Introduction to DevOps

DevOps

- Combination of word development and operations
- It is a practice where collaboration between different disciplines of software development is encouraged.
- Automating software development and deployment processes
  - Reduce time and enhance product quality

DevOps

- Core goal of DevOps is automation and Continuous Delivery.
- Automating repetitive and tedious tasks
  - Leaves more time for human interaction, where true value can be created.
- Accelerate the delivery of work from Development to Operations to our customers

How is it historically done?

- Typical divide between development and operations
- Waterfall processes
- Monolithic applications
- Lack of agility
- Operational inefficiencies

Agile

- Simplify the development process
  - Heavy (waterfall) to light
- Deliver working software frequently
  - Emphasize smaller batch sizes
  - Smaller and well coordinated teams.
- Credited with increasing the productivity
- DevOps finds its roots from Agile

Core business values of DevOps

- Faster Releases
  - Quickly align with business requirements
  - Increase accuracy of releases - avoid downtime
- Save Money
  - Automate manual processes to reduce OPEX
  - Prevent human error and reducing downtime
- Focus on Business
  - Allow high value employees to focus on higher value activities

Seminal

- Allspaw and Paul Hammond gave the seminal presentation:

  “10 Deploys per Day: Dev and Ops Cooperation at Flickr”,

  - *Where they described how they created shared goals between Dev and Ops and used continuous integration practices to make deployment part of everyone’s daily work.*

What does it all mean?

- Traditionally, enterprises have delivered software applications in silos.
- Both process and technology needed to remove the silos
- DevOps
  - Automate Provisioning
  - Automated Build & Deploy
  - Automated Testing
  - Continuous Feedback
- Smaller more frequent releases with tighter coordination

Microservices

What is Microservice

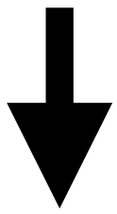
- Minimal function services that are deployed separately but can interact together to achieve a broader use-case
- A small application that does one thing only, and does that one thing well.
- A small component that is easily replaceable, independently developed, and independently deployable.
- Each Microservice Often Has Its Own Datastore
- Organized Around Business Capabilities
- Choice of Technology for Each Microservice
    - REST Calls Over HTTP, Messaging, or Binary

Microservices Are Analogous to Unix Utilities

- Unix Executable: Does one thing and does it well
- Runs independent of other commands
- Produces text-based response
- “*Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*” — **Inventor of Unix Pipe**

Unix & Microservice

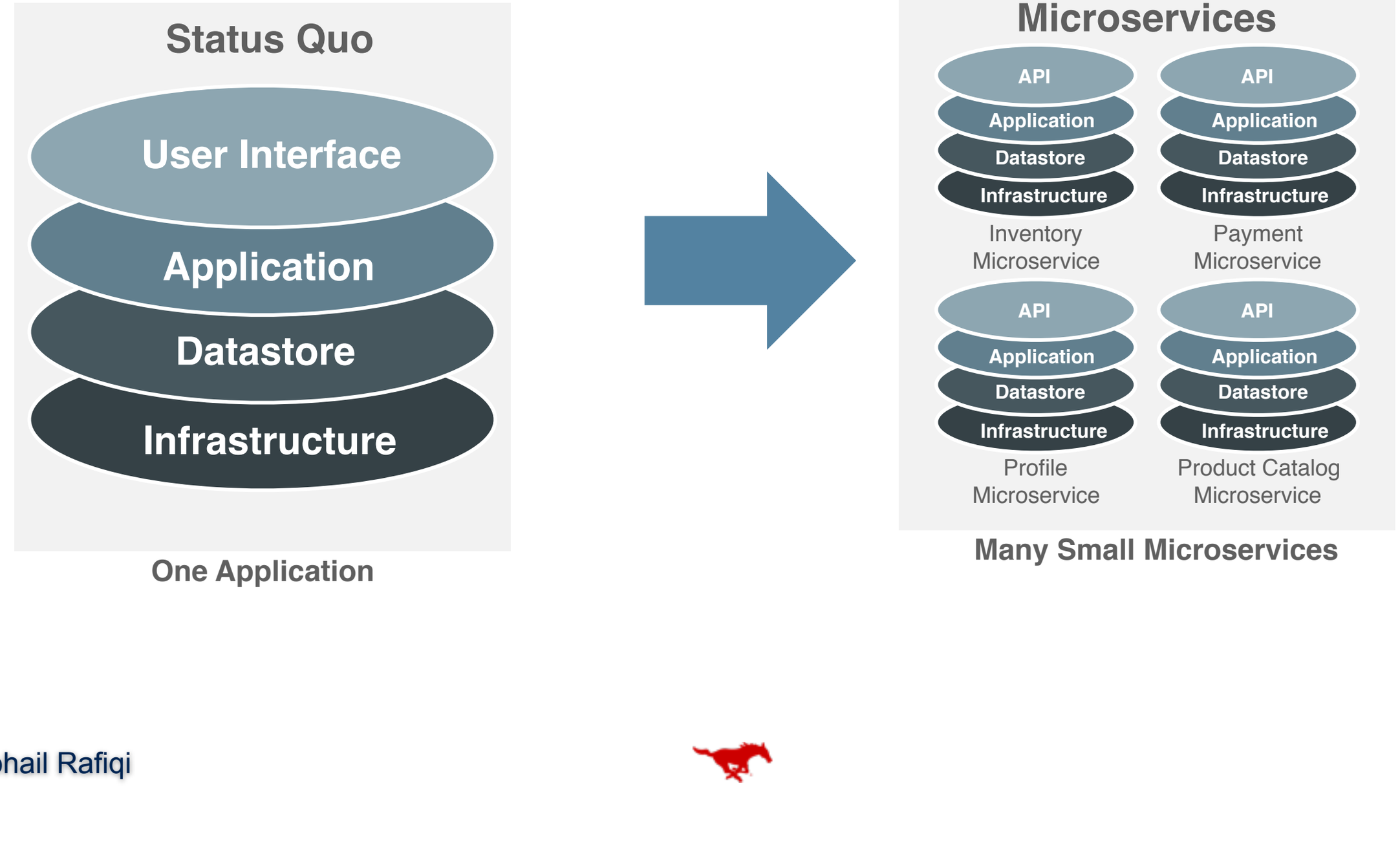
- Unix Executable: Does one thing and does it well
- Runs independent of other commands
- Produces text-based response



curl -v -H "Accept: application/json" -H "Content-type: application/json" -X POST -d '{"productId":645887","quantity":"1"}' "http://localhost:8840/rest/ShoppingCart/"

