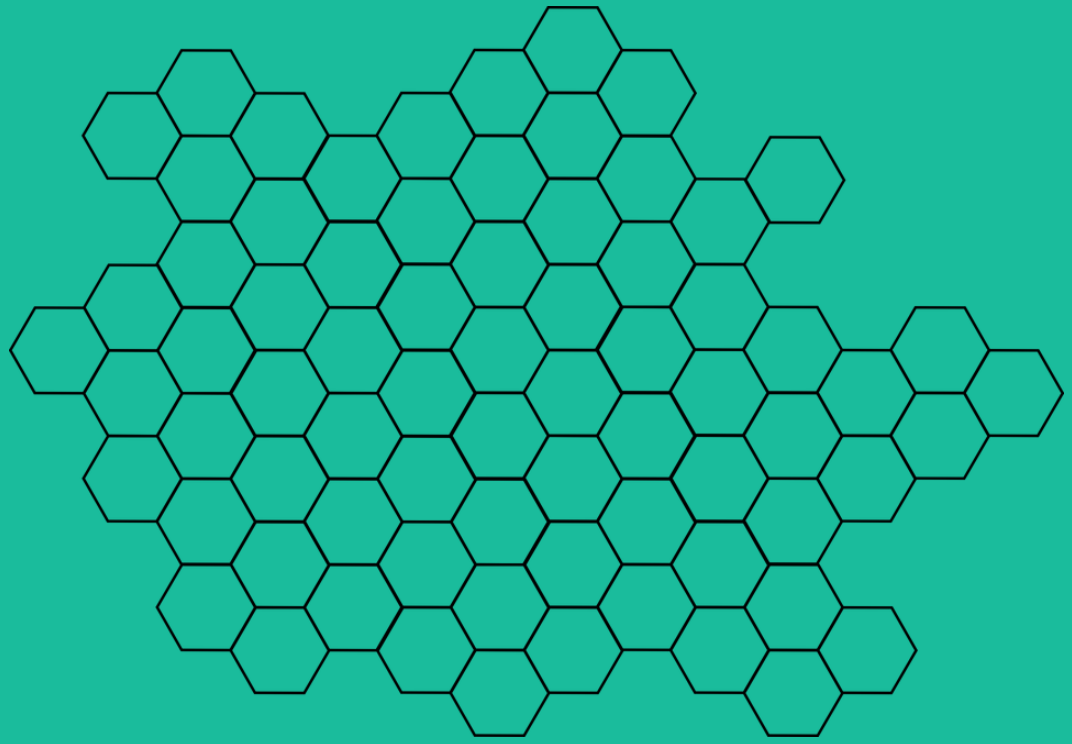
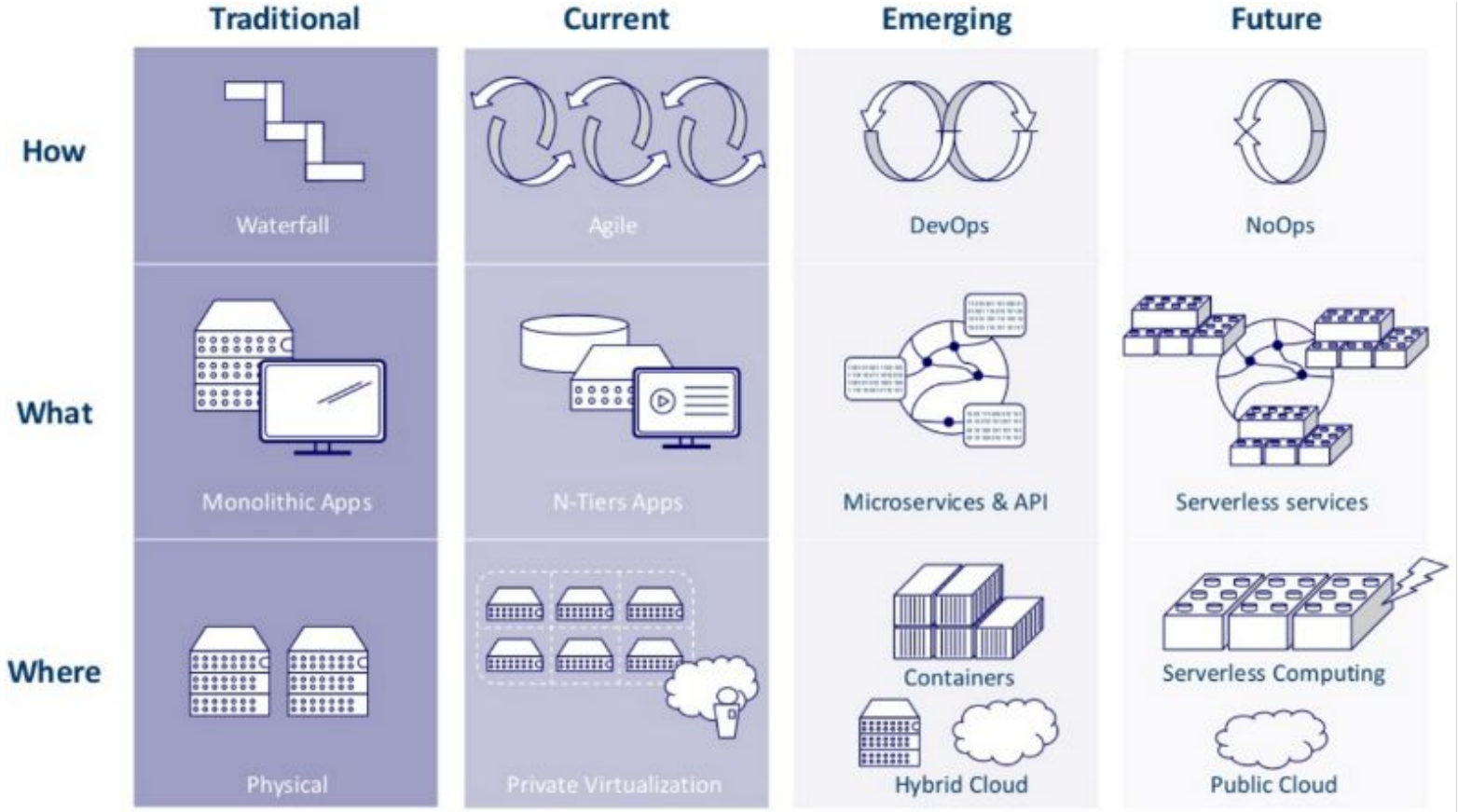


