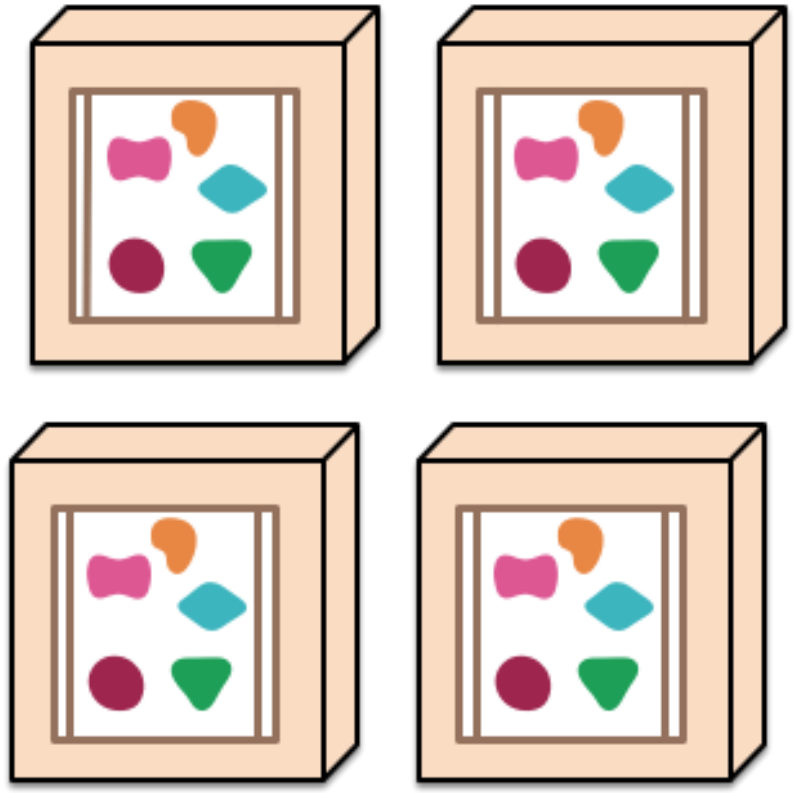

# The 12 Factor App

# Monolithic Applications

- Monolithic application contains all the functionalities in a single application
- Application is scaled by cloning and running the entire application on multiple servers/VM/containers
- Applications typically organized around a service bus
  - Applications are services
  - Bus is the backbone

*A monolithic application puts all its functionality into a single process...*

*... and scales by replicating the monolith on multiple servers*

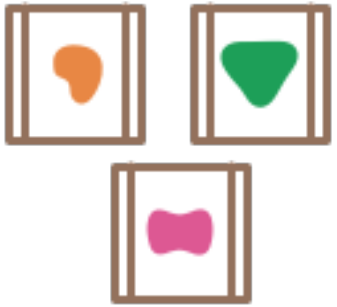


Images from https://martinfowler.com/articles/microservices.html

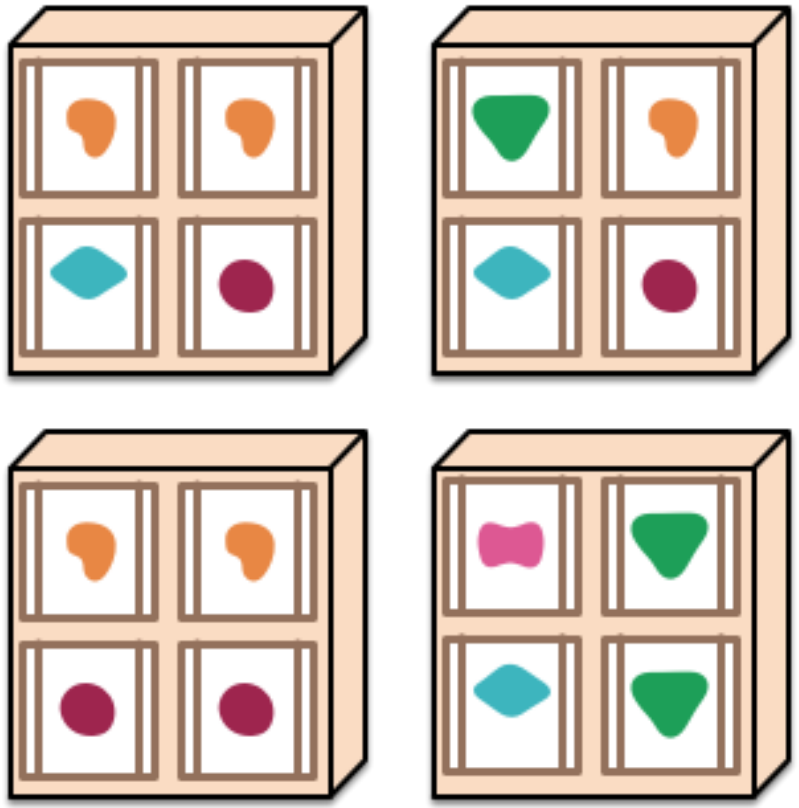
# Microservices

- Functions in an application are separated in to separate smaller services
- Each service is deployed into its own servers/VM/containers
  - Each service own its own data
- Only need to deploy the application's services
- One or more services work together to deliver a business function

*A microservices architecture puts each element of functionality into a separate service...*

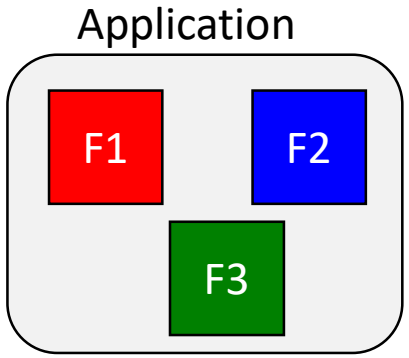


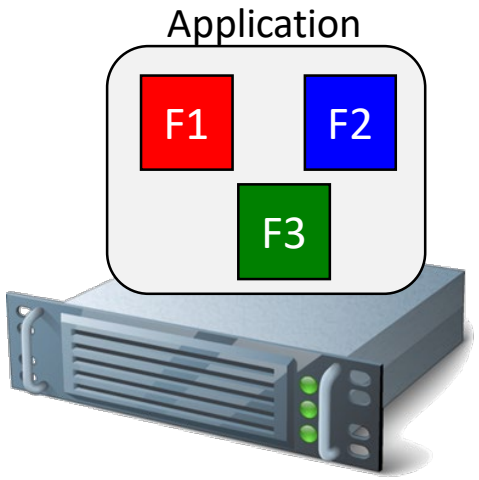
*... and scales by distributing these services across servers, replicating as needed.*

