Cloud Native Applications and Microservices Architecture

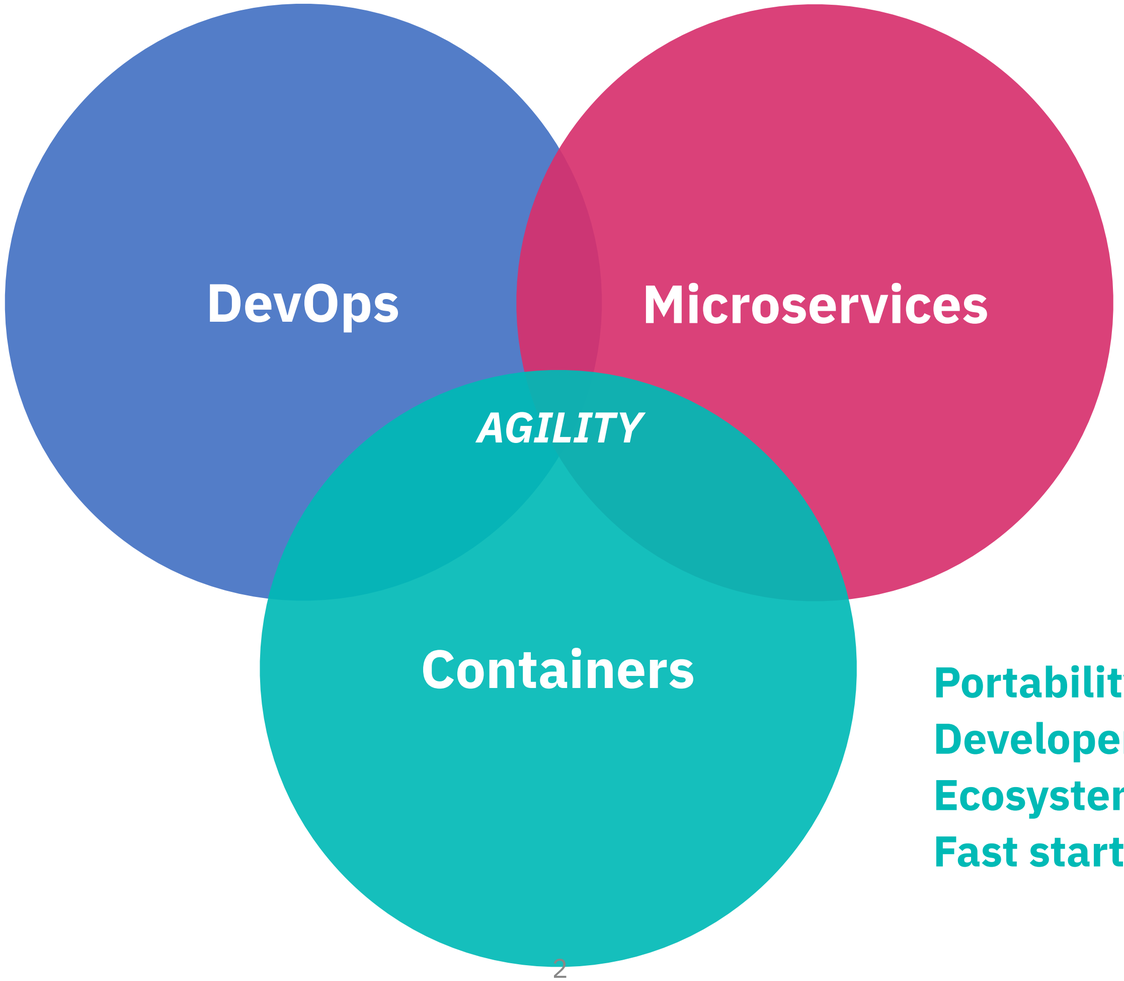
Instructor:

John J Rofrano

Senior Technical Staff Member, DevOps Champion
IBM T.J. Watson Research Center
rofrano@us.ibm.com (@JohnRofrano)