- Microservice: Does one thing and does it well
- Runs independent of other microservices
- Produces text-based response to clients

Microservices Apps Are Developed/Deployed Independently



Fundamentally, Microservices is a Tradeoff

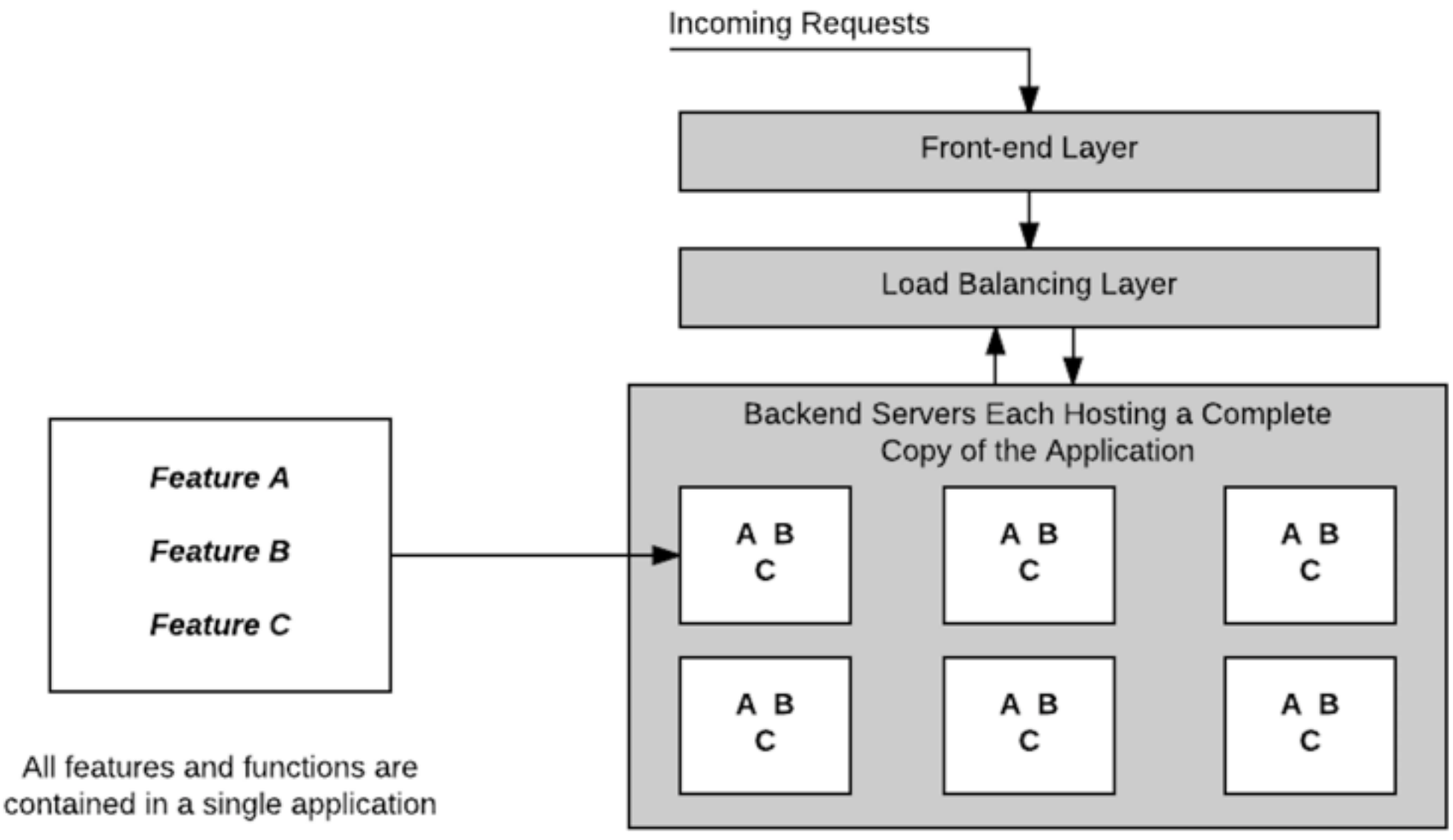
Traditional App Development

- One big block of code, sometimes broken into semi-porous modules
- Complexity handled inside the big block of code
- Each big block is hard to develop but easy to deploy

Microservices

- Many small blocks of code, each developed and deployed independently
- Complexity encapsulated in each microservice
- Each microservice is easy to develop but hard to deploy