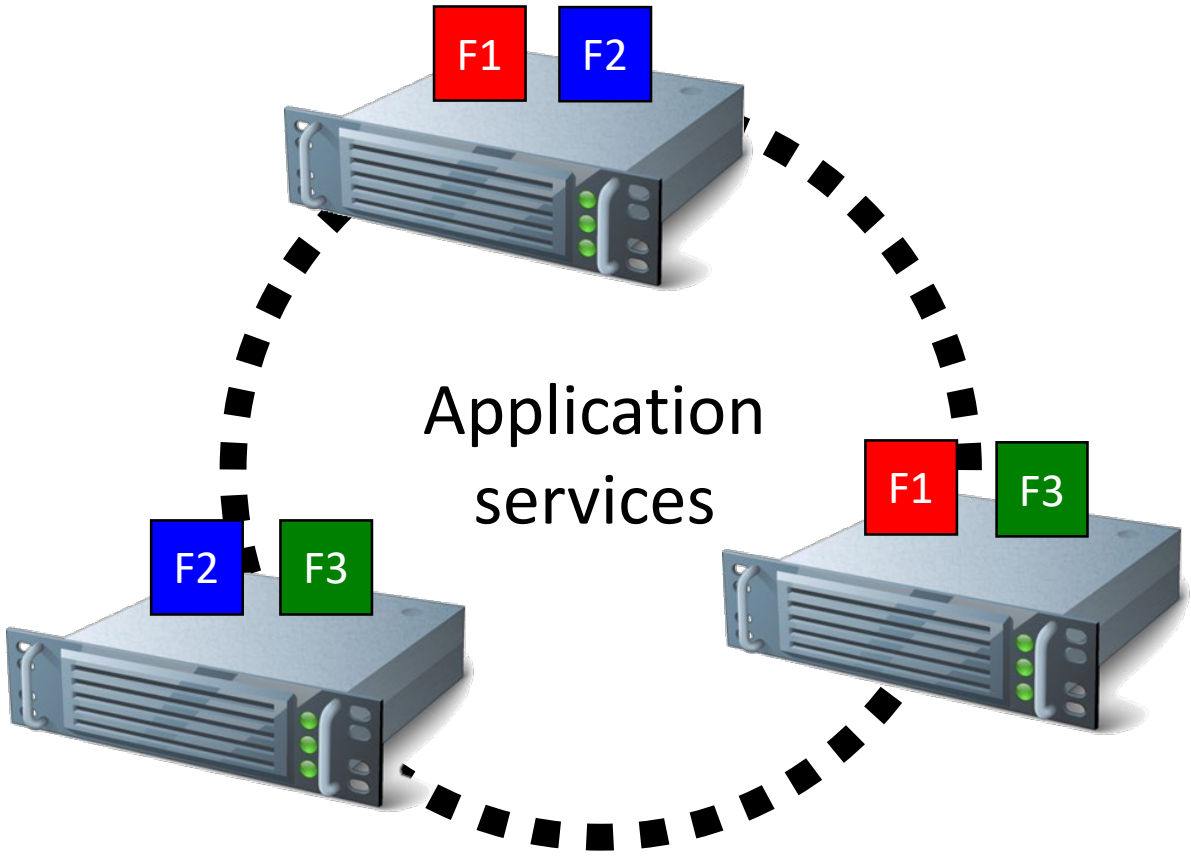


Images from https://martinfowler.com/articles/microservices.html

# Monolithic vs Micro Service



# Refactoring a Monolithic Application

- Breakup applications according to their context
  - Context is dependent on the subject domain
- Service owns and manage the data model and data
  - Bound to the context
- A context cannot update data belonging to another context
- Explicit relationships between contexts/services

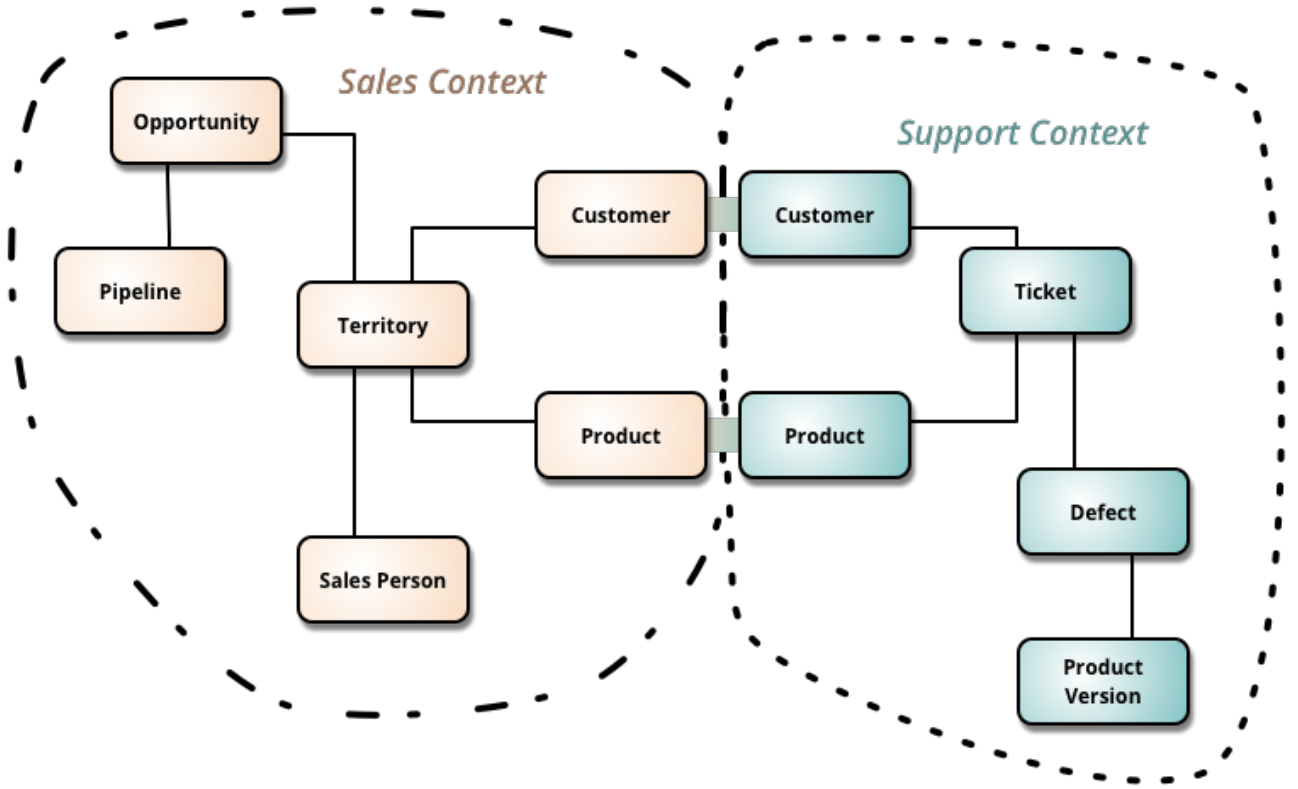
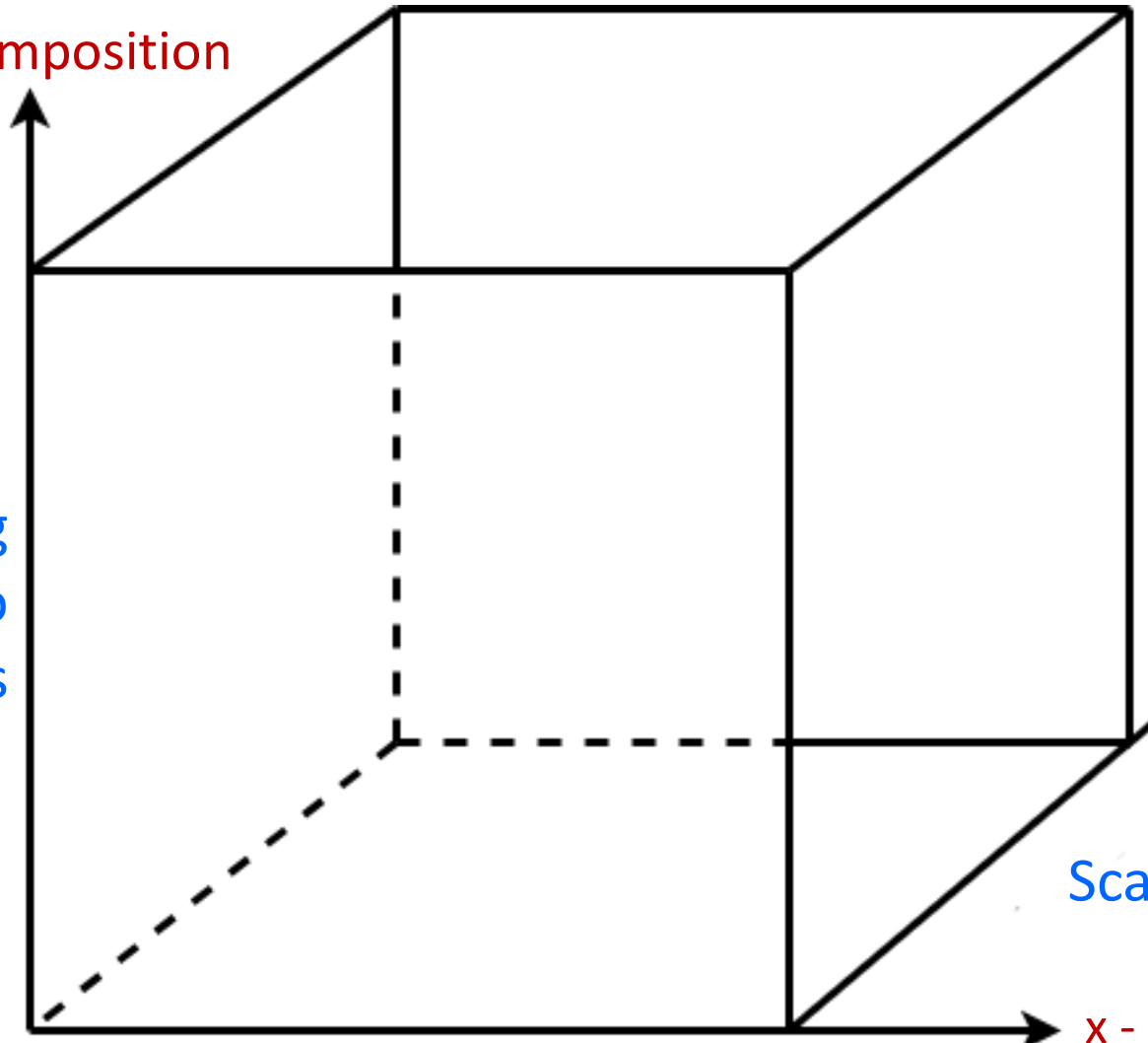