THE PERFECT STORM

Cultural Change
Automated Pipeline
Everything as Code
Immutable Infrastructure

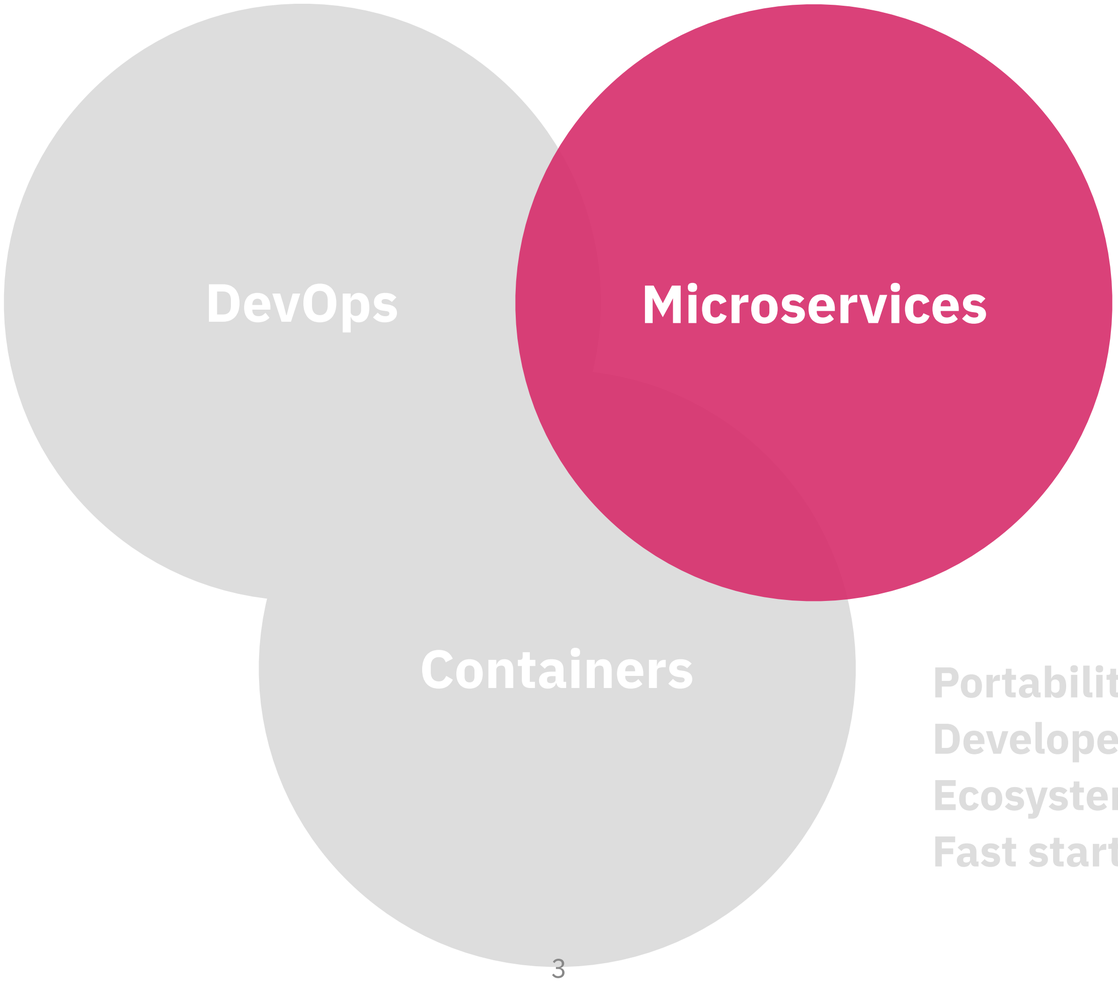


Loose Coupling/Binding
RESTful APIs
Designed to resist failures
Test by break / fail fast

