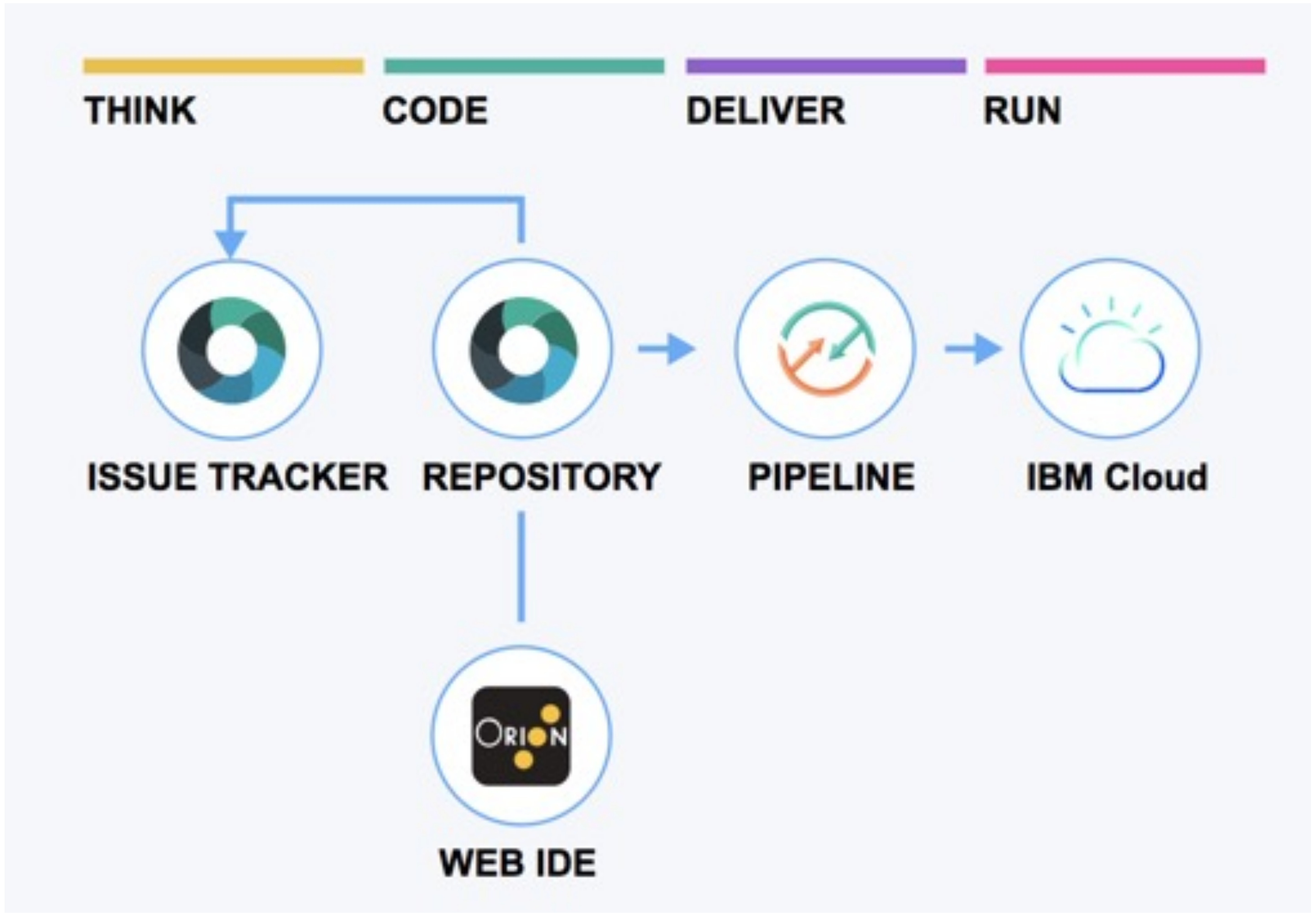
- Track running applications back to a single commit
- Use Dockerfile Maven, Gradle, or npm to manage external dependencies
- Version pinning! Don’t use latest
- No changing code in production



jzaccone committed on **GitHub** Update Dockerfile

src/main	hello message configurable, and controller at root
.gitignore	Initial commit
Dockerfile	Update Dockerfile