Chapter 5. Microservices

Bilkent University | CS443 | 2020, Spring | Dr. Orçun Dayıbaş

Introduction

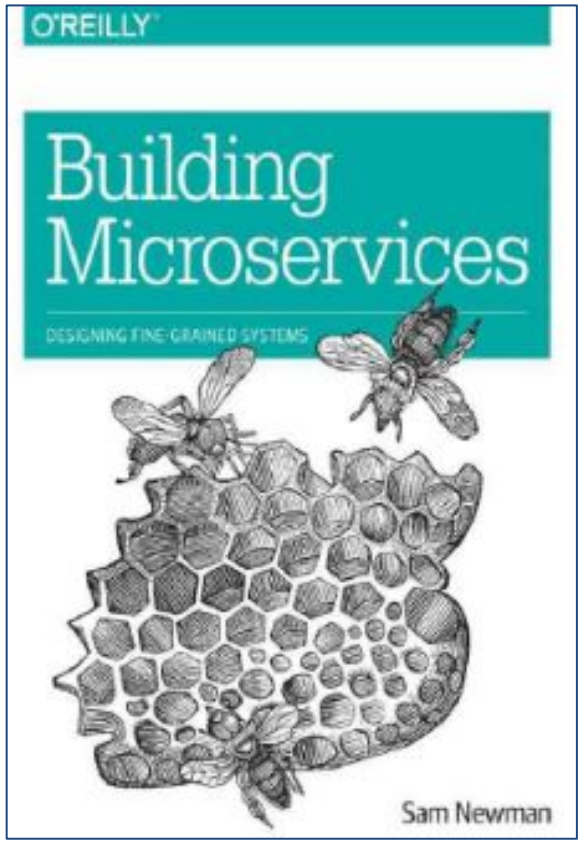
- IT Evolution



Source: http://www.slideshare.net/laurentbel01/it-architecture-evolution

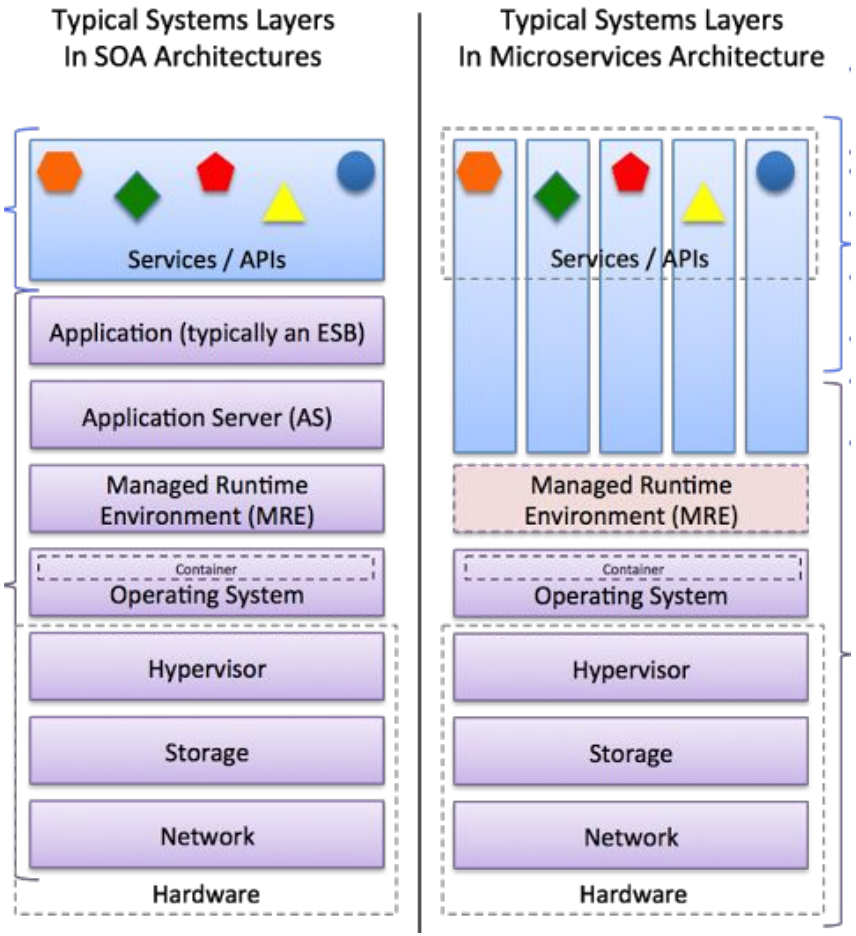
Introduction

- **Definition**
  - Microservices are a more concrete and modern interpretation of service-oriented architectures (SOA) used to build distributed software systems. Like in SOA, services in a microservice architecture are processes that communicate with each other over the network in order to fulfill a goal (Wikipedia).



Motivation

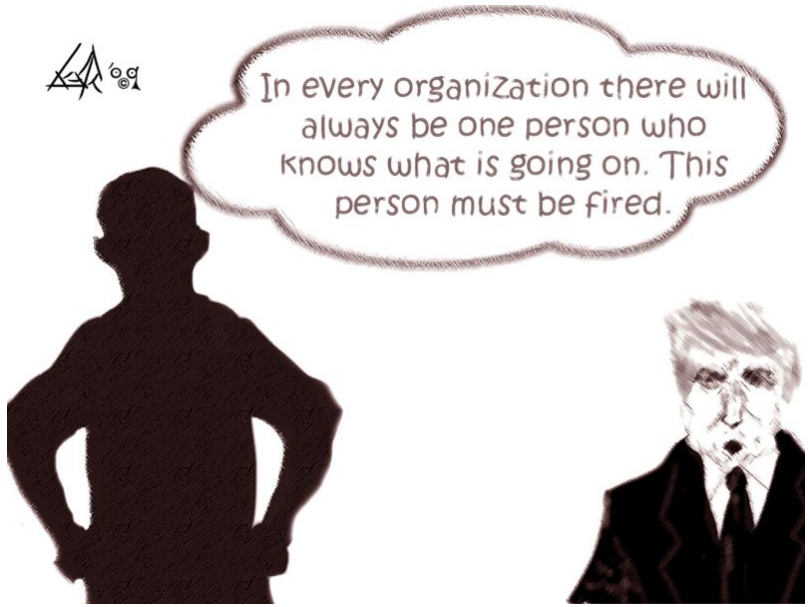
- **SOA vs. Microservices**



Source: http://www.soa4u.co.uk

Introduction

- **Conway's Law:**
  - «Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.»
  - http://www.melconway.com/Home/Conways_Law.html

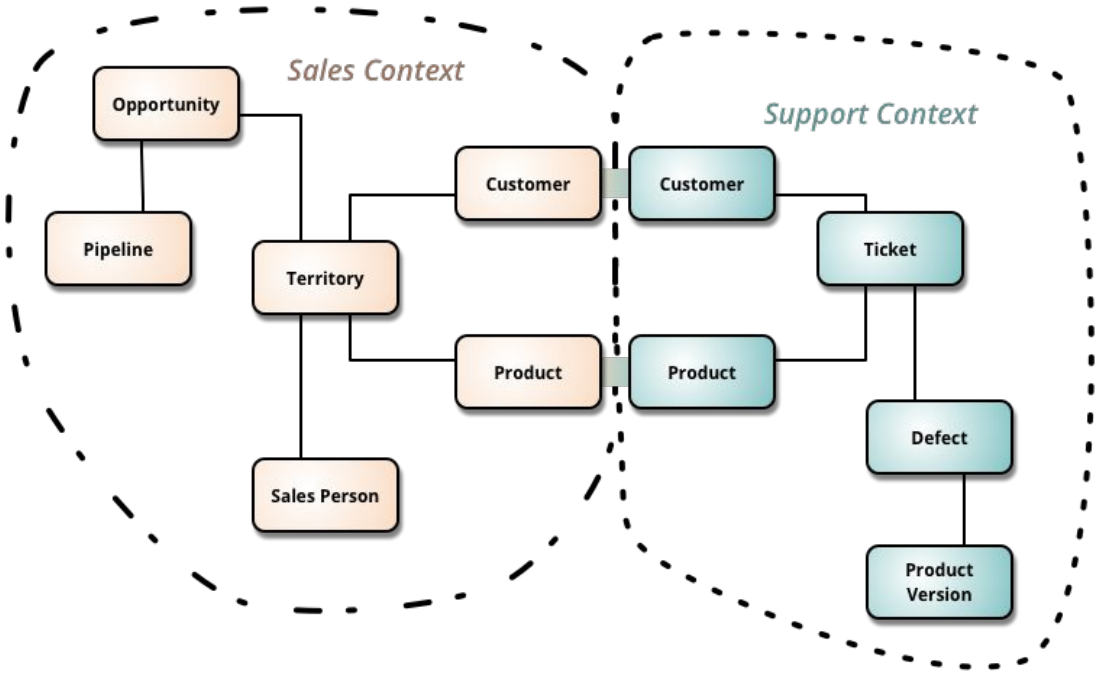


Source: https://commons.wikimedia.org/wiki/File:Conway%27s-Law--2.png

Microservices embraces Conway's Law to leverage the power of distributed teams, making distributed teams the norm irrespective of whether they are located onshore, offshore or nearshore. [source]

How to model services?

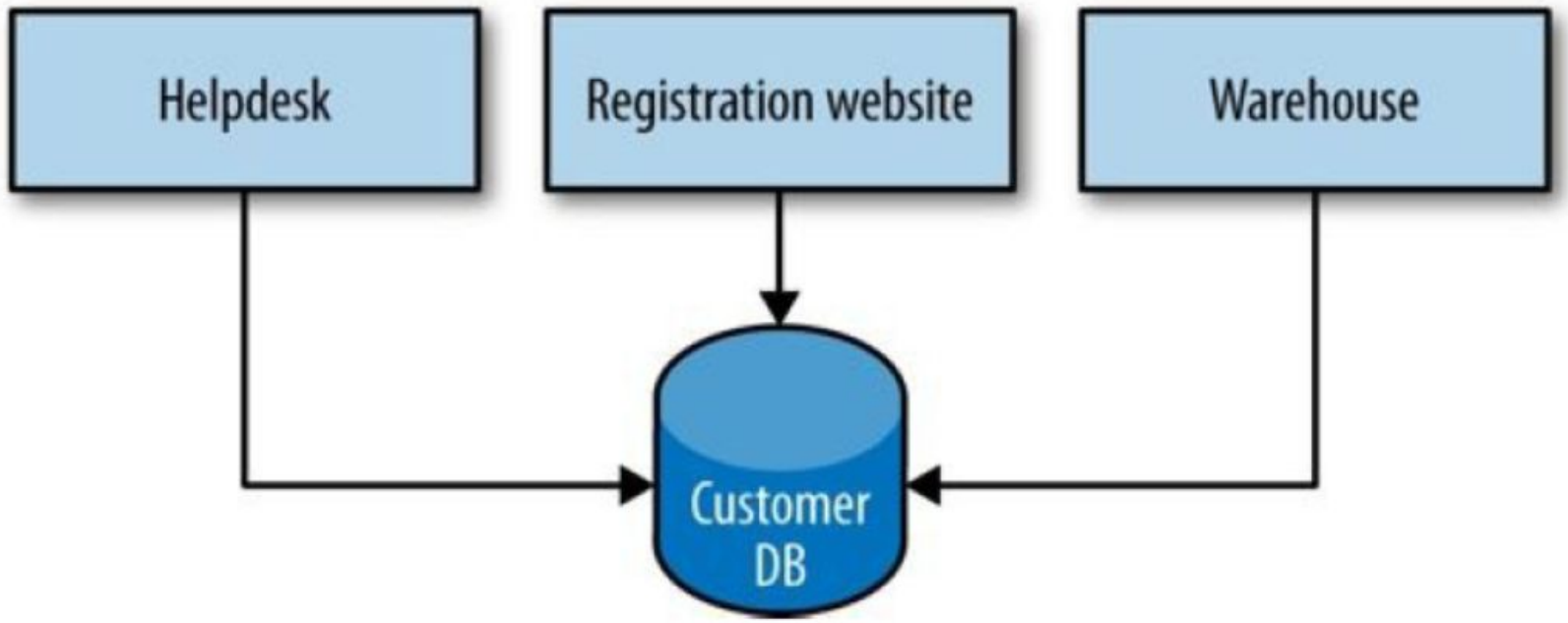
- **Loose Coupling & High Cohesion**
- **The Bounded Context (DDD)**
  - Business Capabilities
  - Avoid anemic CRUD-based services
  - Coarse-grained contexts -> fine-grained ones