Portability
Developer Centric
Ecosystem enabler
Fast startup

THE PERFECT STORM: Microservices

Cultural Change
Automated Pipeline
Everything as Code
Immutable Infrastructure



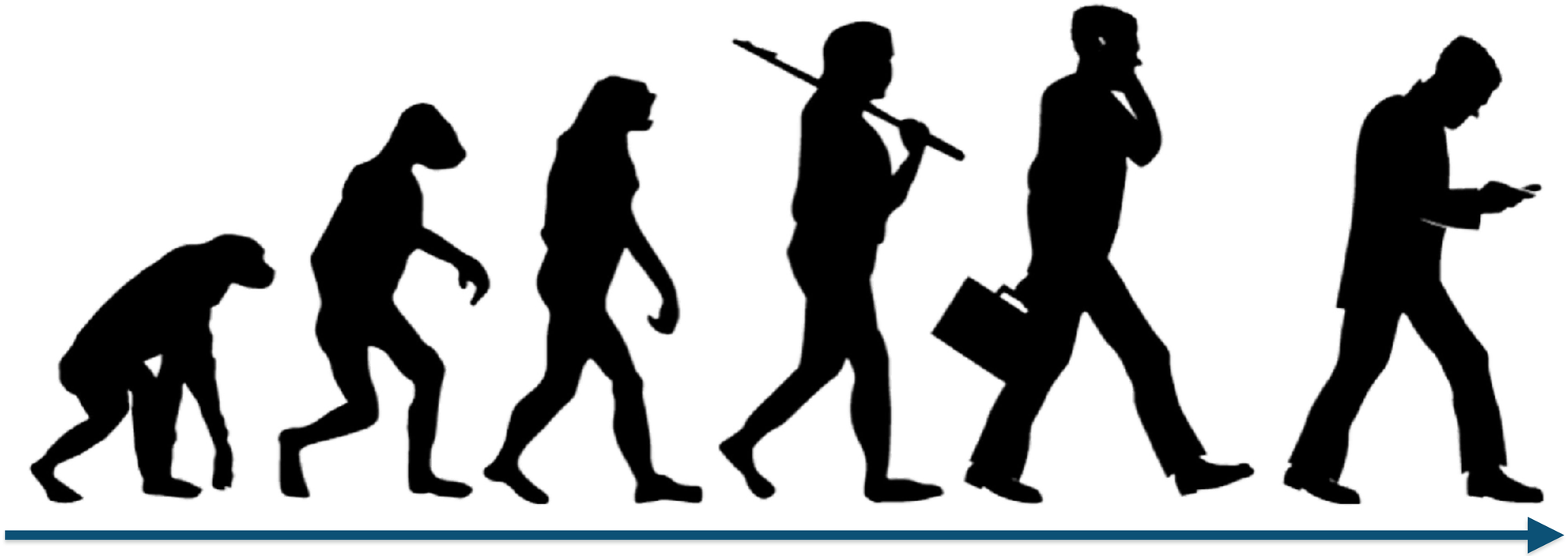
Loose Coupling/Binding
RESTful APIs
Designed to resist failures
Test by break / fail fast

Portability
Developer Centric
Ecosystem enabler
Fast startup

“To fully leverage DevOps,
you need to **think** differently about application design.”


High Resonanceproof Cassette Mechanism
EXTENDED RANGE • LOW DISTORTION
POSITION • HIGH
UD II 90
Mix Tape #1: The Old Days
N.R YES NO

Evolution of App Designs



C, C++
Code and Manual Test
Monolithic Programs
Bare Metal

Java, J2EE
Manual Deployment
Tiered Applications
Virtual Machines

Ruby on Rails, Django
Automated Testing
Service Oriented, RESTful
Platform As A Service

Python Flask, Node, Swift
Automated Deployment
Stateless Microservices
Docker Containers

Cloud Native

What is Cloud Native?

- Cloud-native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model that are built using multiple, independent microservices.
- DevOps drives the patterns of high performing organizations delivering software faster, consistently and reliably at scale
- Leveraging automation to improve human performance in a high trust culture, moving faster and safer with confidence and operational excellence

Core Elements

Architecture

Microservice-based

Practice:

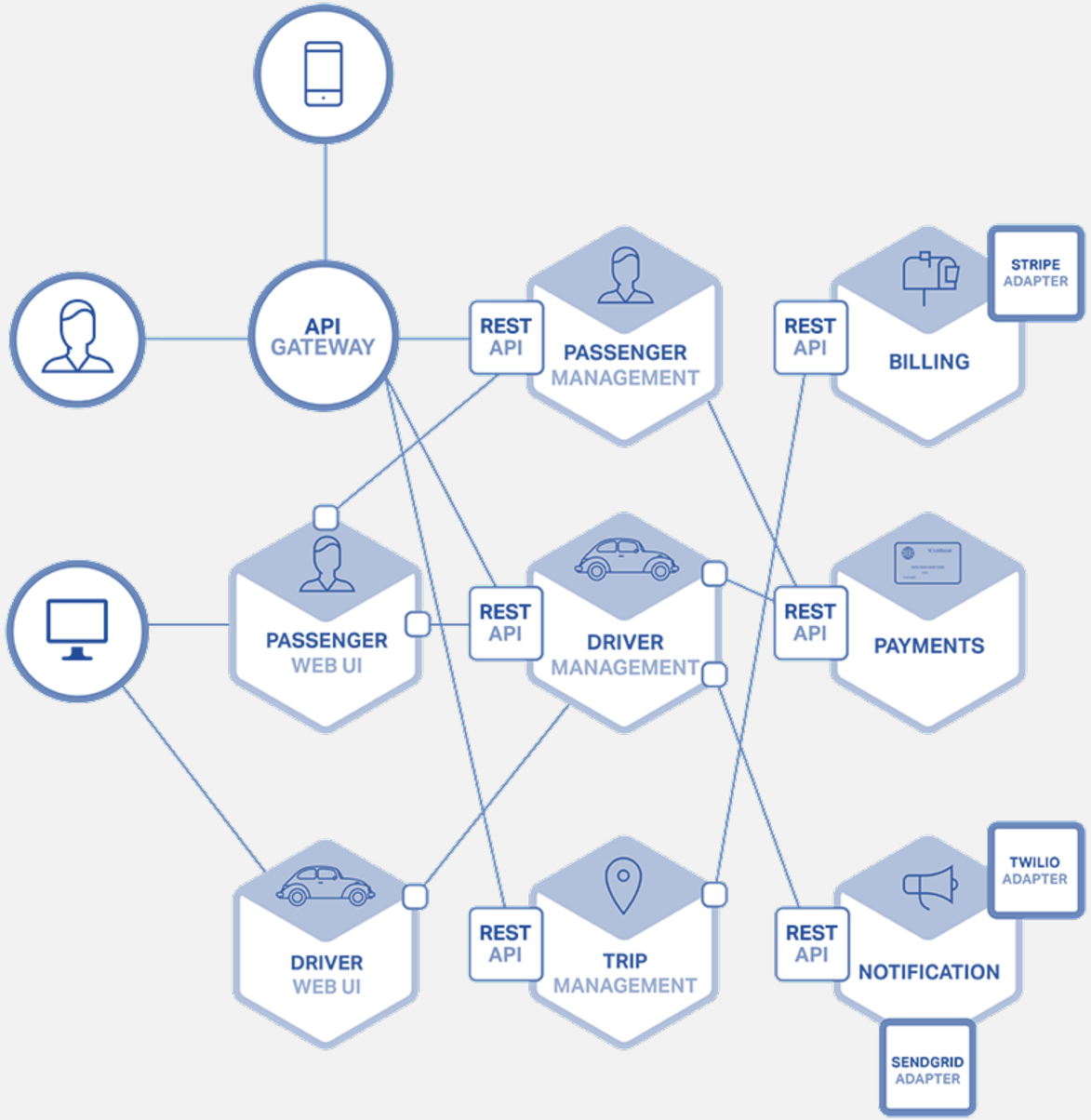
Agile with DevOps & 12-factor methodology

Technology:

Containers / Kubernetes
CI / CD Tools

Cloud Native Applications

- The Twelve-Factor App describes patterns for cloud-native architectures which leverage microservices.
- Applications are designed as a collection of stateless microservices.
- State is maintained in separate databases and persistent object stores.
- Resilience and horizontal scaling is achieved through deploying multiple instances.
- Failing instances are killed and re-spawned, not debugged and patched.
- DevOps pipelines help manage continuous delivery of services.