Image from https://martinfowler.com/bliki/BoundedContext.html

# How to Scale?

y - functional decomposition



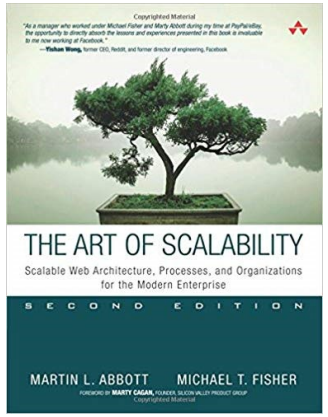
Scale by splitting application into smaller modules

z - data partitioning

Scale by sharding the database

x - horizontal duplication

Scale by cloning the application



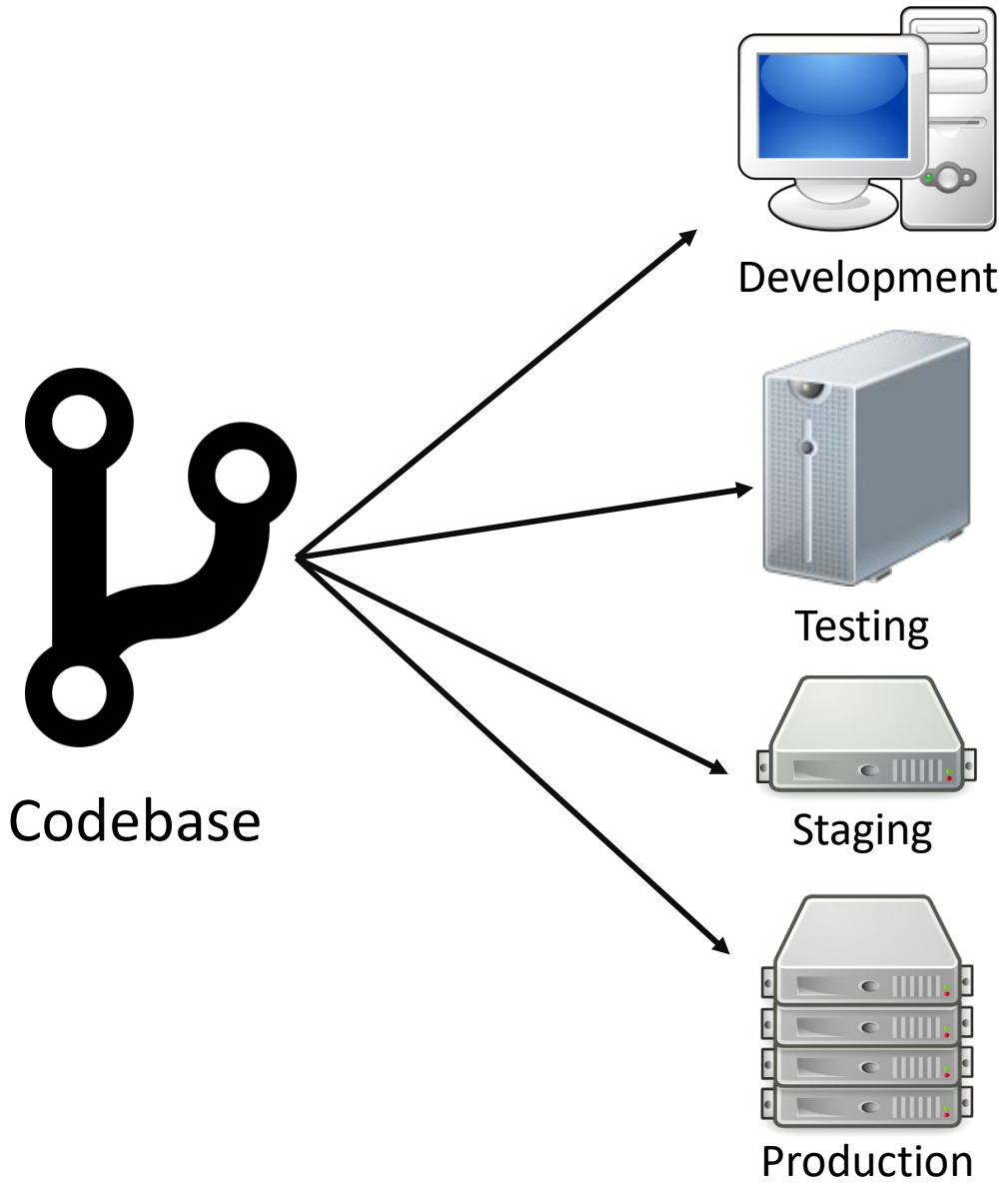
# What is the 12 Factor App?