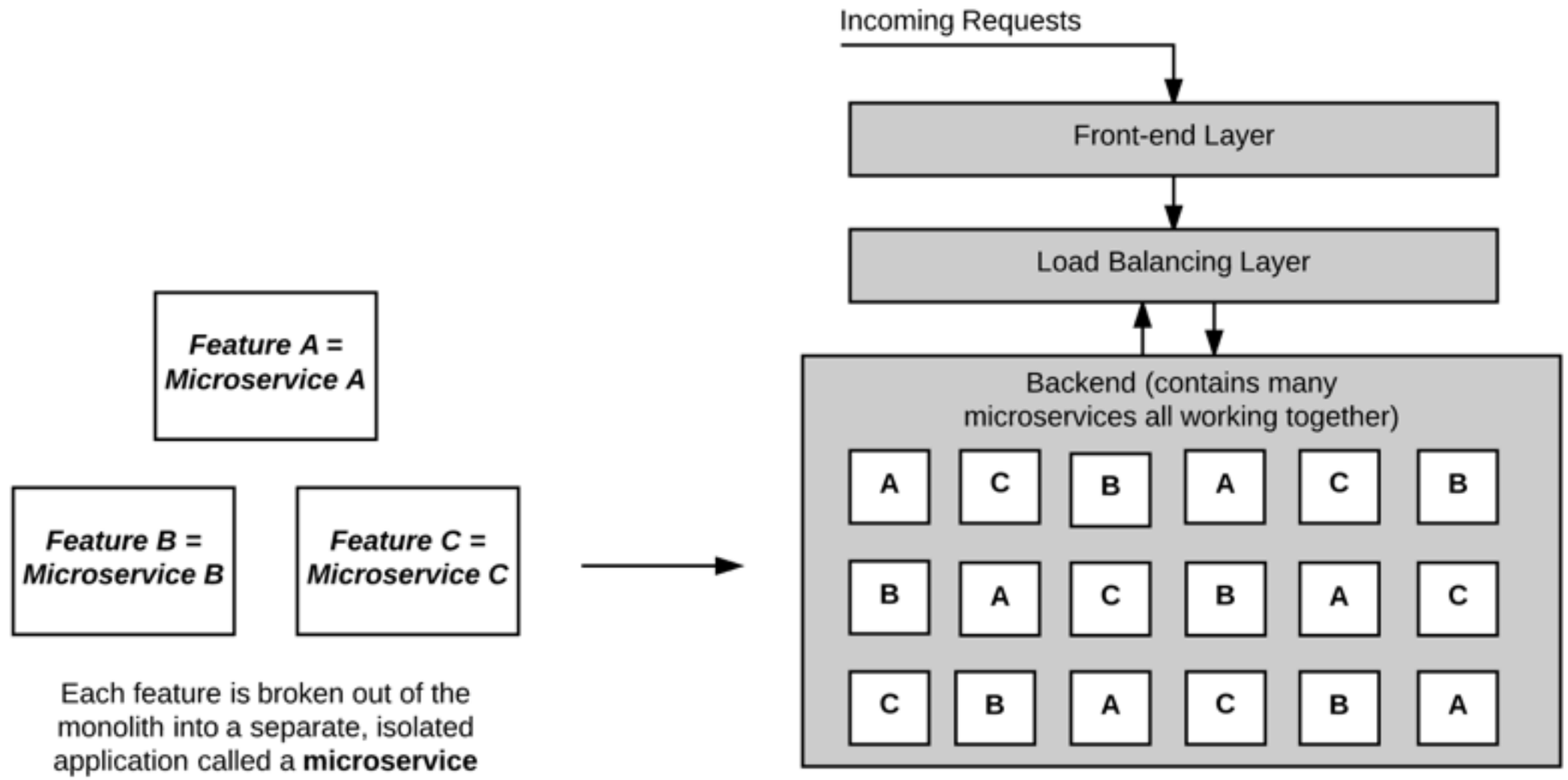
Monolith Application



Issues with Traditional Approach

- Complexity grows as new features are added
- Large number of tests have to be written to test even small changes
  - New functionality
  - Regression
- Operations become difficult
  - Deployment and
  - Support become very challenging

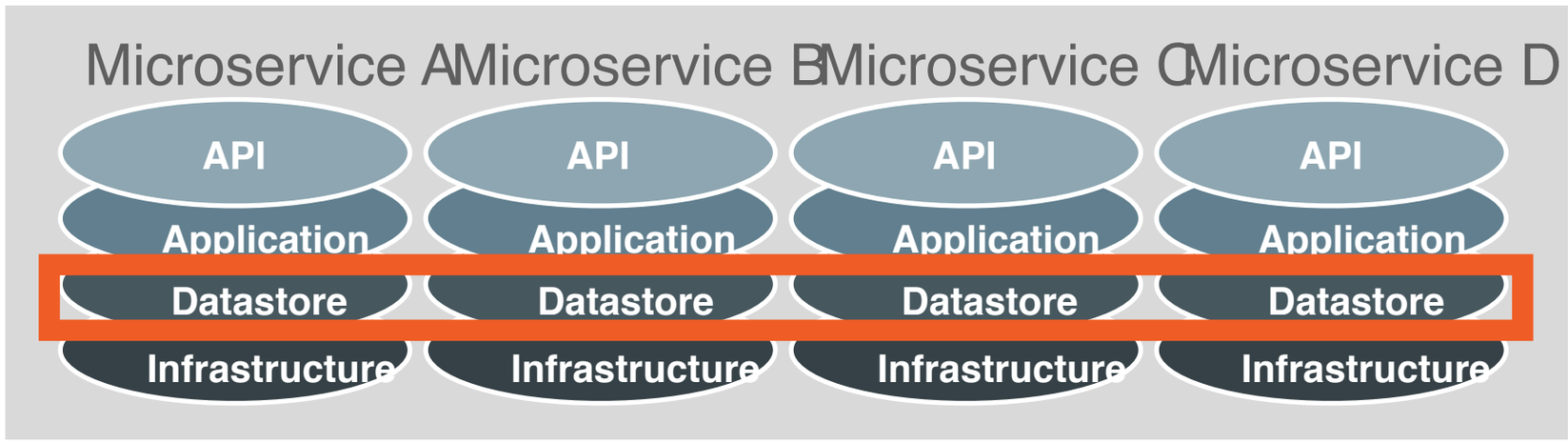
Microservice



Splitting Into Microservices

- Require great deal of architectural effort
- Careful deliberation to successfully break up into micro services
    - Reorganize and restructure code and teams
- First step
    - Identify components that should be written as independent services
    - Understand overall functionality and identify
- Second — Assign several services to one team.
- Third — Necessary horizontal and/or vertical scaling application