https://www.nginx.com/blog/introduction-to-microservices/

Architecture: Microservices

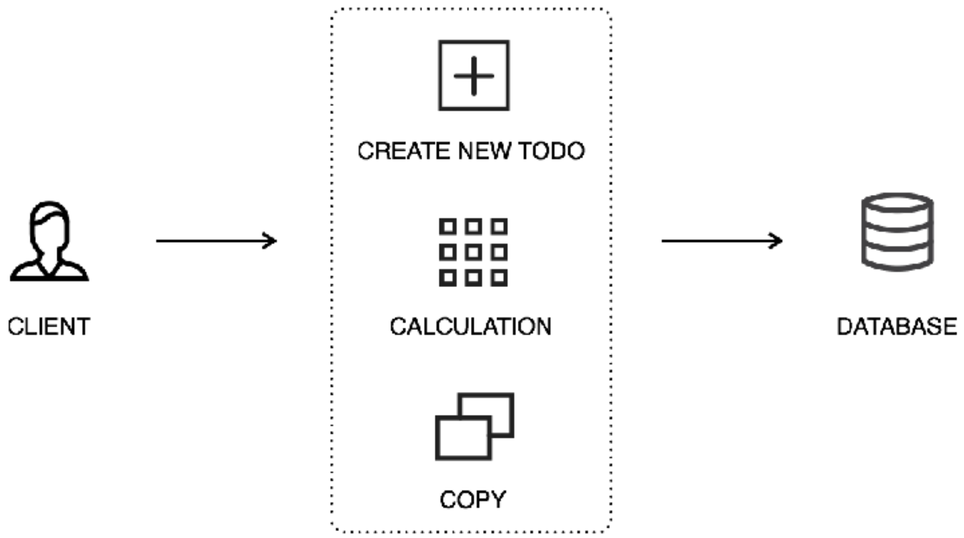
What are Microservices?

"...the microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery."

- Martin Fowler and James Lewis

Monolith vs Microservices Architecture

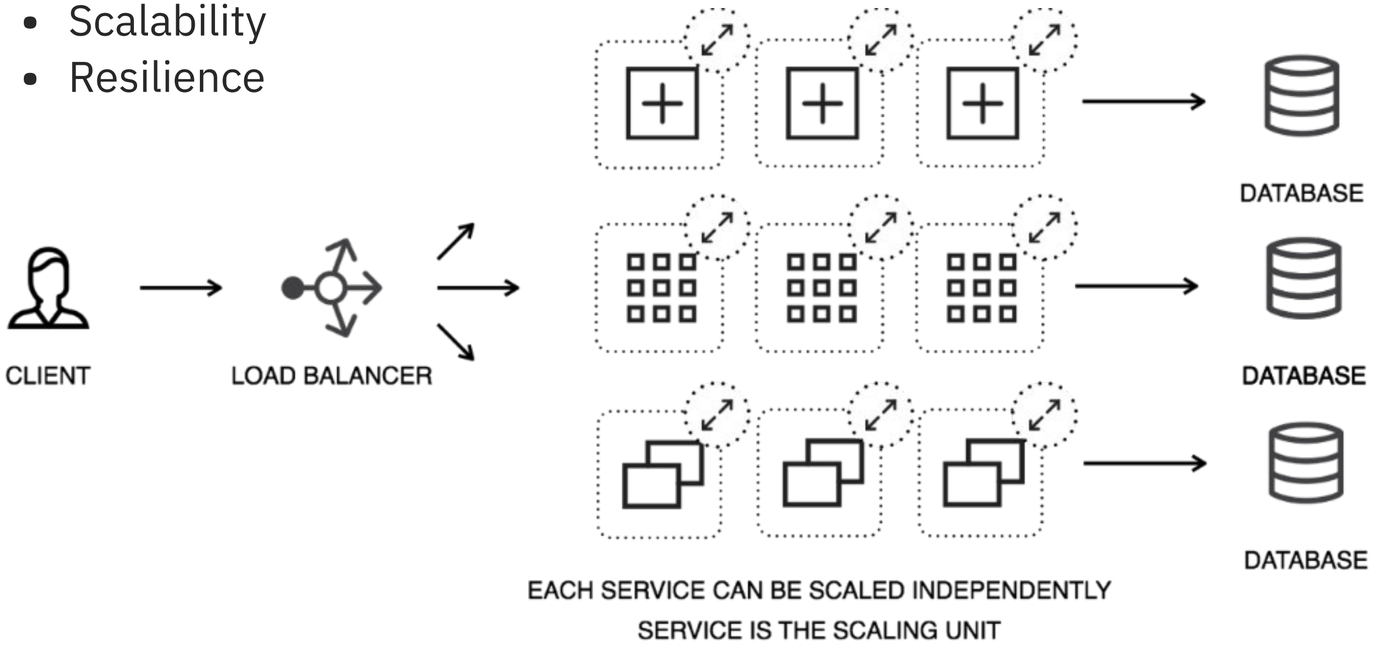
Monolith



All services combined into one build,
written in the same language and application framework