- Software development methodology for building applications using the microservices approach
  - Drafted by developers in Heroku, presented by Adam Wiggins circa 2011
- Includes best practices to allow application to scale, portable and resilient to failure when deployed to the web
- Consider as part of how to develop a cloud native application
- Most of the ‘factors’ are applicable to popular runtime
  - Python, JavaScript
- Criticism that the methodology is specific to Heroku

# 1. Codebase

## One codebase tracked in revision control, many deploys

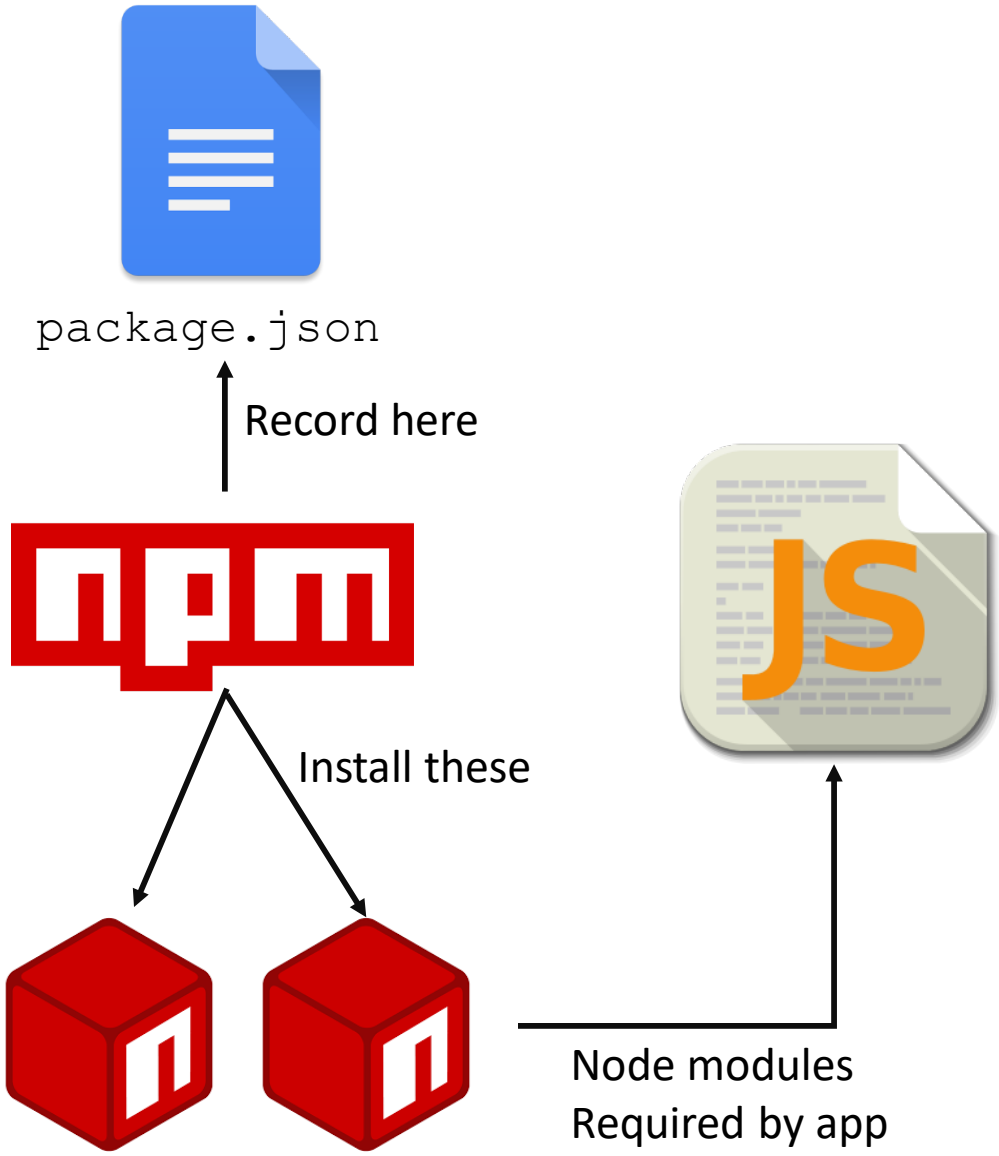
- One codebase for one application
- Tracked by version control
  - Good rule of thumb, 1 repo one codebase



# 2. Dependencies

## Explicitly declare and isolate dependencies

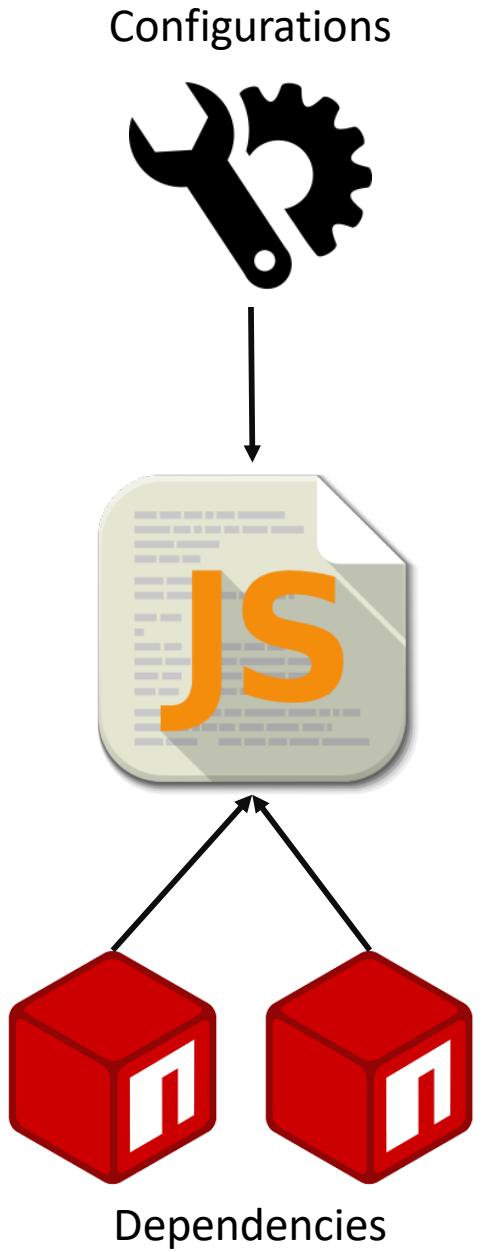
- Most applications depend on libraries
  - During the build – eg. pre-processors
  - For execution – eg. RxJava
- Externalize the dependencies in a manifest
  - Eg. `requirements.txt`, `package.json`, `pom.xml`
- Externalizing dependencies create repeatable builds