Source:
http://martinfowler.com/bliki/BoundedContext.html

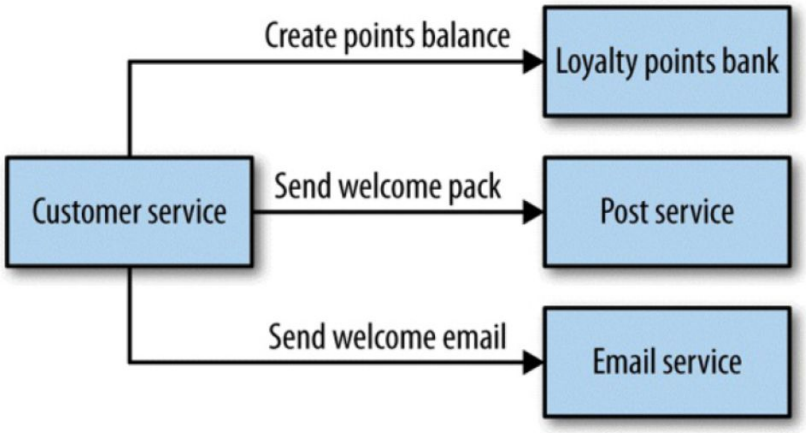
Integration

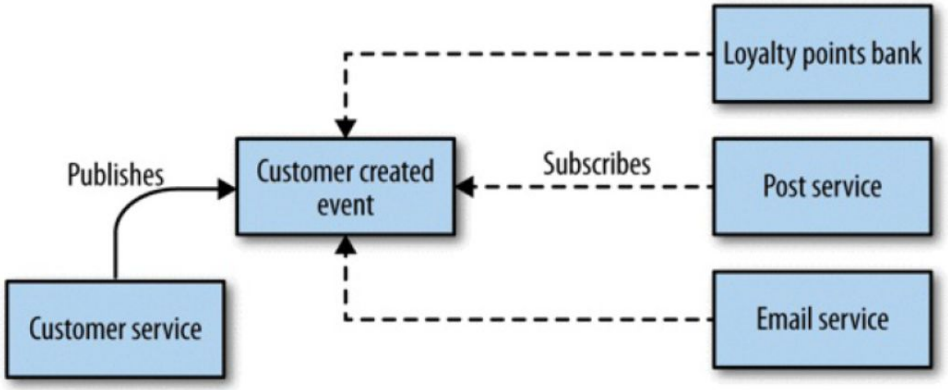
- **Shared DB**
  - DB-drivers → goodbye loose coupling
  - Spreaded logic → goodbye cohesion



Integration

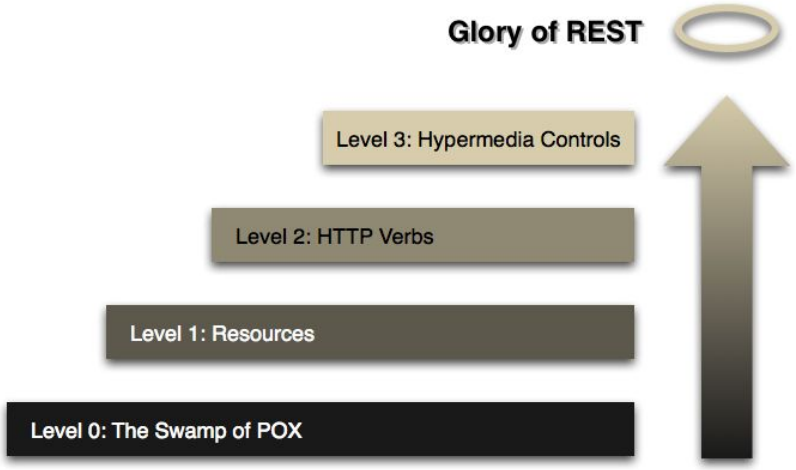
- **Service Communication**
  - Synchronous vs. Asynchronous
    - Request/response (sync. and quasi async.)
    - Event-based (real async.)
  - Inter-service comm. (Orchestration vs. Choreography)
    - Choreography decreases coupling





Integration

- **Service Communication**
  - RPC (SOAP, Thrift, Protocol Buffers)
    - Brittleness (still better than DB integration)
    - Preferred method for in composite/integration services
  - REST
    - Richardson Maturity Model (diff. Styles of REST are compared)
    - Preferred method for API/edge services (external facing)



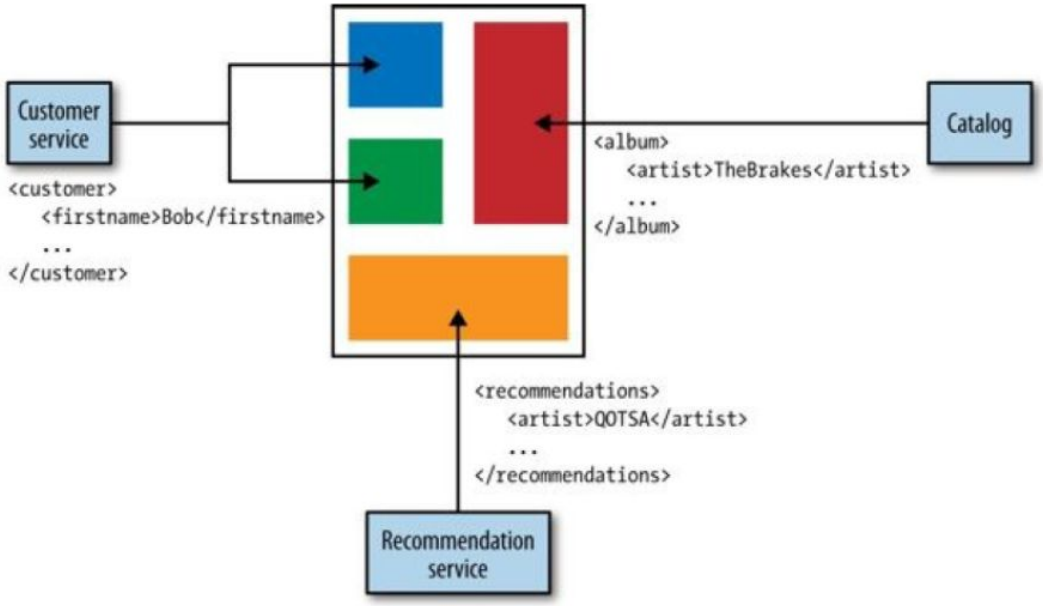
Source:
http://martinfowler.com/articles/richardsonMaturityModel.html