Microservices

- Agility
- Scalability
- Resilience

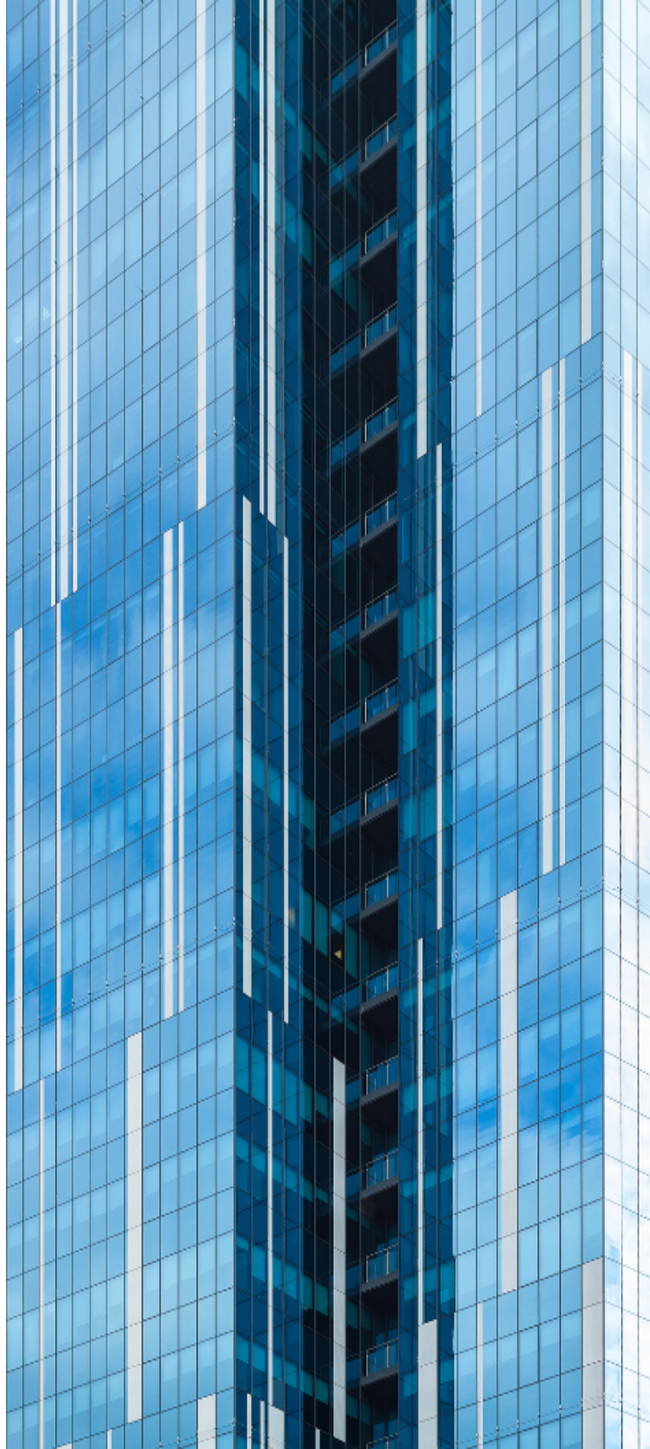


A microservice is a granular, decoupled component within a broader application

Microservice Architecture

An architecture style aimed to achieve flexibility, resiliency and control, based on the following principles:

- Single purpose Loose Coupling bounded context
- Independent life cycle: developed, deployed and scaled… and hopefully, fail independently
- Design for resiliency and owns it’s own data
- Polyglot — independent code base
- Built by autonomous teams with end-to-end responsibility, doing Continuous Delivery
- Communicates with other services over a well defined API



Advantages of Microservices

- Developed by a single team
- Developed independently
- Developed on its own timetable
- Each can be developed in a different language
- Manages its own data
- Scales and fails independently

Microservices are a paradise RIGHT?