Codebase: IBM Cloud continuous delivery toolchains



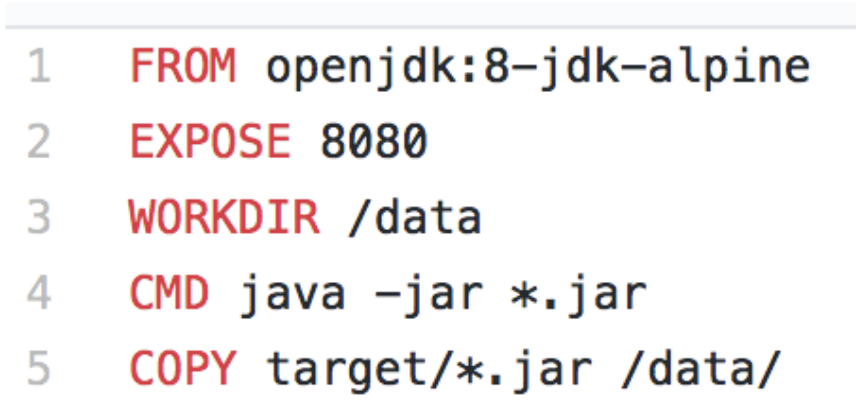
A *toolchain* is a set of tool integrations that support development, deployment, and operations tasks.

Tool integrations with the source code repository and delivery pipelines can drive multiple deployments from a single repository.

II. Dependencies

- Explicitly declare and isolate dependencies. AKA: Remove system dependencies

- **How?**
- Step 1: Explicitly declare dependencies (Dockerfile)
- Step 2: Isolate dependencies to prevent system dependencies from leaking in (containers)



```
FROM openjdk:8-jdk-alpine
EXPOSE 8080
WORKDIR /data
CMD java -jar *.jar
COPY target/*.jar /data/
```

III. Config

- Store config in the environment (not in the code).

- **How?**
- Inject config as environment variables (language agnostic)
- ConfigMap in Kubernetes does this ^

- `$ docker run -e POSTGRES_PASSWORD=abcd postgres`

IV. Backing Services

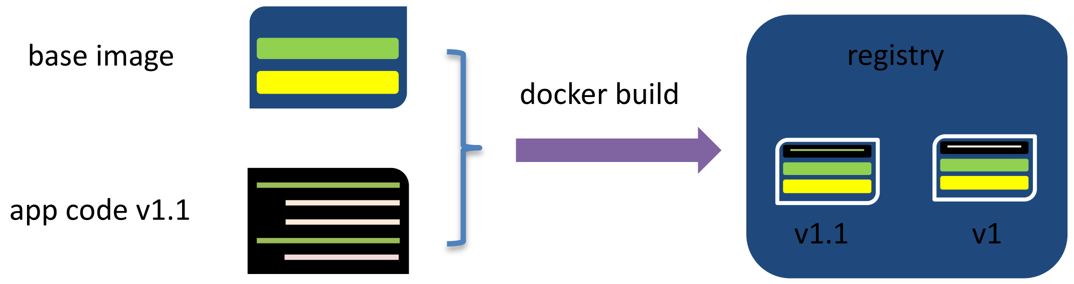
- Treat backing resources as attached services. Swap out resources.

- **How?**
- Pass in URLs via config (see III.)
- K8s built in DNS (entry in the etcd component) allows for easy service discovery

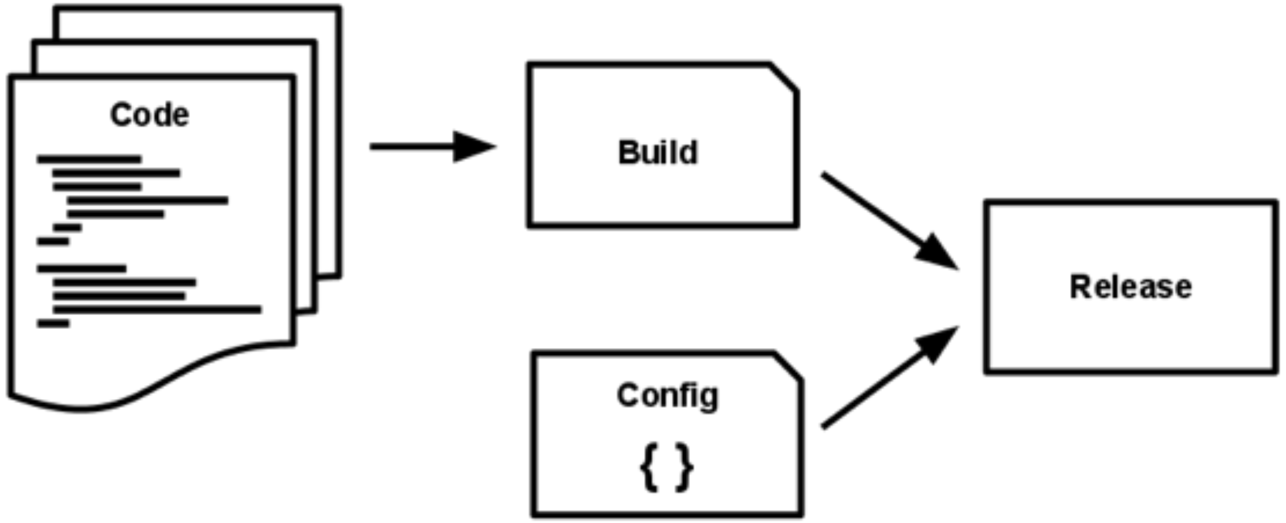
```
services:

  account-api:
    build:
      context: ./compute-interest-api
    environment:
      DATABASE_URL: http//account-database

  account-database:
    image: jzaccone/account-database
```