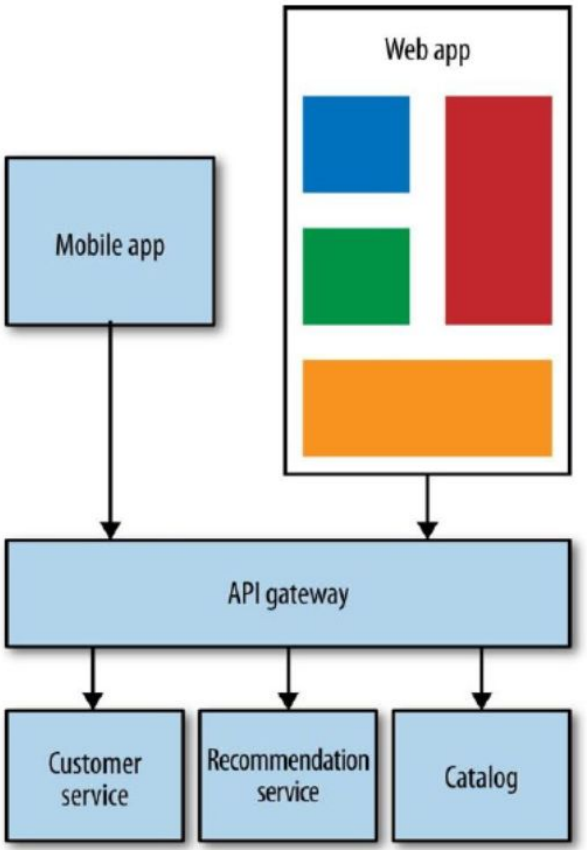
Integration

- **Reactive Manifesto**
  - http://www.reactivemanifesto.org/
- **DRY Principle in Microservices**
  - Bounded context vs. Reuse
- **Versioning**
  - Semantic Versioning: http://semver.org/
- **Postel's Law**
  - «Be conservative in what you do, be liberal in what you accept from others»

Integration

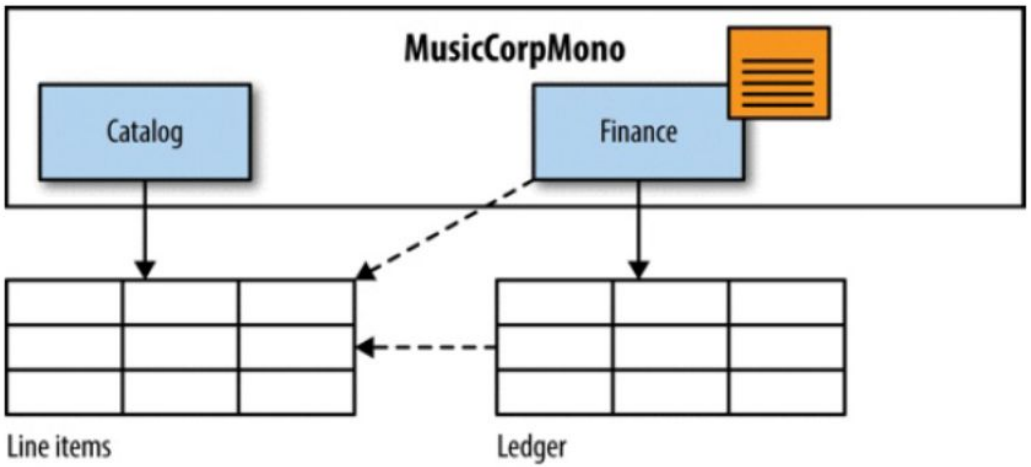
- User Interfaces

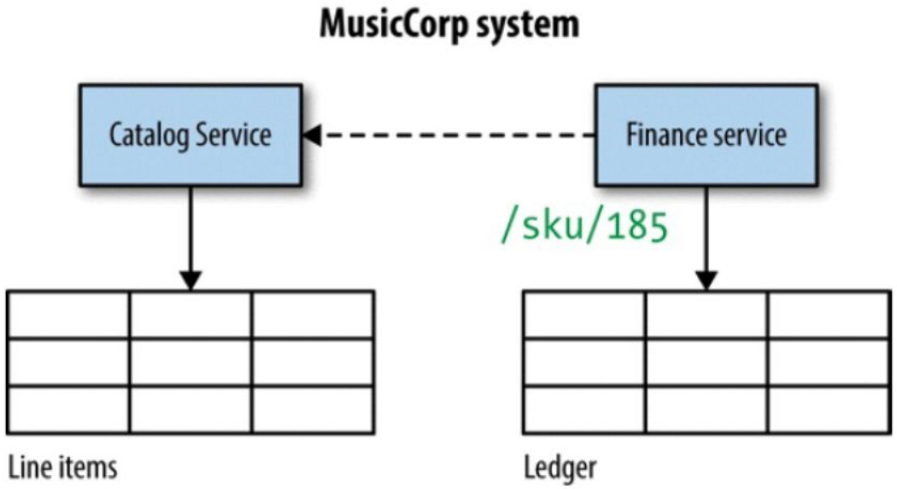




Splitting the monolith

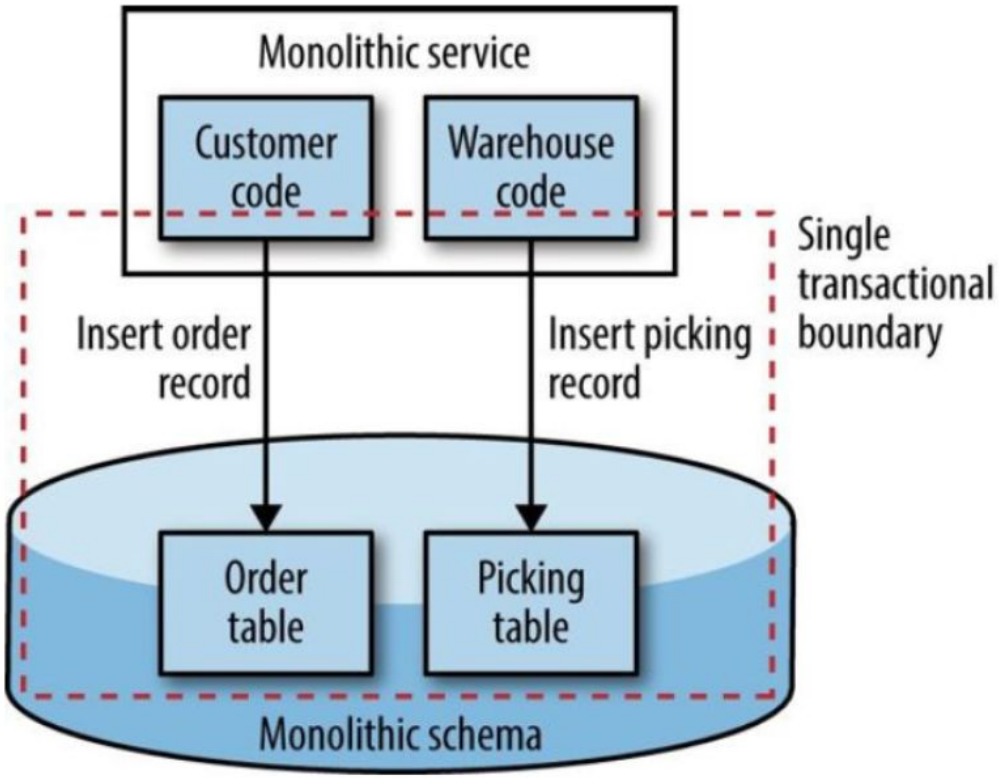
- **Database**
  - Getting rid of DB integration (ORM mapping file per bounded context)
  - Breaking foreign key relations
  - Shared static data (e.g. country codes)
    - db → property/code or db → services

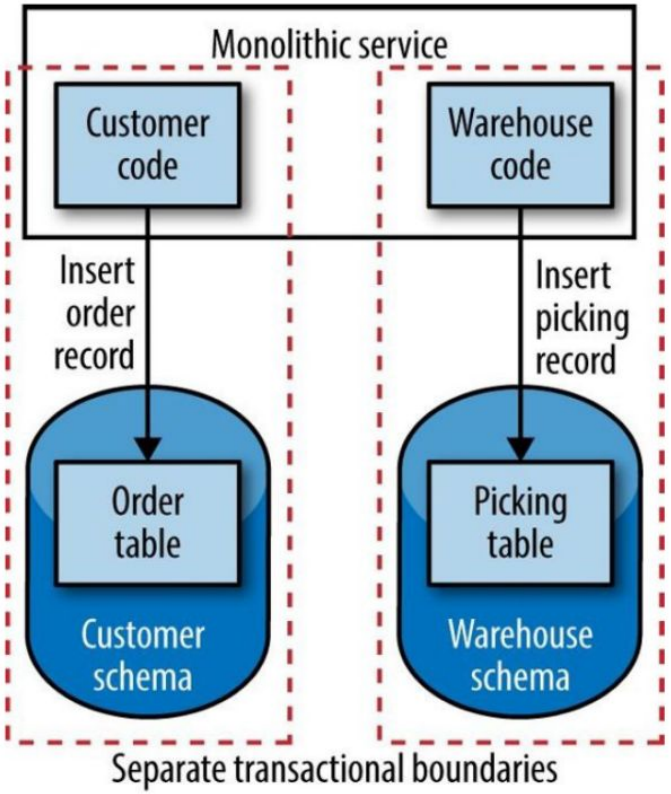




Splitting the monolith

- **Database**
  - Transactional boundaries
  - Distributed Transactions (e.g. JTA)