# 3. Configurations

## Store configurations in the environment

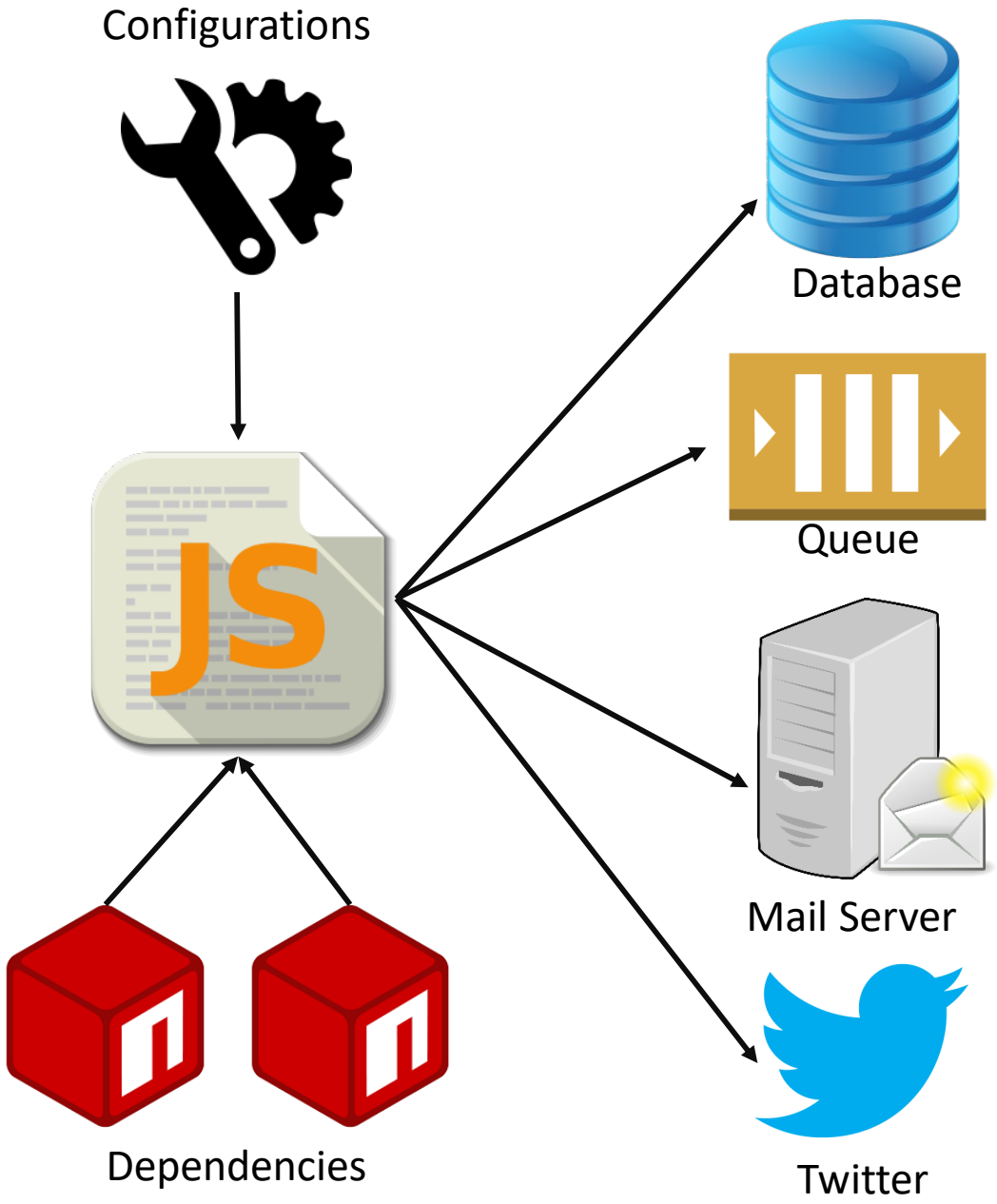
- Configurations are information required by an application to run
  - Eg. credentials, IP address, etc.
  - Vary between deployments
    - Different database for testing and production
  - Eg. Can be set as environment variables, command line arguments
- Configuration enable repeatable deployments



# 4. Backing Services

## Treat backing services as attached resources

- Resources that the application uses as part of its operation
  - Eg. database, queues, authentication, etc
- Connection details for these resources are stored in the configuration



# 5. Build, Release, Run

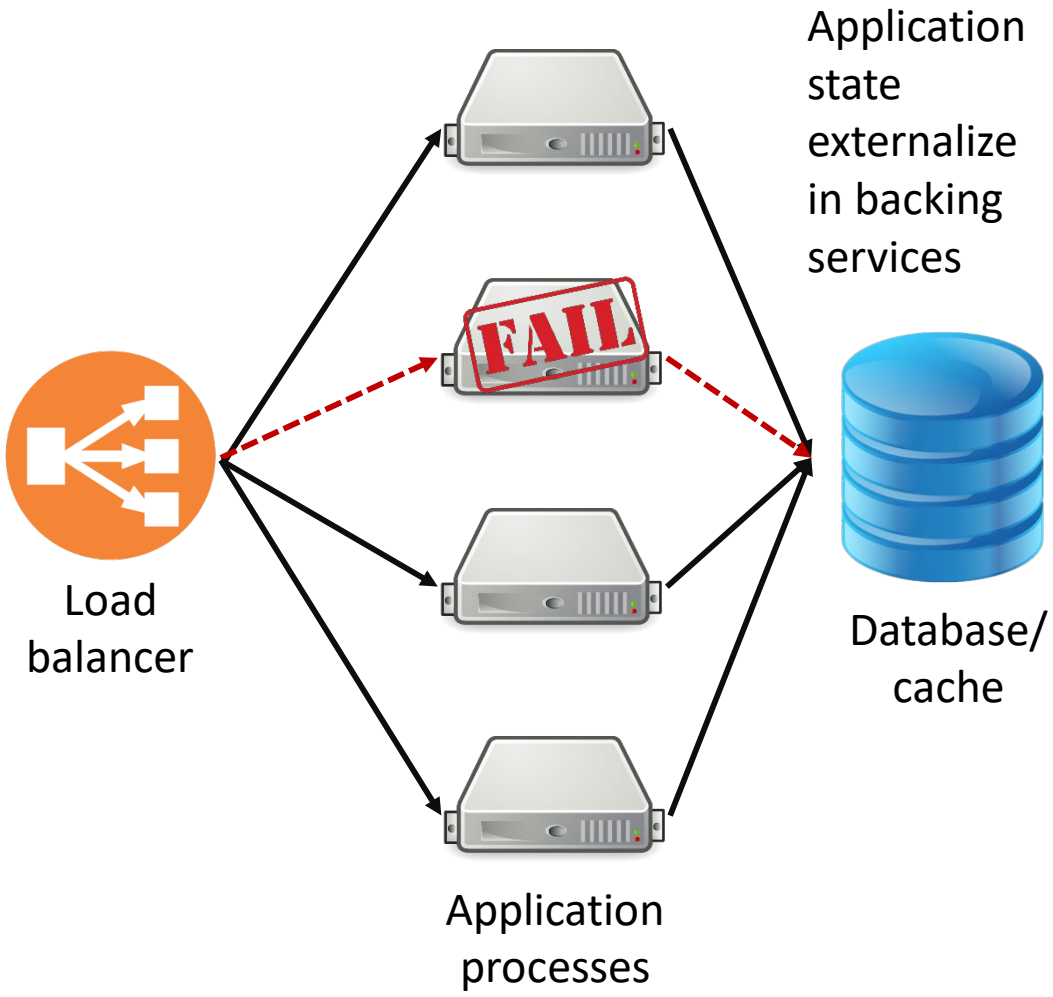
## Strictly separate build and run stages