Microservices Forces Move To Distributed Computing

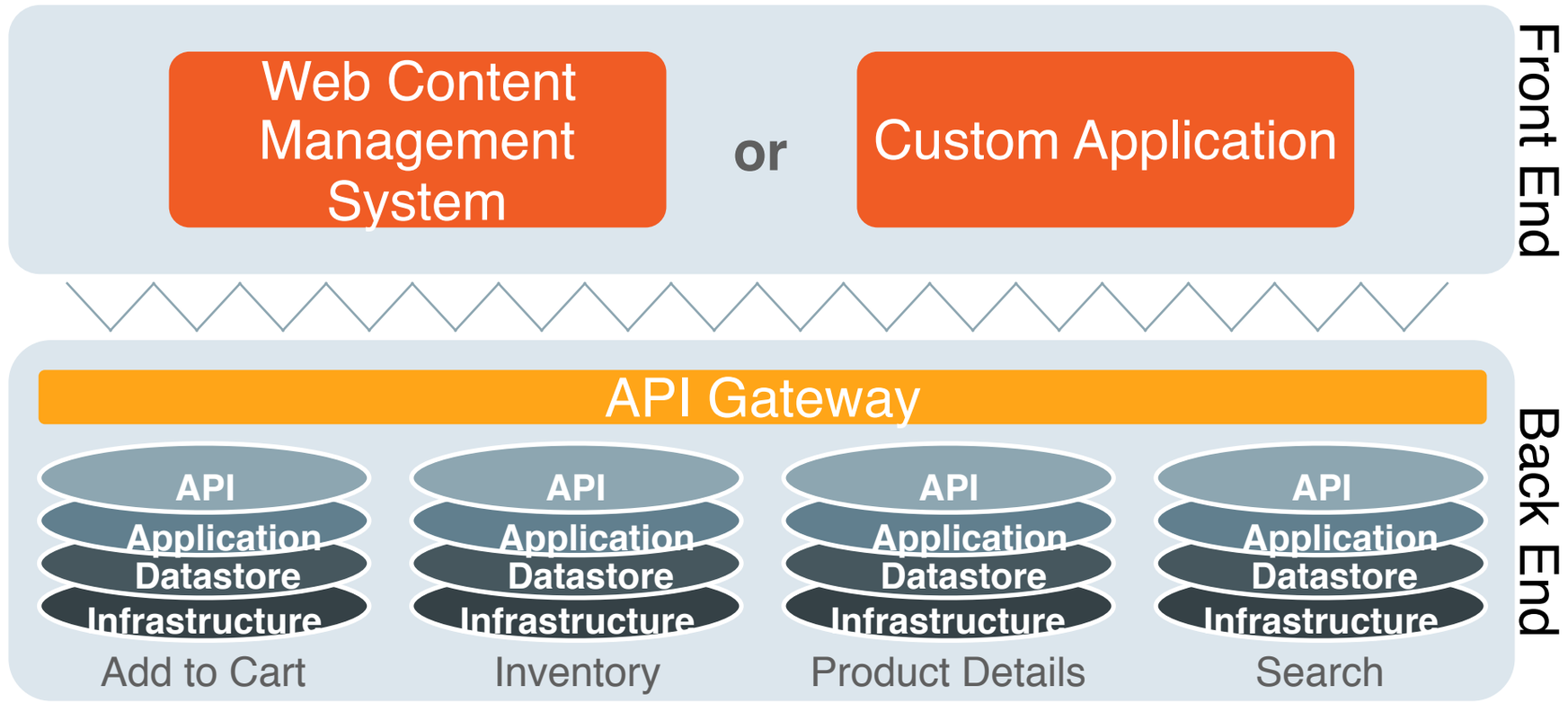


- Distributed computing is a natural consequence of microservices because each microservice has its own datastore
- Sharing datastores across microservices introduces coupling – very bad!
- There will always be latency between microservices
- Latency = eventual consistency
- All data exchange between microservices must be through API layer or messaging – no accessing datastores cross-microservices
- Must implement high-speed messaging between microservices. REST + HTTP probably isn't fast enough
- May end up duplicating data across datastores – e.g. a customer's profile