Deployment

- **Continuous Integration → Cont. Deployment**

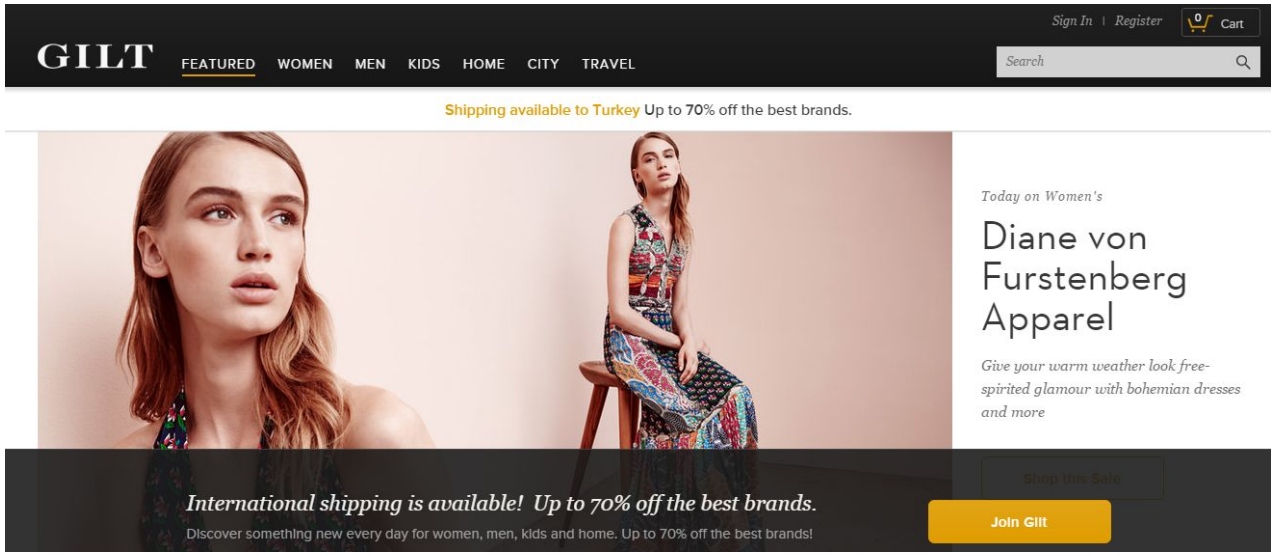
- **Platform-specific artifacts (e.g. jar)**
  - Embedded http process makes jars executable but the others may need additional sw to be launched. Puppet, ansible & chef can help here
- **OS artifacts**
- **Custom Images**
  - Drawbacks: time consuming task and large image files

Deployment

- **Immutable server**
  - To avoid configuration drift
  - Any change has to go through a build pipeline (disable SSH on server?)
- **Service-to-host mapping**
  - Multiple services per host → hard to track services (CPU load etc..).
  - Application containers → constraints tech choices
  - Single service per host: easiest solution if you don't have PaaS
- **PaaS**
  - Ex: Azure Service Fabric

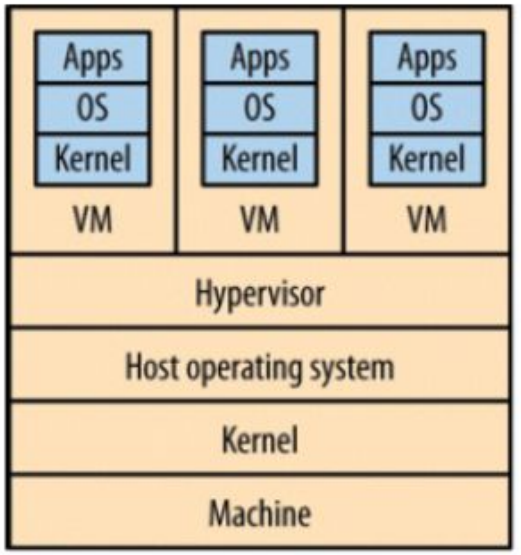
Deployment

- **Automation**
  - Service-per-host is difficult without automation
  - Example: Gilt (Eng. blog: http://tech.gilt.com/)
    - **2009:** Decided to migrate to microservices
    - **2010:** They had 10 microservices live
    - **2012:** Over 100
    - **2014:** Over 450 (3 services per developer)

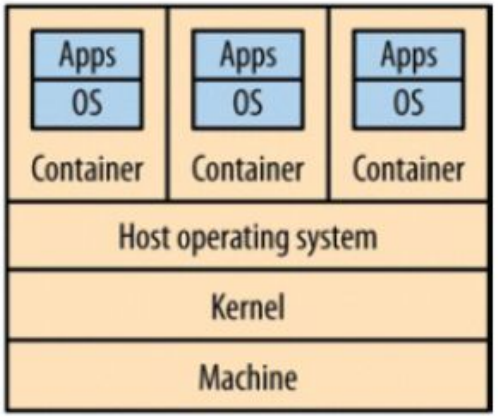


Deployment

- **From Physical to Virtual**
  - **Vagrant** (VM provisioning)
  - Linux containers: virtualization without hypervisor, lightweight
  - **Docker:** handling containers for you (container provisioning).
  - **CoreOS:** stripped-down Linux that provides only essential services to allow docker to run. Rather than using package mng.
  - **Kubernetes:** container-orchestration system for automating application deployment, scaling, and management



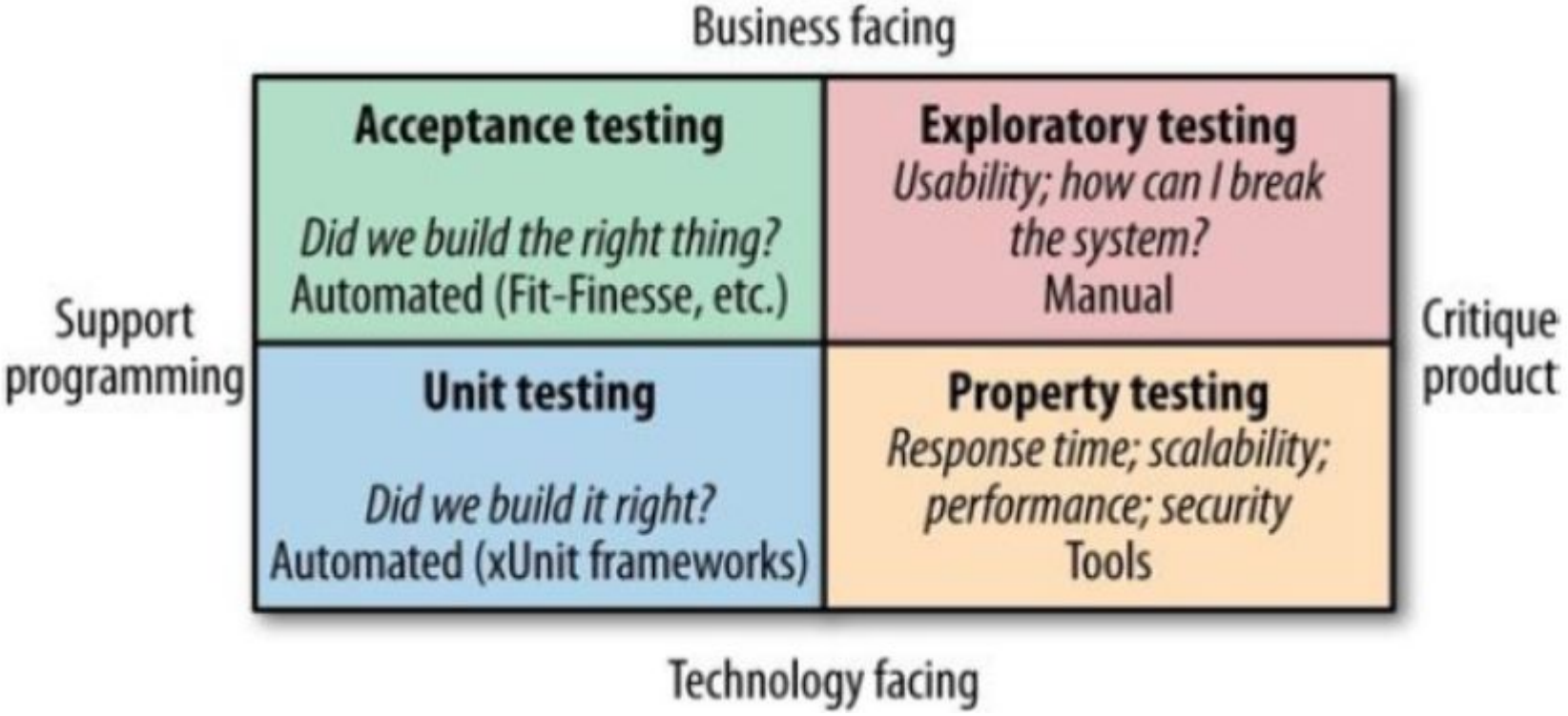
Standard virtualizations



Container-based virtualizations (LXC)