- Build stage compiles the app along with all the dependencies
  - Produces a binary or some deployment artefact eg. WAR file
- Release stage runs test on the application
  - Application is suitable for release if it passes all test
  - Note: original definition for release is to produce a binary that combines with the configuration
- Run stage executes the binary with the configuration

# 6. Processes

## Execute the app as one or more stateless processes

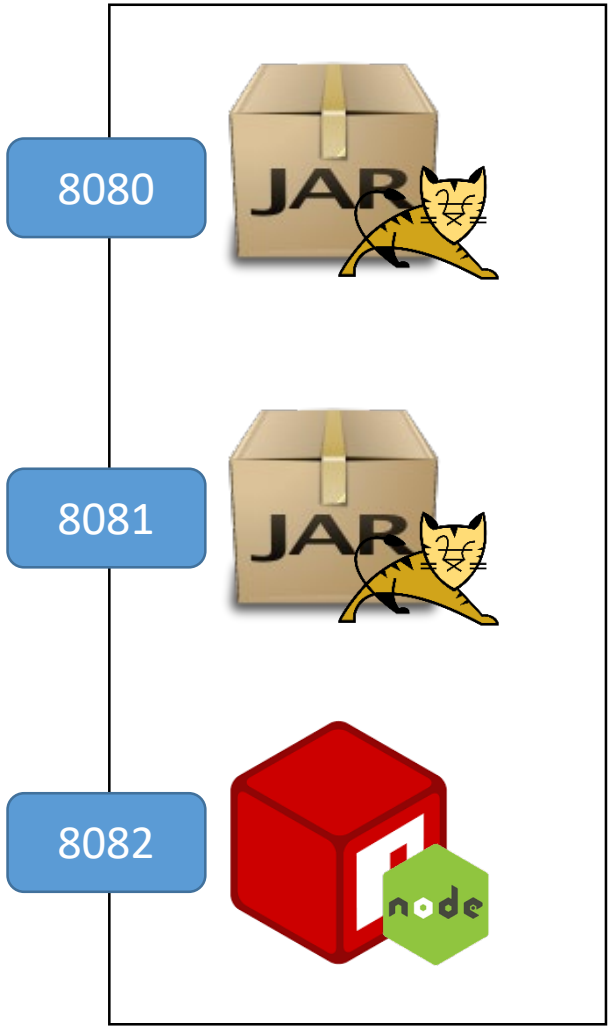
- Stateless app are apps that store nothing in the process' memory or local hard disk
  - Assume local storage is ephemeral
- All data are externalized to backing store
  - Eg. Database, cache
- App instances can fail without affecting the client
  - Client request can get routed to any other process



# 7. Port Binding

## Export services via port binding

- To selectively determine which port an application listens on
  - Eg. HTTP listen to 3000 instead of 80
- Flexibility to by the deployer decide which port an application listens to at deployment/run time
  - Avoid listening to the same port especially for same application



# 8. Concurrency

## Scale out via the process model

- Increase the number of processes as the workload increases
  - Eg. Add more web servers when there are more request
  - Possible only if the process are stateless
- Horizontal scaling / scaling out
- Process can be started either in the same server or in another server
  - For processes in the same server, port binding avoids binding to an already allocated port

Horizontal scaling / scaling out

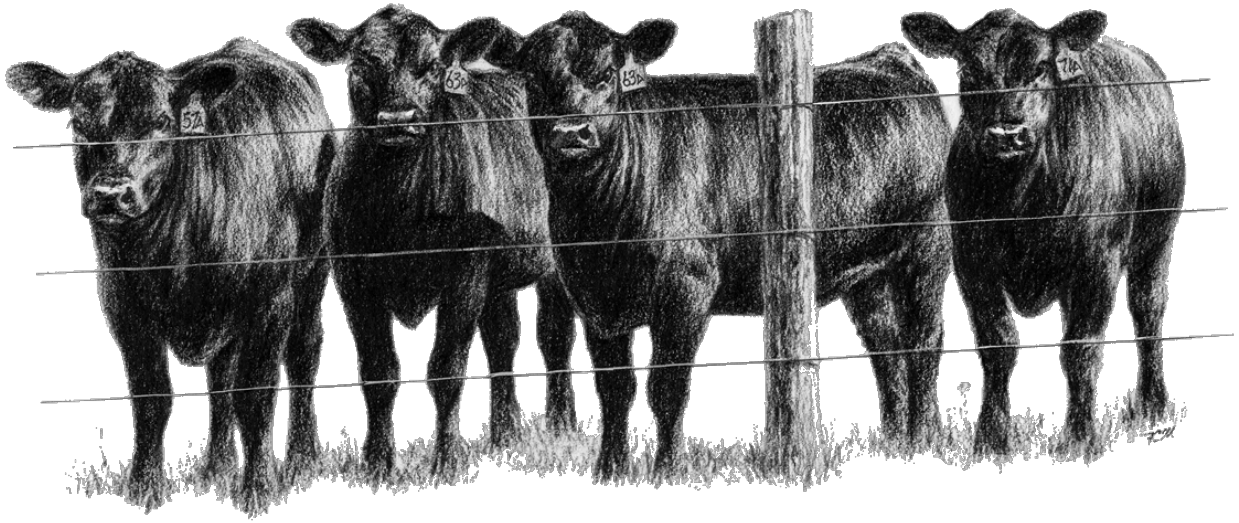

Vertical scaling / scaling up

# 9. Disposability

## Maximize robustness with fast startup and graceful shutdown

- Processes/applications can be started or stopped at anytime
  - Stateless processes
  - Cattle vs Pets
- Ideally startup should be fast
- Shutdown should be graceful
  - Deployer will notify the application when it intends to kill the process
  - Eg. By signal SIGHUP or by invoking a URL

Pet



Cattle

# 10. Dev/Prod Parity

## Keep development, staging and production as similar as possible

- Traditionally great disparity between development environment and deployment
  - Development might be Window, deployment may be Linux
- Keep the 3 environments as similar as possible
  - Minimize errors – building, testing and running
- Use configurations set the backing services