Separate Front End and Back End

Very Different Requirements

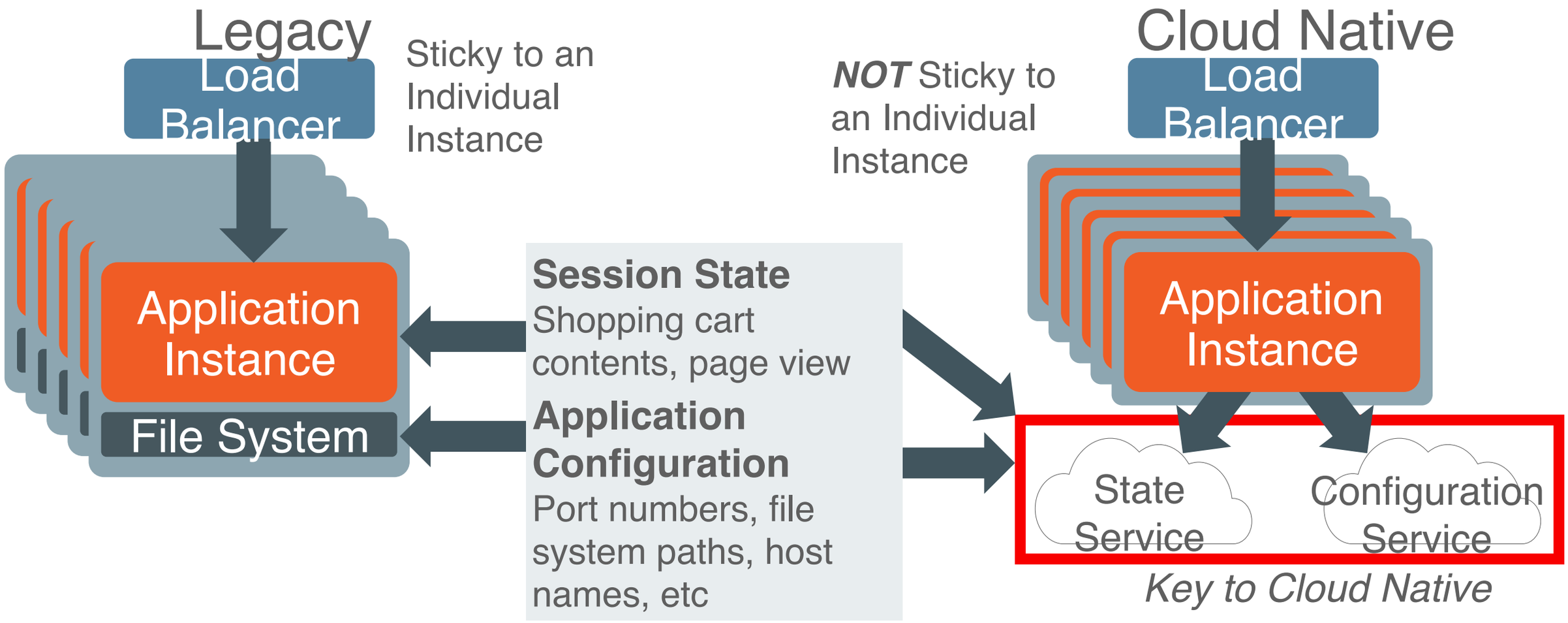
- Security
- Elasticity
- Performance
- Traffic patterns



Design, develop, deploy and manage your front and back ends differently

Sohail Rafiqi

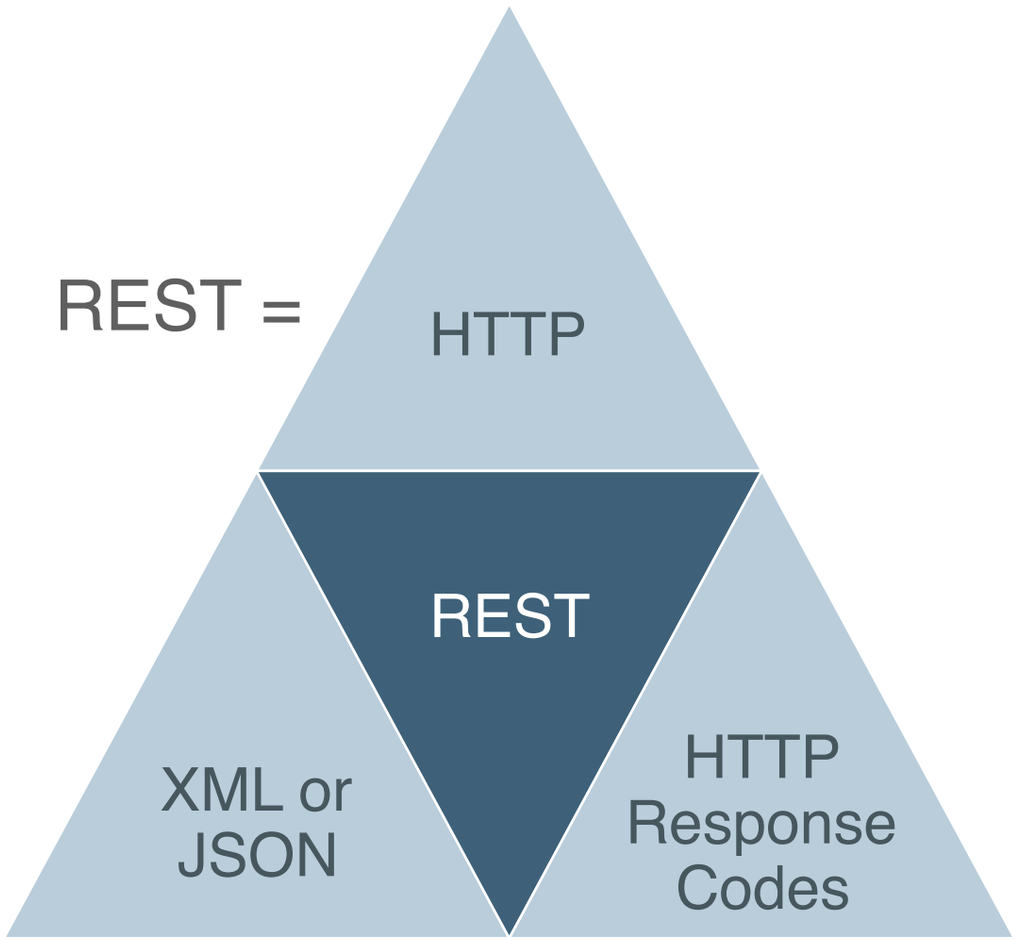
Make Your Middle Tier Stateless If You Can



Approaches to Synchronous Network Calls

	XML/JSON Over HTTP	Binary Over Wire
Primary Use	Communicating with clients over the public internet	Communicating with other microservices over a private network
Pros	▪ Universally understood format ▪ Easy to implement and understand	▪ Very fast
Cons	▪ Slow since it's text-based	▪ Can be hard to implement
Implementations	▪ No special software required – natively supported by all major programming languages ▪ HTTP is the language of the web!	▪ Oracle Portable Object Format ▪ Google Protocol Buffers ▪ Apache Avro ▪ Apache Thrift