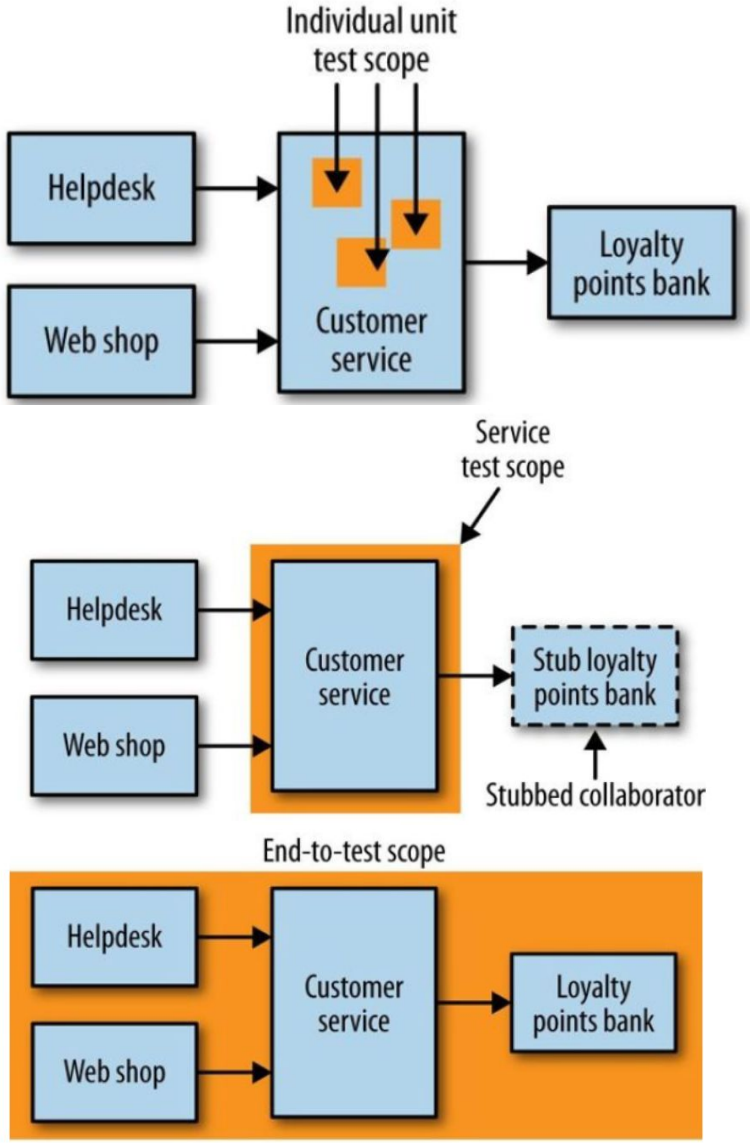
Testing

- **Types of tests**



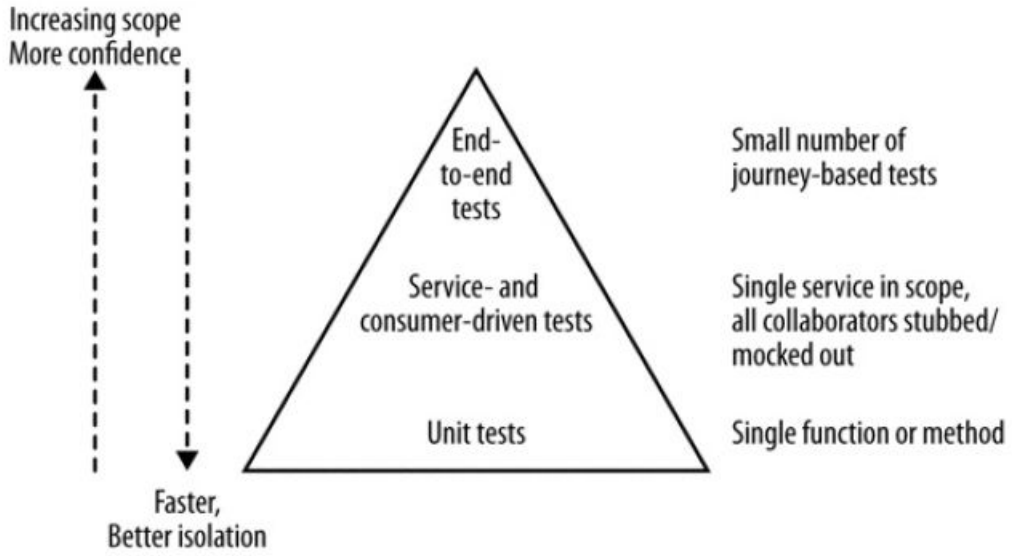
Testing

- Unit Tests
- Service Tests
- E2E Tests



Testing

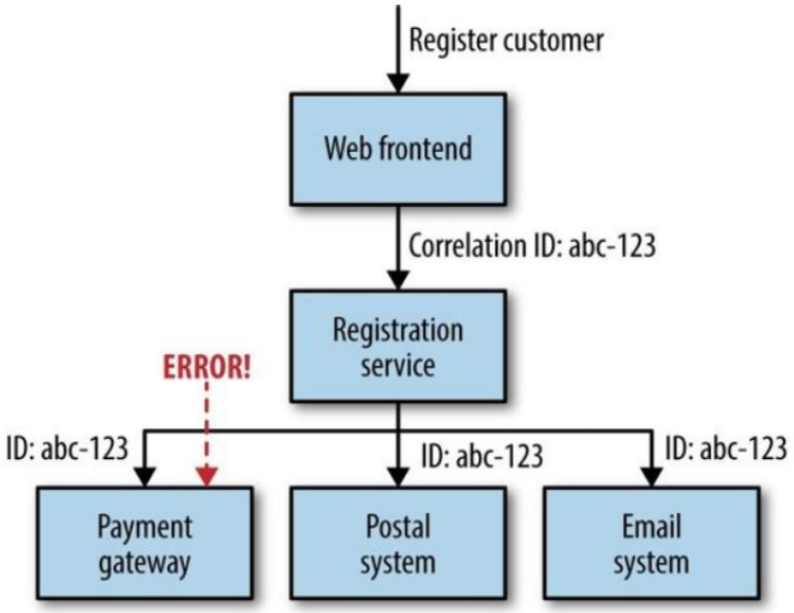
- **Test Pyramid**



- **Testing after production**
  - Canary releasing vs. blue/green releasing
  - MTTR over MTBF
- **Cross-functional (Non-functional) testing**
  - Performance tests

Monitoring

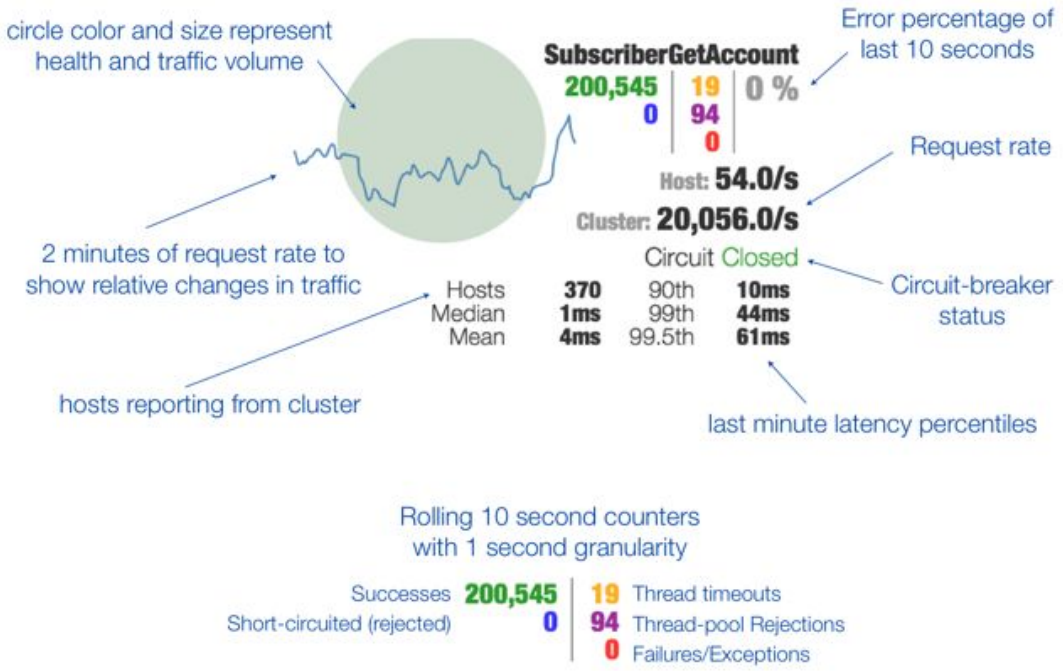
- **Correlation Ids**
  - Twitter's Zipkin: http://zipkin.io/
  - Google's Dapper: http://research.google.com/pubs/pub36356.html