# 11. Logs

## Treat logs as event stream

- Logs provide visibility into what is happening inside an application
- Log meaningful data
  - Can be used for diagnostics, insight into your business
  - Eg. Use Apache log format for request
- Treat it as stream of time ordered event
  - Can be monitored at real-time or routed to data store

# 12. Admin Processes

## Run admin/management task as one-off processes

- Admin process are operations like archive data, delete in active users
- These should be run in the same environment as the long running processes
  - Should be the same codebase/release as the application
  - Using the same configuration
  - Eg. `npm run migrate-data`

# DevOps

**Dev**

- Code base
- Dependencies
- Configurations
- Backing services
- Dev/ops parity
- Build, release, run

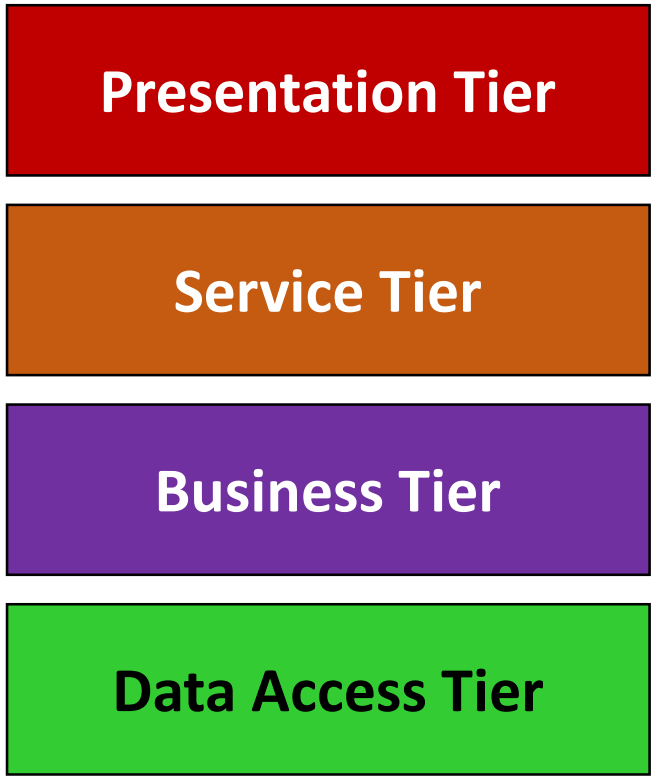
**Ops**

- Processes
- Port binding
- Concurrency
- Disposability
- Dev/ops parity
- Logs
- Admin processes

# Appendix

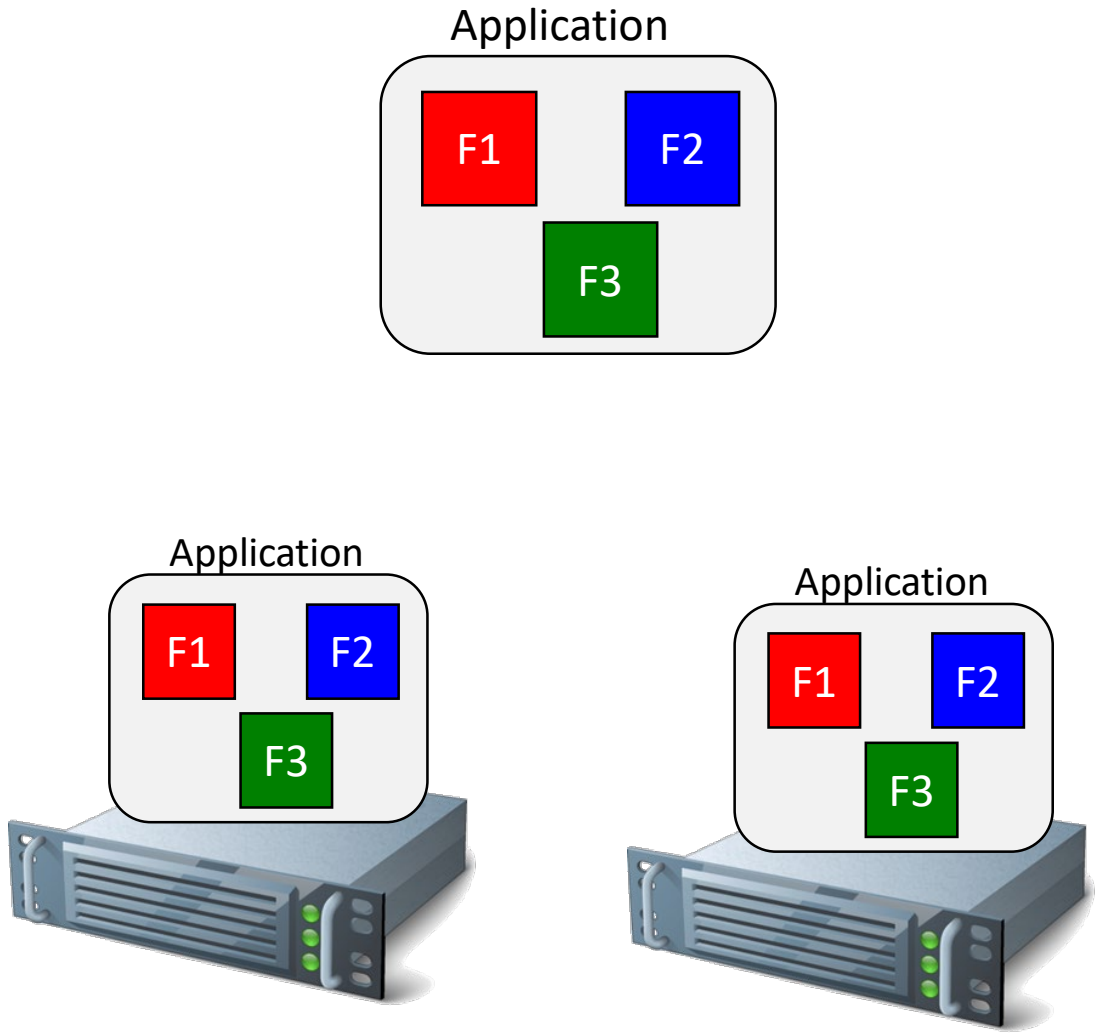
# Traditional Tiered Application

- Application implements all the requirements
- Application is structured around tiers
  - Each tier is responsible for some aspects of the total application
- Tiers are independent of each other logically
  - Coupled at the code
- A single database is shared across all tiers
- Monolithic