V. Build, Release, Run

- Strictly separate build and run stages.
- **Why?**
- Rollbacks, elastic scaling without a new build
- **How?**
- Use Docker images as your handoff between build and run
- Tag images with version. Trace back to single commit (see I. Codebase)
- Single command rollbacks in Kubernetes



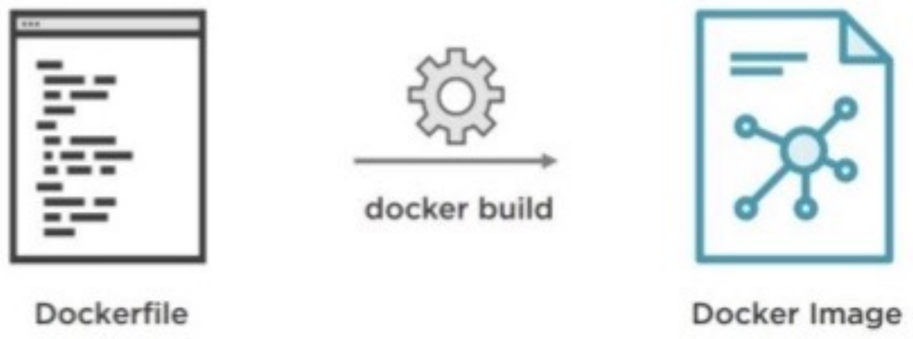
- Build stage
  - converts a code repo into an executable bundle known as a build
  - fetches dependencies and compiles binaries and assets
- Release stage
  - takes the build produced by the build stage and combines it with the deploy’s current config
  - ready for immediate execution in the execution environment
- Run stage
  - runs the app in the execution environment



- An immutable image does not get changed – only used for deploying instances
- If it needs to change, you delete it and create a new one

Examples

- Docker containers
  - A container image is created from a Dockerfile
  - After that, it is only deployed as a container, not changed
  - If you need to make changes, make a new container image by creating a new build and running the Dockerfile again