REST: Representational State Transfer

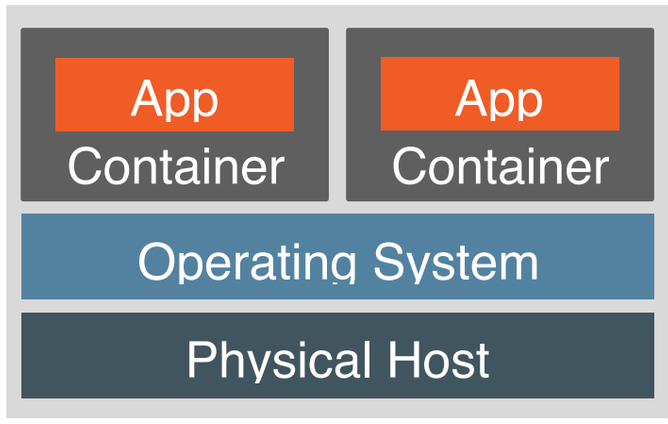


- Much simpler alternative to SOAP
- Uses GET, POST, PUT, DELETE, etc – just like web browsers do
- Synchronous inter-microservice communication often occurs over binary
- Can version APIs - /v1.2/customer
- Can use XML or JSON
  - XML is often better - supports XPath, CSS selectors
- Can't generate strongly typed stubs

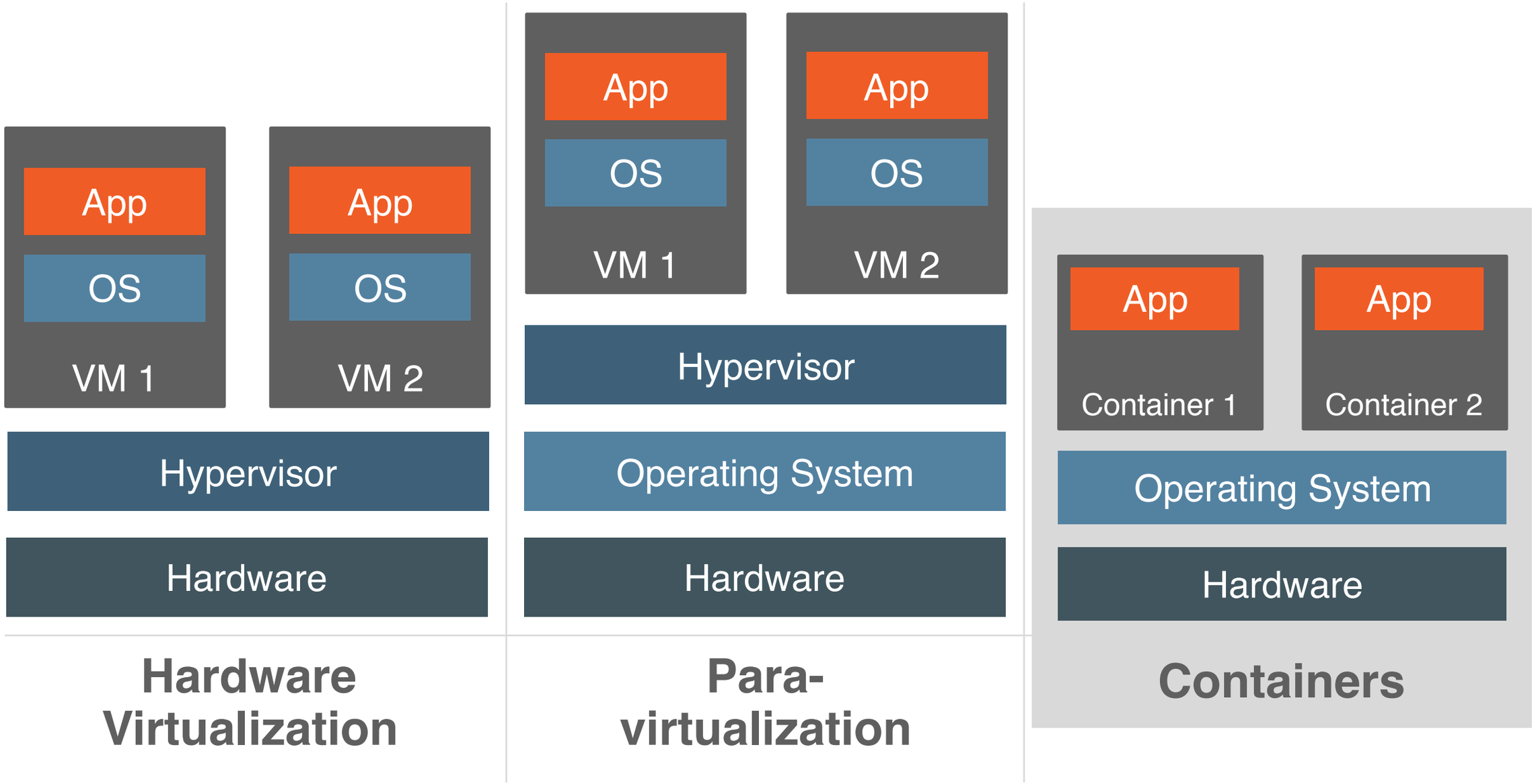
Docker Containers

What is a Container

- A self-contained sealed unit of software
- Contains everything required to run the code
- Includes OS
- A container includes:
  - Code
  - Config
  - Processes
  - Networking