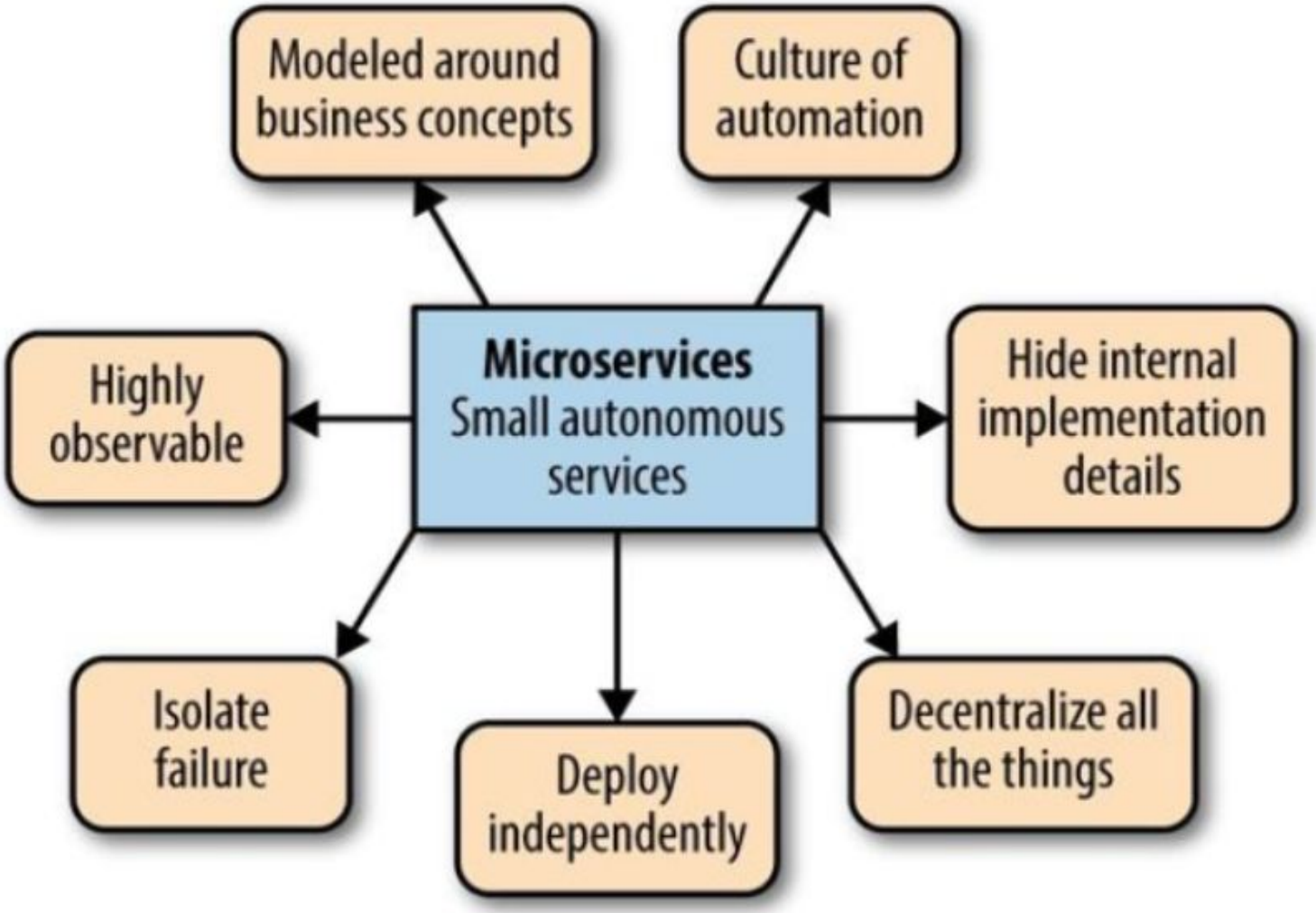

```
15-02-2014 16:01:01 Web-Frontend INFO [abc-123] Register
15-02-2014 16:01:02 RegisterService INFO [abc-123] RegisterCustomer…
15-02-2014 16:01:03 PostalSystem INFO [abc-123] SendWelcomePack…
15-02-2014 16:01:03 EmailSystem INFO [abc-123] SendWelcomeEmail…
15-02-2014 16:01:03 PaymentGateway ERROR [abc-123] ValidatePayment…
```

Microservices at Scale

- **What do we need to do to handle failure in our systems?**
  - Timeouts
  - Circuit breakers
    - Netflix's Hystrix: https://github.com/Netflix/Hystrix



Principles of Microservices



What to avoid?

- **When shouldn't you use microservices?**
  - Less well you understand a domain harder to find bounded contexts
  - Greenfield projects
  - Doing things manually → harder to scale

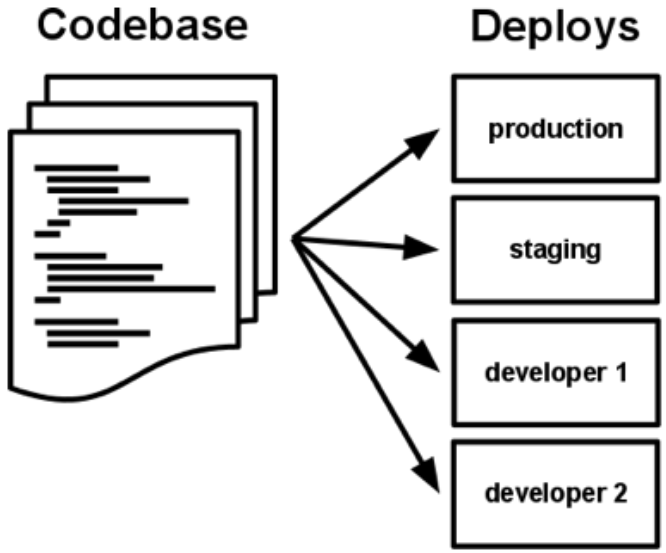
Source: https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/

Any methodology?

- **Microservices is not an architecture, it is a style**
  - It has a couple of principles & you can adapt an arch. (refer to chapter-3)
  - Cloud deployed → Cloud-native
    - There should be a tested lists of rules (e.g. SOLID in OOP) that could guide development of good implementation
- **The 12-factor App**
  - https://12factor.net/
  - Use declarative formats for setup automation
  - Have a clean contract with the underlying OS
  - Minimize divergence between development & production

The 12-factor App

- **1. Codebase**
  - *“One codebase tracked in a VCS (Version Control System) with many deploys”*
  - Do not create two different repositories when all you need to do is different setup for production
  - Multiple apps sharing the same code is a violation of twelve-factor → make it a lib. and use dep. manager



The 12-factor App

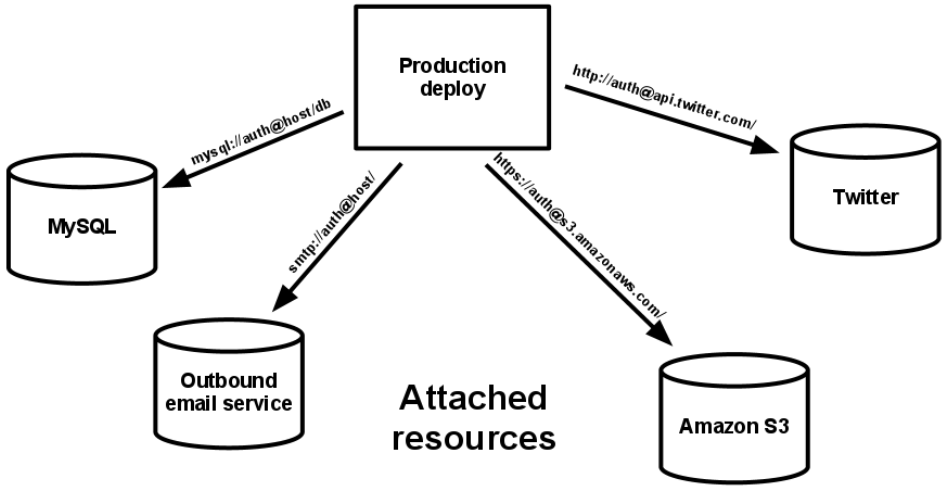
- **2. Dependencies**
  - *“Explicitly declare and isolate dependencies”*
  - Do not rely on implicit existence of anything (system-wide packages, default configurations, etc.)
    - Ex: Using curl or ImageMagick in your application
  - As long as you use a standard build tool (npm, yarn, maven, gradle, NuGet) you have the basics covered
  - Using configuration management tools (like Chef, Puppet, Ansible) can solve system-wide inconsistency issues