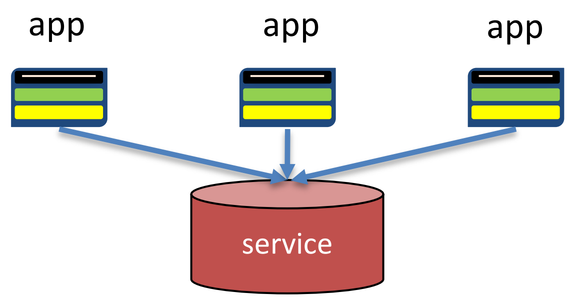


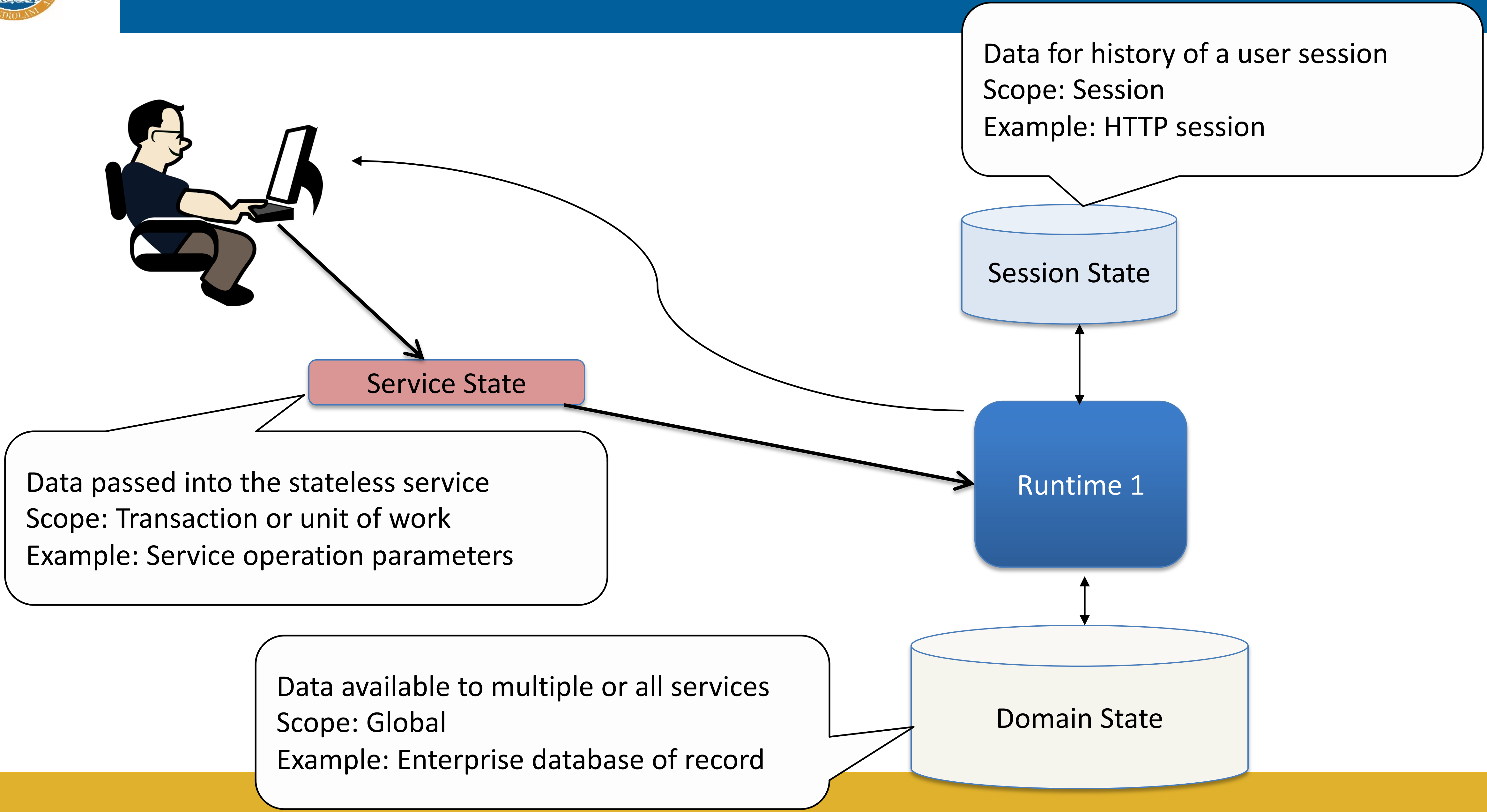
VI. Process

- Execute app as stateless process

- **Why?**
- Stateless enables horizontal scaling

- **How?**
- Remove sticky sessions
- Need state? Store in volume or external data service
- Use persistent volumes in Kubernetes for network wide storage




Data for history of a user session
Scope: Session
Example: HTTP session
Session State
Service State
Runtime 1
Data passed into the stateless service
Scope: Transaction or unit of work
Example: Service operation parameters
Data available to multiple or all services
Scope: Global
Example: Enterprise database of record
Domain State