App Virtualization



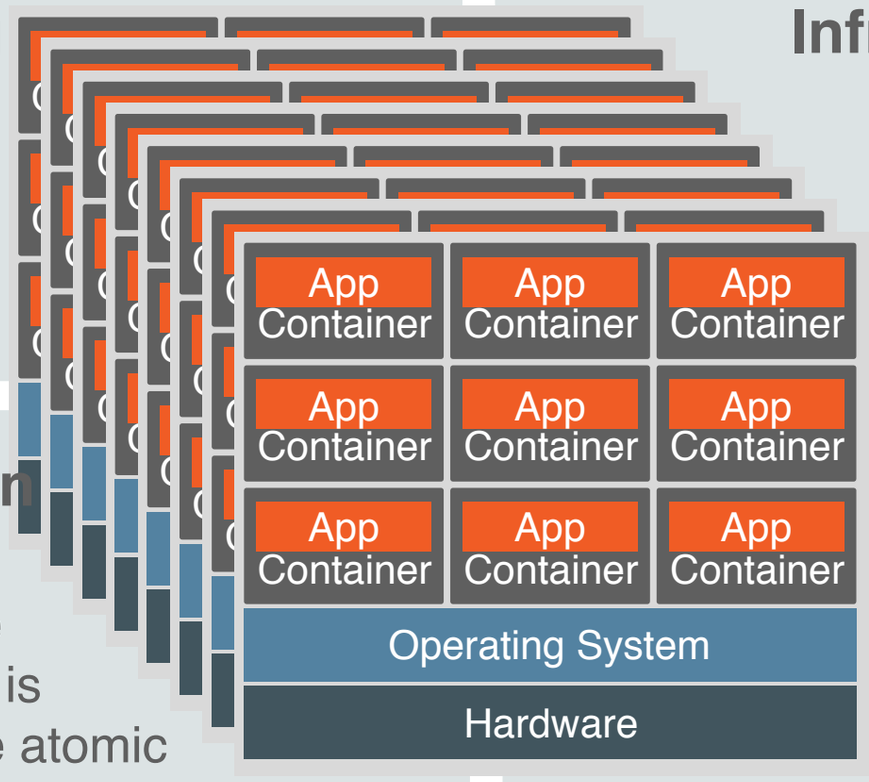
Containers make microservices easier

- #1 value – app packaging
- Microservices doesn't rely on containers but they do help:
  - Higher density
  - Easy to start/stop
  - Portability
- Containers are lightweight, just like microservices themselves

Use Cases

Application Packaging

Neatly package applications and supporting environment in immutable, portable containers



Infrastructure Consolidation

Get infrastructure utilization up to 100% (vs 5-10% with VMs) due to over-subscription of resources and near bare metal performance.

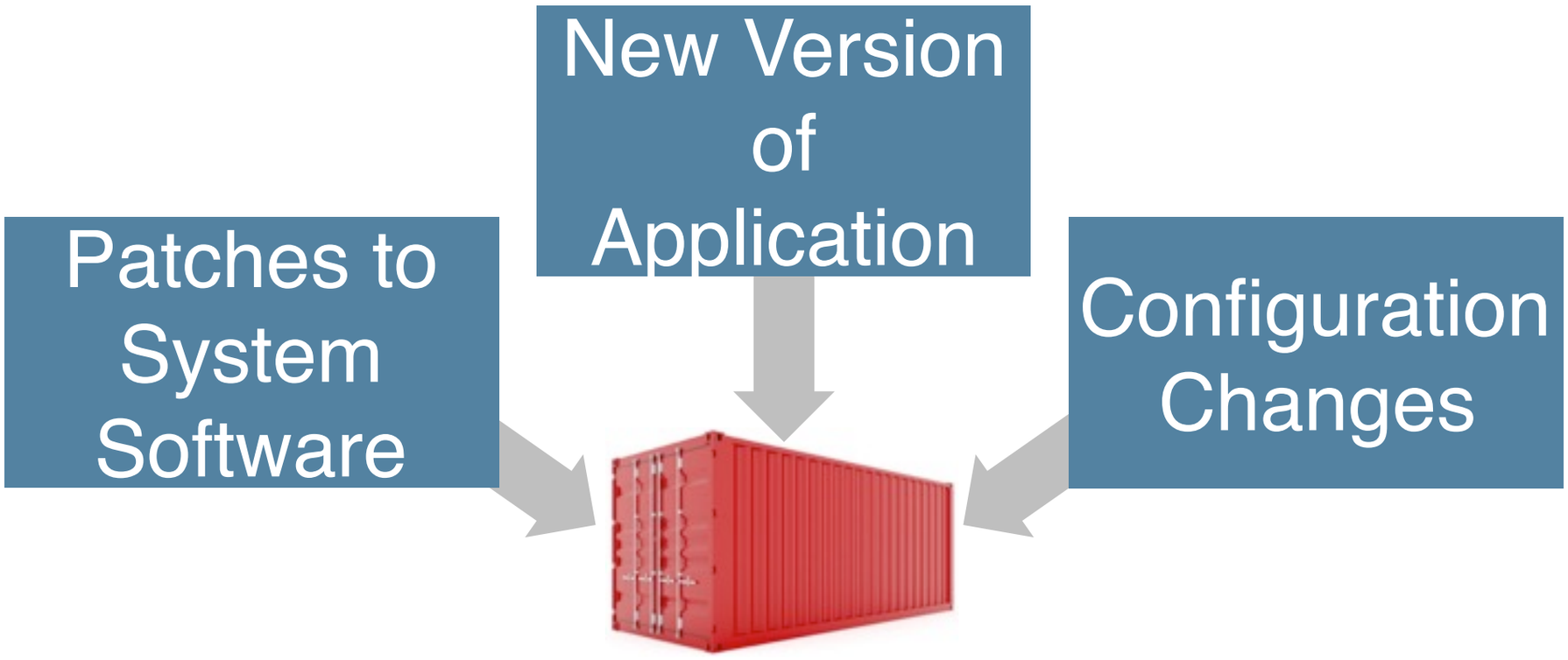
Continuous Integration

All changes to an app are contained in one immutable container image. Container is tested and deployed as one atomic unit

DIY PaaS

Build a simple PaaS by wiring up containers to a load balancer. New code, patches, etc pushed as new immutable containers.