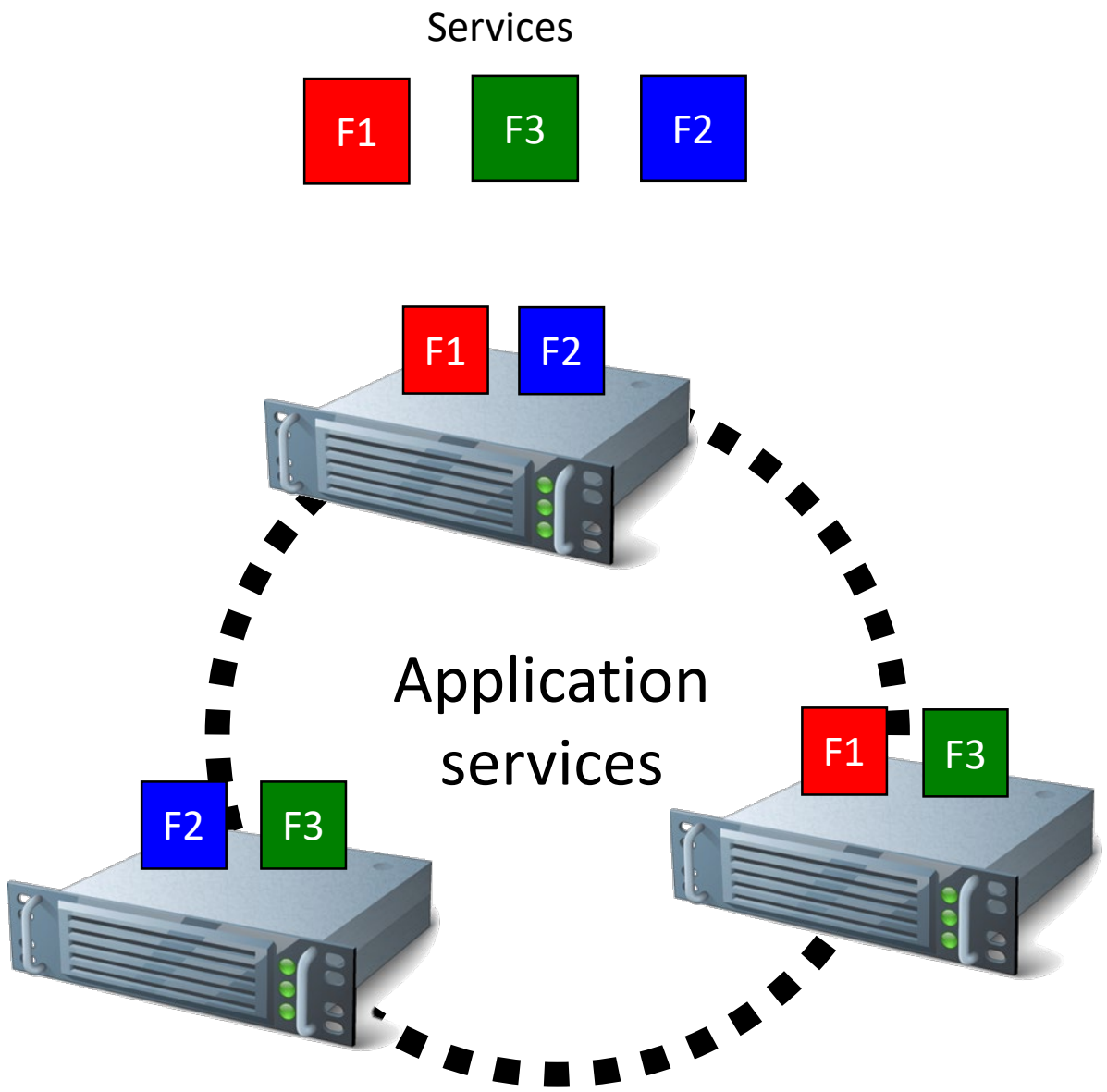
# Monolithic Application

- Monolithic application contains all the functionalities in a single application
- Application is scaled by cloning and running it on multiple different servers/VM/containers



# Microservices Approach

- Functions in an application are separated in to separate smaller services
- Each service is deployed into its own servers/VM/containers
  - Each service own its own data
- Only need to deploy the application's services
- Services work together to deliver the application service



# Service Decomposition

- Loose coupling
  - Changes in one service should not require a change to another service
  - Services should know as little as possible about the service that it is interacting with
- High cohesion
  - Related behaviours to be in the same service

- Bounded context
  - Service owns and is responsible for the data/message

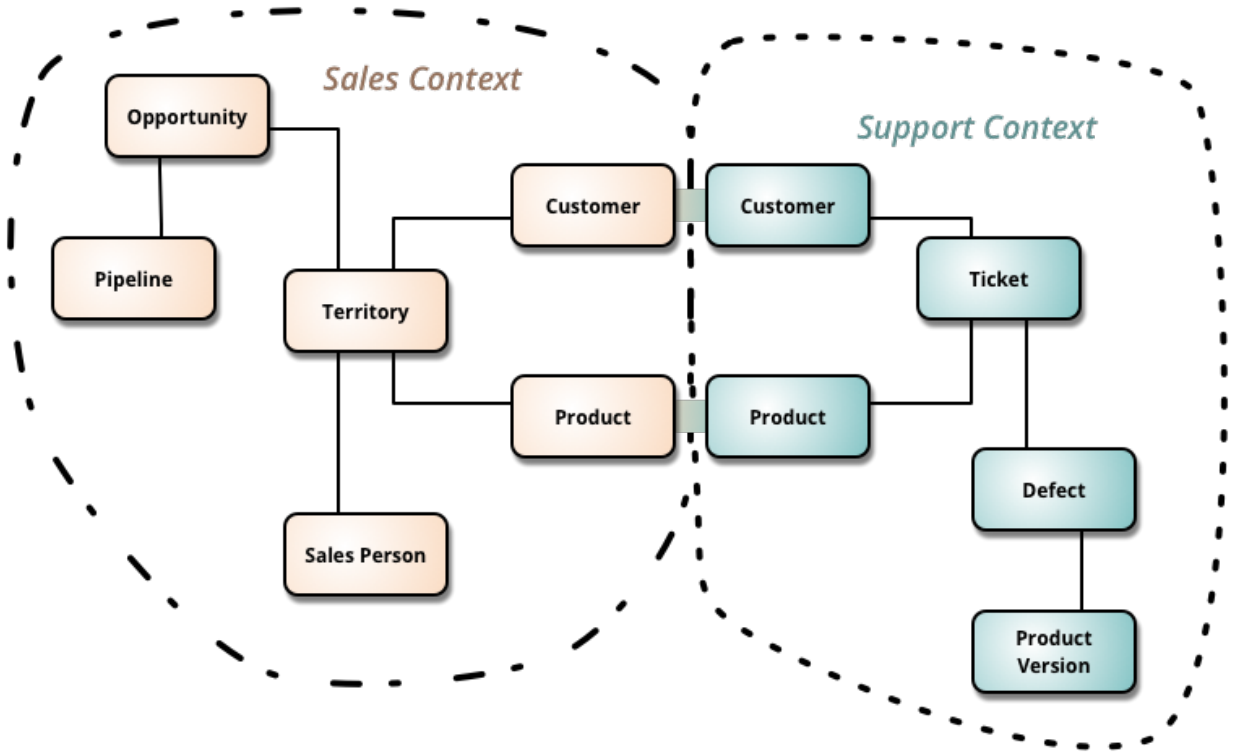



Image from https://martinfowler.com/bliki/BoundedContext.html