VII. Port Binding

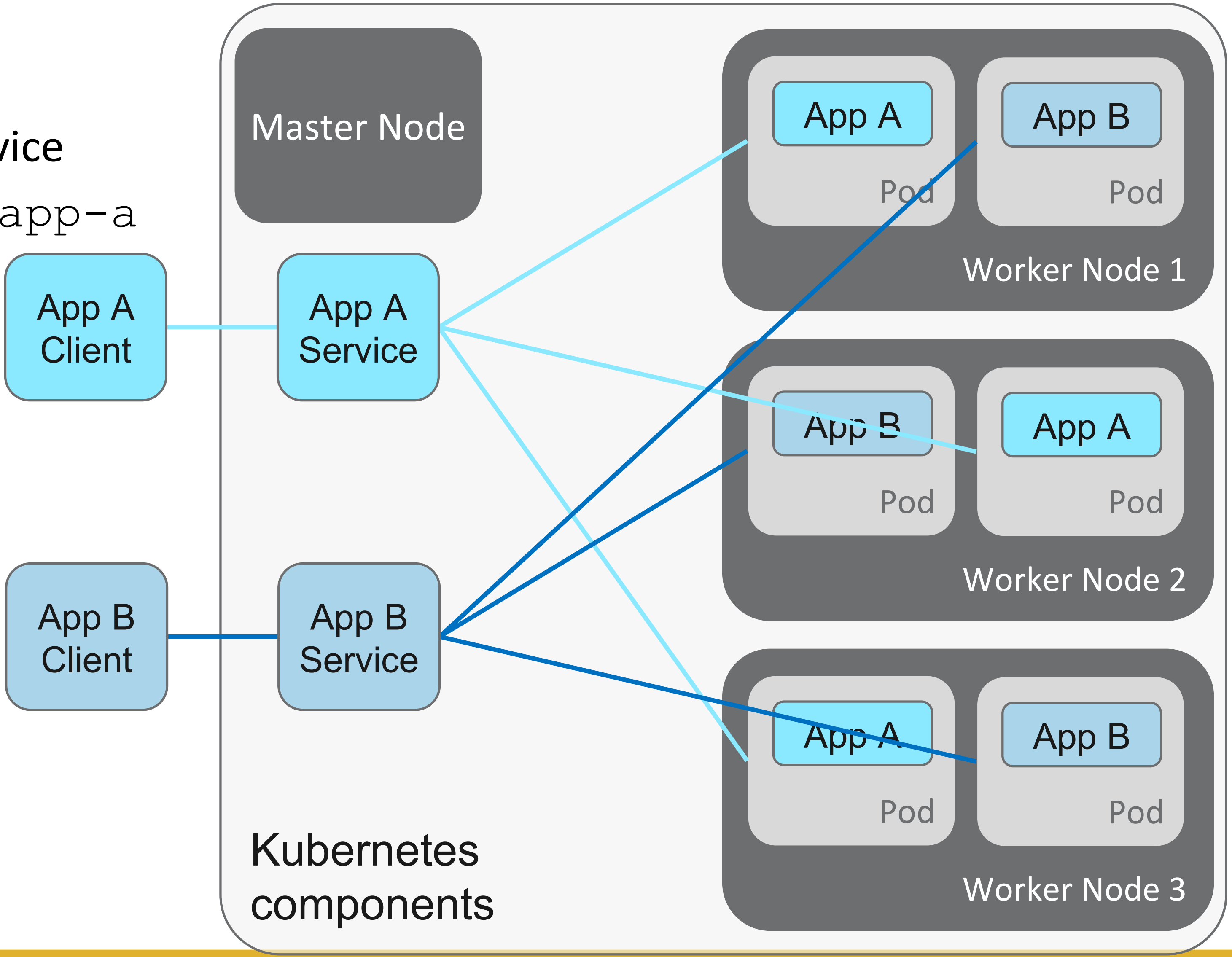
- Export services via port binding. Apps should be self-contained.

- **Why?**
- Avoid “Works on my machine”

- **How?**
- Web server dependency should be included inside the Docker Image
- To expose ports from containers use the —publish flag

- Expose a set of pod replicas as a service

```
kubectl expose deployment/app-a
    --type=LoadBalancer
    --port=8080
    --name=app-a-service
    --target-port=8080
```

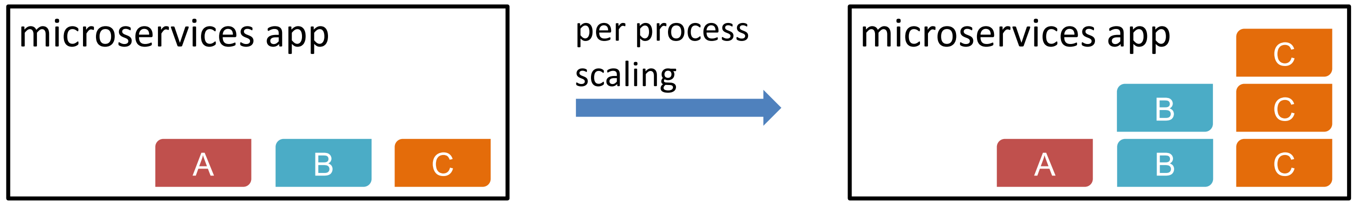


VIII. Concurrency

- Scale out via the process model. Processes are **first-class citizens**

- **Why?**
- Follow the Unix model for scaling, which is simple and reliable

- **How?**
- Scale by creating more processes
- **Docker**: really just a process running in isolation
- **Kubernetes**: Acts as process manager: scales by creating more pods
  Don’t put process managers in your containers