Containers Should Be Immutable

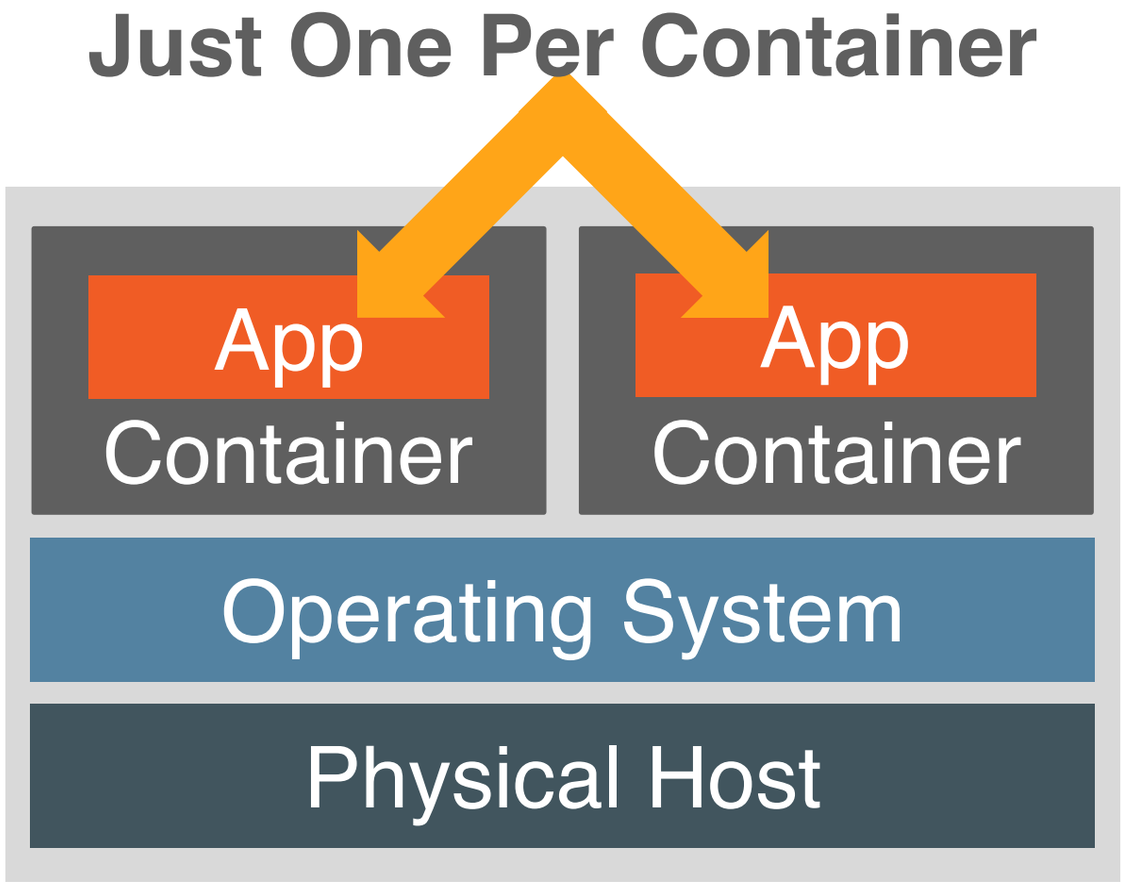


Build and deploy a new container

Never touch a container that's already been built

One Instance Per Container is Typical

- Best to run one instance (unique host/port combination) per container
- Running multiple instances of the same application or different applications will make scheduling very difficult
- Expose one port per container



Docker

- Client & Server program
- Builds container from code
- Takes your code along with its dependencies and bundles it up into a container
- Docker carves up a running linux system into small containers
  - Each is complete (all code, config, contains all processes)
  - Has all the dependencies that systems needs
  - Enough OS
- Designed to be portable
  - Can easily be shipped from one place to another

What docker does

- Begins with an Image
- Image makes up just enough of the OS to do what you need to do
    - Traditionally you will install a whole OS with everything for each apps you do
    - Docker pairs it way down so that you have a little container with enough OS
- docker images
- docker run

Looking at Container Output

- docker logs
  - Keeps the output of containers
  - View with:
    - docker logs container_name

Stopping and Removing Containers

- Killing and removing containers
  - docker kill container_name
  - docker rm container_name

Private Container Networking

- Programs in containers are isolated from the internet by default
- You can group your containers into “private” networks
- You explicitly choose who can connect to whom

Dockerfile

- Build images with code
- This is a small program to create an image
- Run this program with:

  docker build -t name-of-result .
- When it finishes, the result will be in local docker registry

Dockerfile

- Docker files looks like shell scripts
  - But they are not
- Processes you start on one line will not be running on the next line
- Environment variables you set will be set on the next line
  - If you use the ENV command, remember that each line is its own call to docker run

Example Dockerfile

```
# Install a more up to date mongodb than what is included in the default ubuntu repositories.


FROM ubuntu

MAINTAINER Kimbro Staken

RUN apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10

RUN echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen" | tee -a /etc/apt/
sources.list.d/10gen.list

RUN apt-get update

RUN apt-get -y install apt-utils

RUN apt-get -y install mongodb-10gen

#RUN echo "" >> /etc/mongodb.conf

CMD ["/usr/bin/mongod", "--config", "/etc/mongodb.conf"]
```