The 12-factor App

- **3. Config**
  - *“Store config in the environment”*
  - An app’s config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:
    - **Resource handles** to the database, Memcached, and other backing services
    - **Credentials** to external services such as Amazon S3
    - Per-deploy values such as **the canonical hostname** for the deploy
  - Config files vs. **Env. variables**
    - EVs are easy to change between deploys without changing any code
    - Unlike config files, there is little chance of them being checked into the code repo accidentally
    - EVs are a language&OS-agnostic standard

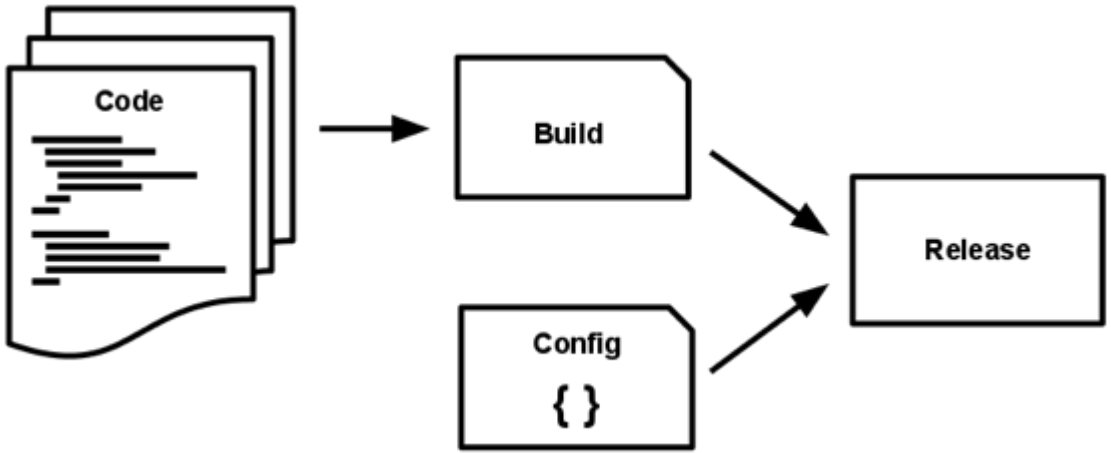
The 12-factor App

- **4. Backing Services**
  - *"Treat backing services as attached resources"*
  - A backing service is any service (DB, cache, etc.) the app consumes over the network as part of its normal operation
  - Resources (local or 3rd party) can be attached to and detached from deploys at will.
    - For example, if the DB is misbehaving, the app's administrator might spin up a new database server restored from a recent backup (all **without any code changes**).



The 12-factor App

- **5. Build, release, run**
  - *“Strictly separate build and run stages”*
  - Build stage: converting code repo into an executable bundle
  - Release stage: getting the build and combining it with a config on a certain environment- ready to run
  - Run stage: starting the app in the deployment (production).



The 12-factor App

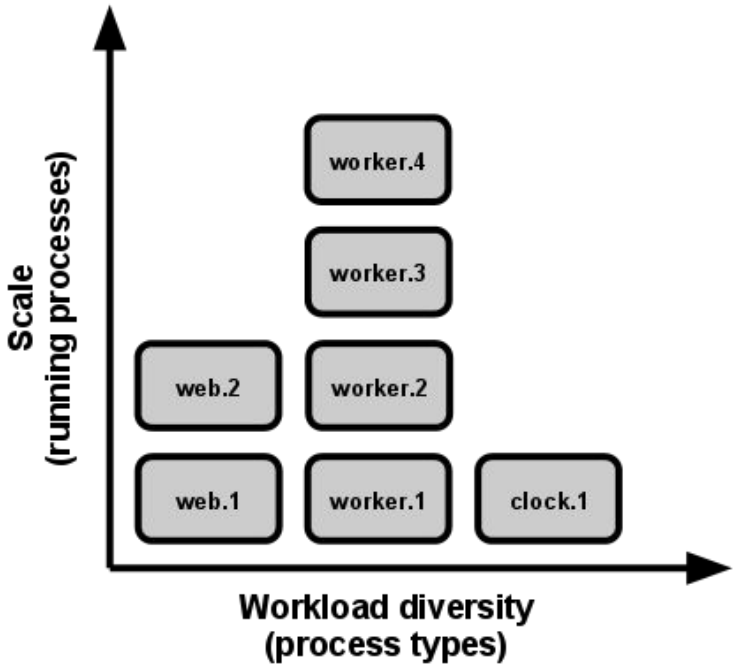
- **6. Processes**
  - *“Execute the app as one or more stateless processes”*
  - Twelve-factor processes are **stateless** and **share-nothing**. Any data that needs to persist must be stored in a stateful backing service, typically a database
  - Some web systems rely on “sticky sessions”
    - It is caching user session **data in memory of the app’s process and expecting future requests from the same visitor** to be routed to the same process.
    - **Sticky sessions are a violation of twelve-factor** and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.

The 12-factor App

- **7. Port Binding**
  - *“Export services via port binding”*
  - Web apps are sometimes executed inside a webserver container
    - Ex: Java apps might run inside Tomcat.
  - The twelve-factor app is completely **self-contained** and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service.
    - The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.
  - The port-binding approach means that **one app can become the backing service for another app**, by providing the URL to the backing app as a resource handle in the config for the consuming app.

The 12-factor App

- **8. Concurrency**
  - *"Scale out via the process model"*
  - The idea is that, as you need to scale, you should be deploying **more copies of your application (processes)** rather than trying to make your application larger (by running a single instance on the most powerful machine available)
  - The share-nothing, horizontally partitionable nature of twelve-factor app processes means that **adding more concurrency is a simple and reliable operation**.



The 12-factor App

- **9. Disposability**
  - *“Maximize robustness with fast startup and graceful shutdown”*
  - The twelve-factor app’s processes are disposable, meaning they can be **started or stopped at a moment’s notice**. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.
  - Fast startup & Graceful shutdown
    - Processes should strive to **minimize startup time**. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs.
    - Processes **shut down gracefully** when they receive a `SIGTERM` signal from the process manager.

The 12-factor App

- **10. Dev/prod parity**
  - *"Keep dev., staging, and production as similar as possible"*
  - Historically, there have been substantial gaps between development and production
    - **The time gap:** A developer may work on code that takes days, weeks, or even months to go into production.
    - **The personnel gap:** Developers write code, ops engineers deploy it.
    - **The tools gap:** Developers may be using a stack like Nginx, SQLite, and OS X, while the production deploy uses Apache, MySQL, and Linux.
  - The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small
    - **Make the time gap small:** a developer may write code and have it deployed hours or even just minutes later.
    - **Make the personnel gap small:** developers who wrote code are closely involved in deploying it and watching its behavior in production.
    - **Make the tools gap small:** keep development and production as similar as possible (containerization helps a lot).

The 12-factor App

- **11. Logs**
  - *“Treat logs as event streams”*
  - Logs provide visibility into the behavior of a running application
  - Logs are the stream of **aggregated, time-ordered events** collected from the output streams of all running processes and backing services.
  - **A twelve-factor app never concerns itself with routing or storage of its output stream.**
    - It should not attempt to write to or manage log files. Instead, each running process writes its event stream, unbuffered, to `stdout`.
  - The event stream for an app can be routed to a file, or the stream can be sent to a log indexing and **analysis system** (e.g. Splunk, ElasticSearch) or a general-purpose **data warehousing** system (e.g. Hadoop/Hive).

The 12-factor App

- **12. Admin Process**
  - *“Run admin/management tasks as one-off processes”*
  - One-off administrative or maintenance tasks for the app
    - Running **database migrations** (e.g. manage.py migrate in Django, rake db:migrate in Rails).
    - Running a console (also known as a REPL shell) to **run arbitrary code** or inspect the app’s models against the live database.
    - Running **one-time scripts** committed into the app’s repo (e.g. php scripts/fix_bad_records.php).
  - Admin tasks should be run from the relevant servers (e.g. production servers).
    - This is easiest done by **shipping admin code with application code** to provide these capabilities. The tools should be there even if they are not part of the standard execution of the service.
    - Twelve-factor strongly favors languages which provide a **REPL shell** out of the box, and which make it easy to run one-off scripts.

Q/A