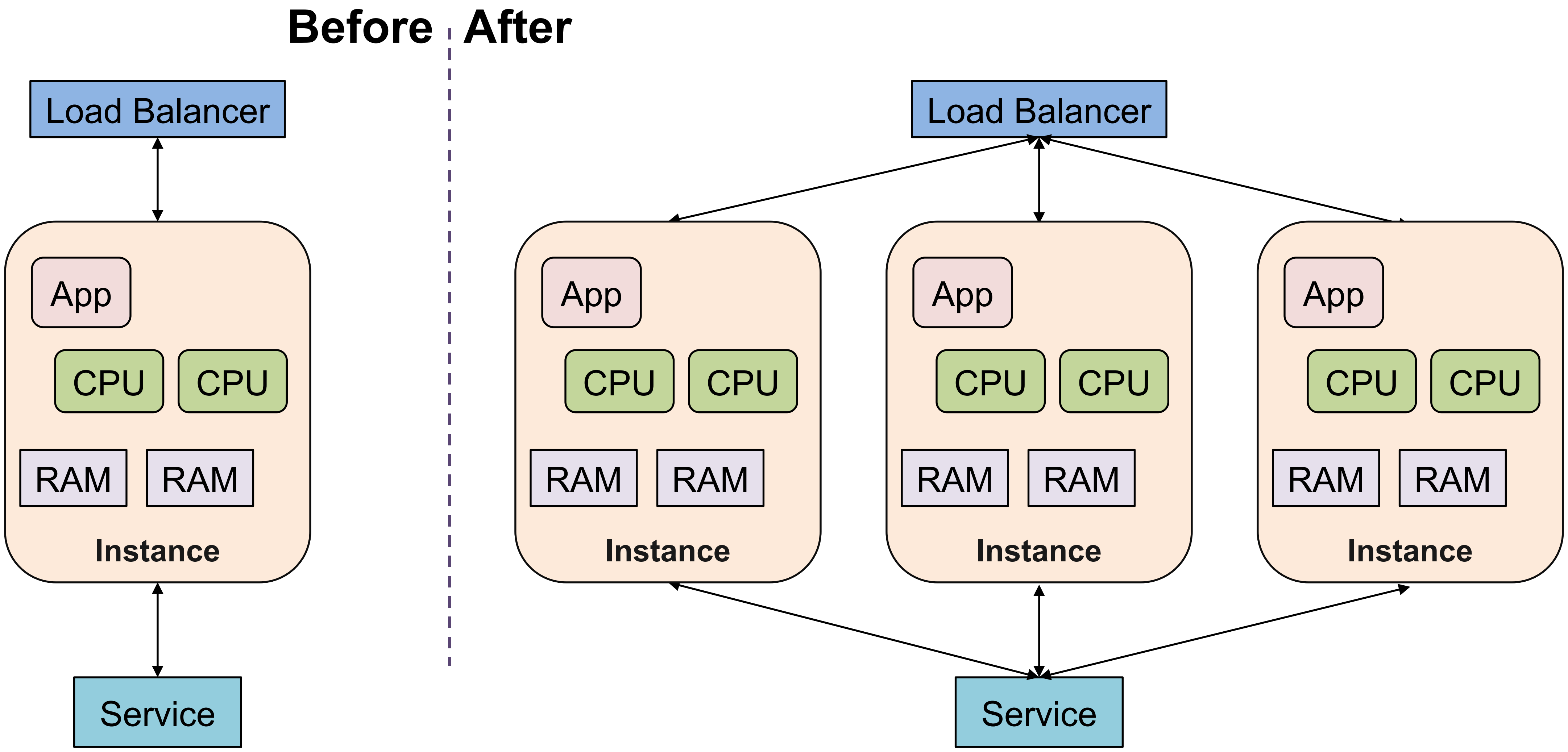


Bad Example

```
# Start the first process
./my_first_process -D
status=$?
if [ $status -ne 0 ]; then
  echo "Failed to start my_first_process: $status"
  exit $status
fi

# Start the second process
./my_second_process -D
status=$?
if [ $status -ne 0 ]; then
  echo "Failed to start my_second_process: $status"
  exit $status
fi
```

```
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

Containers should be a single process!

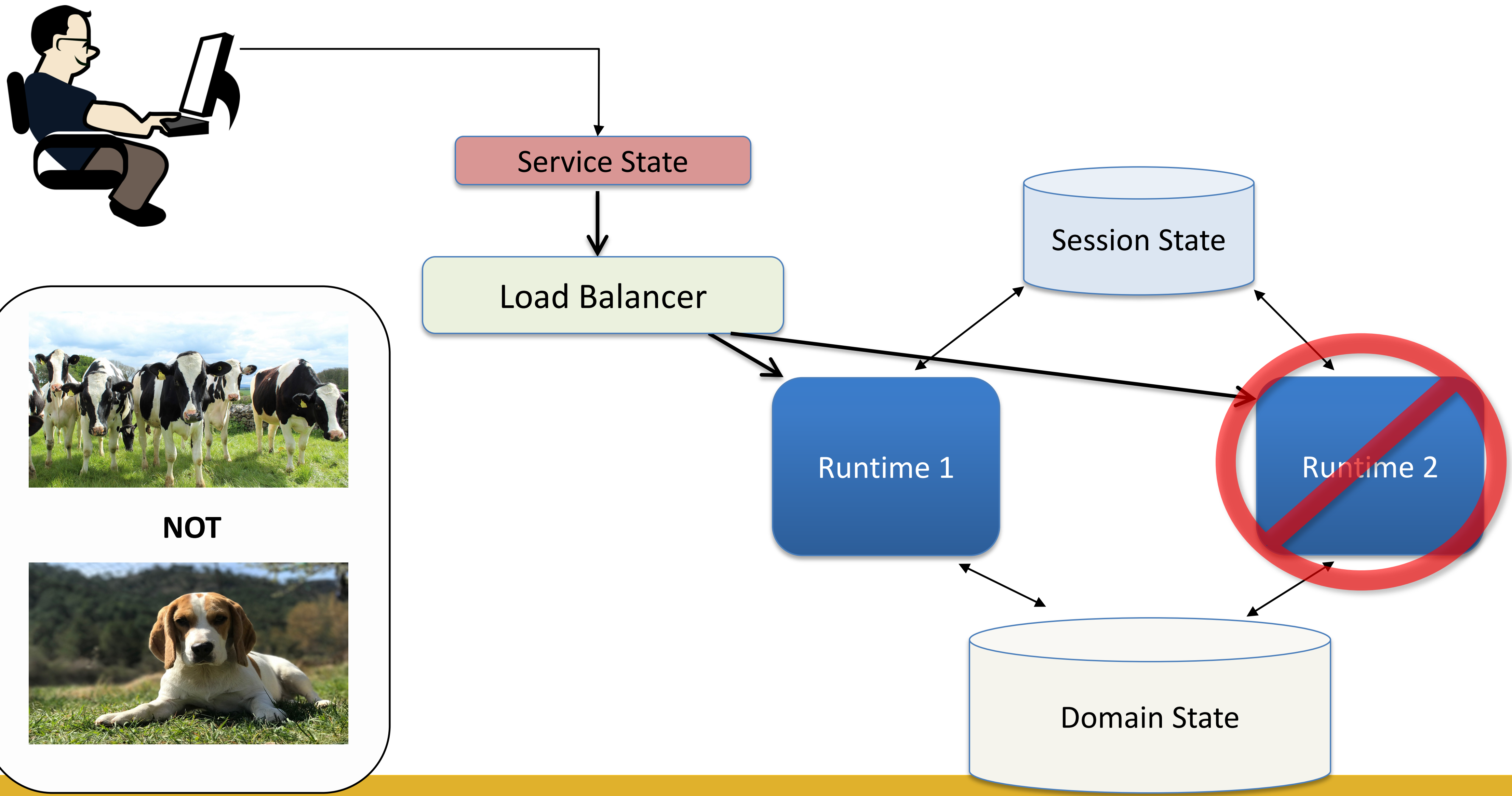

Before
After
Load Balancer
App
CPU
CPU
RAM
RAM
Instance
Service
Load Balancer
App
CPU
CPU
RAM
RAM
Instance
App
CPU
CPU
RAM
RAM
Instance
App
CPU
CPU
RAM
RAM
Instance
Service

IX. Disposability

- Maximize robustness with fast startup and graceful shutdown

- **Why?**
- Enables fast elastic scaling, robust production deployments. Recover quickly from failures.

- **How?**
- No multi-minute app startups!
- Docker enables fast startup: Union file system and image layers
- In best practice: Handle SIGTERM in main container process.


Service State
Load Balancer
Session State
Runtime 1
Runtime 2
Domain State
NOT

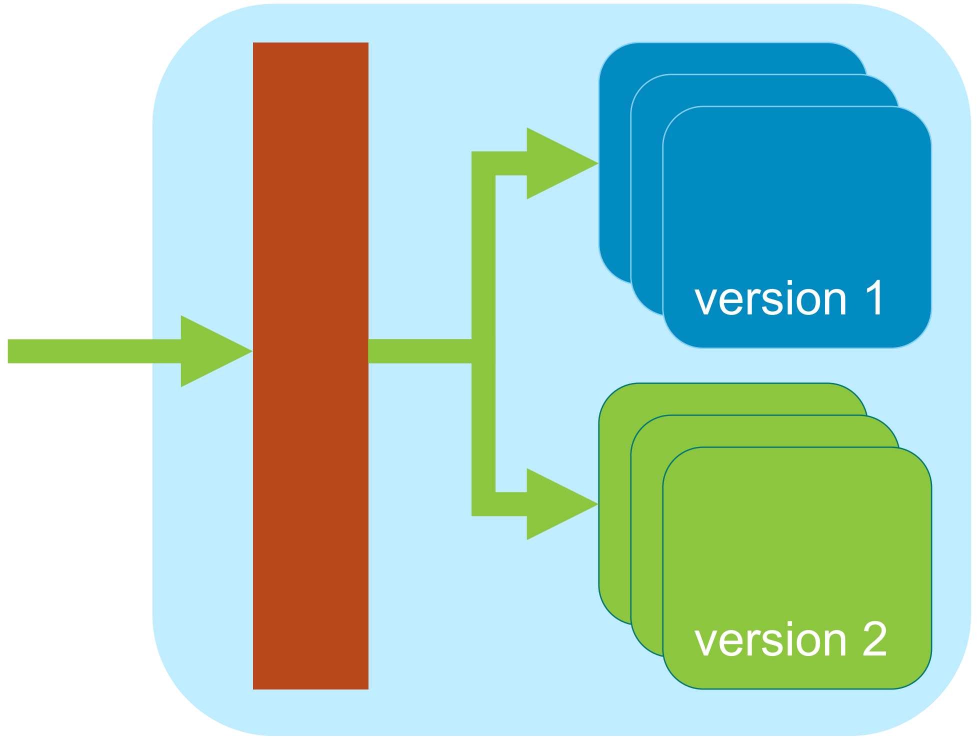
Zero-downtime Deployment

- Deploy a new version without causing an outage
  - Cloud makes this much easier to achieve
  - Two approaches: rolling deployment and blue/green deployment

Zero-downtime Deployment

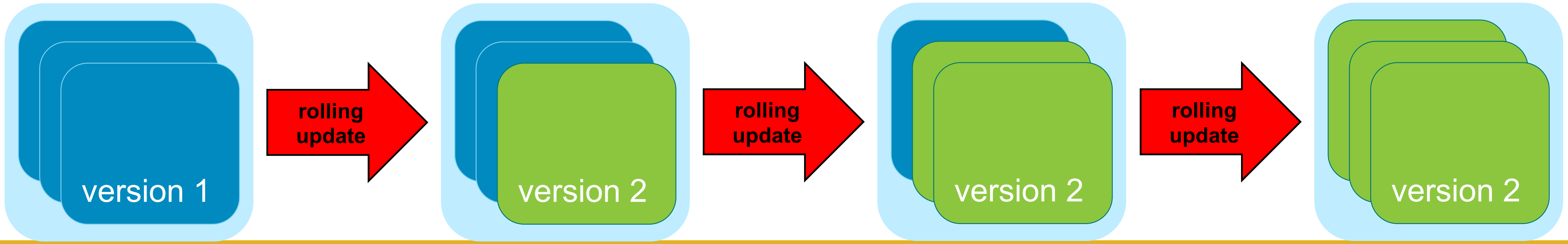
- Deploy a new version without causing an outage
  - Cloud makes this much easier to achieve
  - Two approaches
- Blue/green deployment
  - Deploy v2 alongside v1
  - Shift user load from v1 to v2
  - Requires 2x capacity

Blue/Green Deployment



Zero-downtime Deployment

- Deploy a new version without causing an outage
  - Cloud makes this much easier to achieve
  - Two approaches
- Blue/green deployment
  - Deploy v2 alongside v1
  - Shift user load from v1 to v2
  - Requires 2x capacity
- Rolling deployment
  - Replace instances of v1 with v2
  - Requires 1x capacity



X. Dev/Prod Parity

- Keep development, staging and production as similar as possible.
- Minimize time gap, personnel gap and tools gap.

- **How?**
- **Time gap:** Docker supports delivering code to production faster by enabling automation and reducing bugs caused by environmental drift.
- **Personnel gap:** Dockerfile is the point of collaboration between devs and ops
- **Tools gap:** Docker makes it very easy to spin up production resources locally by using `docker run …`

XI. Logs

- Treat logs as event streams

- **How?**
- Write logs to stdout (Docker does by default)
- Centralizes logs using ELK or [your tool stack here]

  Don’t write logs to disk!

  Don’t retroactively inspect logs! Use ELK to get search, alerts

  Don’t throw out logs! Save data and make data driven decisions

XII. Admin Processes

- Run admin/management tasks as one-off processes.
- Don’t treat them as special processes
- **How?**
- Follow 12-factor for your admin processes (as much as applicable)
- Option to collocate in same source code repo if tightly coupled to another app
- “Enter” namespaces to run one-off commands via `docker exec …`

Administrative processes: Scripting

- Create an SQL database
  - DDL script that creates the schema
  - SQL script that populates initial data
  - Script that runs these scripts as part of creating the database
- Migrate data to a new schema
- Software-defined data center (SDDC)
- Deployment scripts as part of a CI/CD pipeline
  - Creates service instances, deploys runtimes, and binds them

- Scripts are repeatable
  - Test scripts in stage environment
  - Scripts run the same in production
- Store scripts in software configuration management (SCM) system

Q&A

Thanks

for your attention!

Questions?

Federico Accetta federico_accetta@it.ibm.com