Beware There be Dragons

Microservice Challenges

- Developers must have significant operational and development skills (DevOps / Multiple languages)
- Service interfaces and versions
- Duplication of effort across service implementations
- Extra complexity of creating a distributed system with these issues, among others:
  - Network latency
  - Fault tolerance
  - Serialization

Microservice Challenges

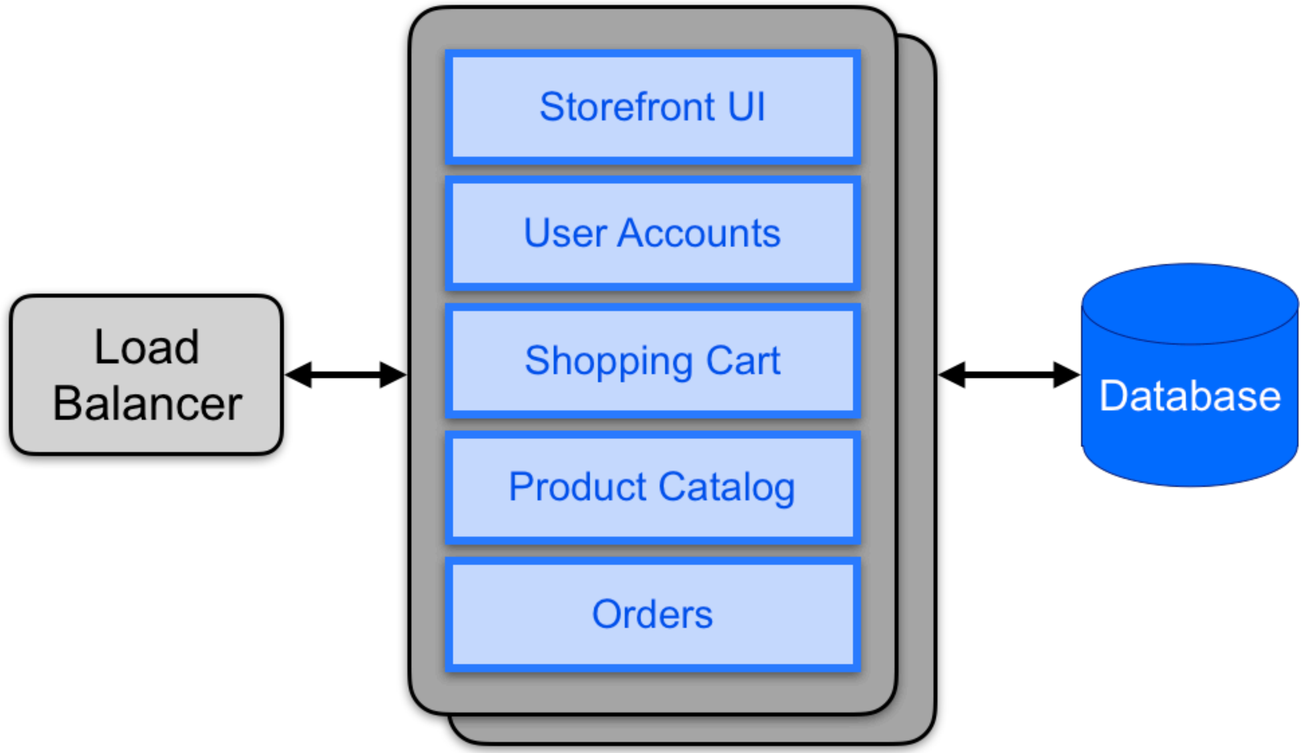
- Designing decoupled non transactional systems is difficult
- Avoiding latency of large numbers of small service invocations
- Locating service instances
- Maintaining availability and consistency with partitioned data
- End-to-end automated testing

“Knowing **when** to use a technology is as important as knowing **how** to use it.”

http://www.magnasoma.com/content/images/Magnasoma-Monolith-3-01.jpg

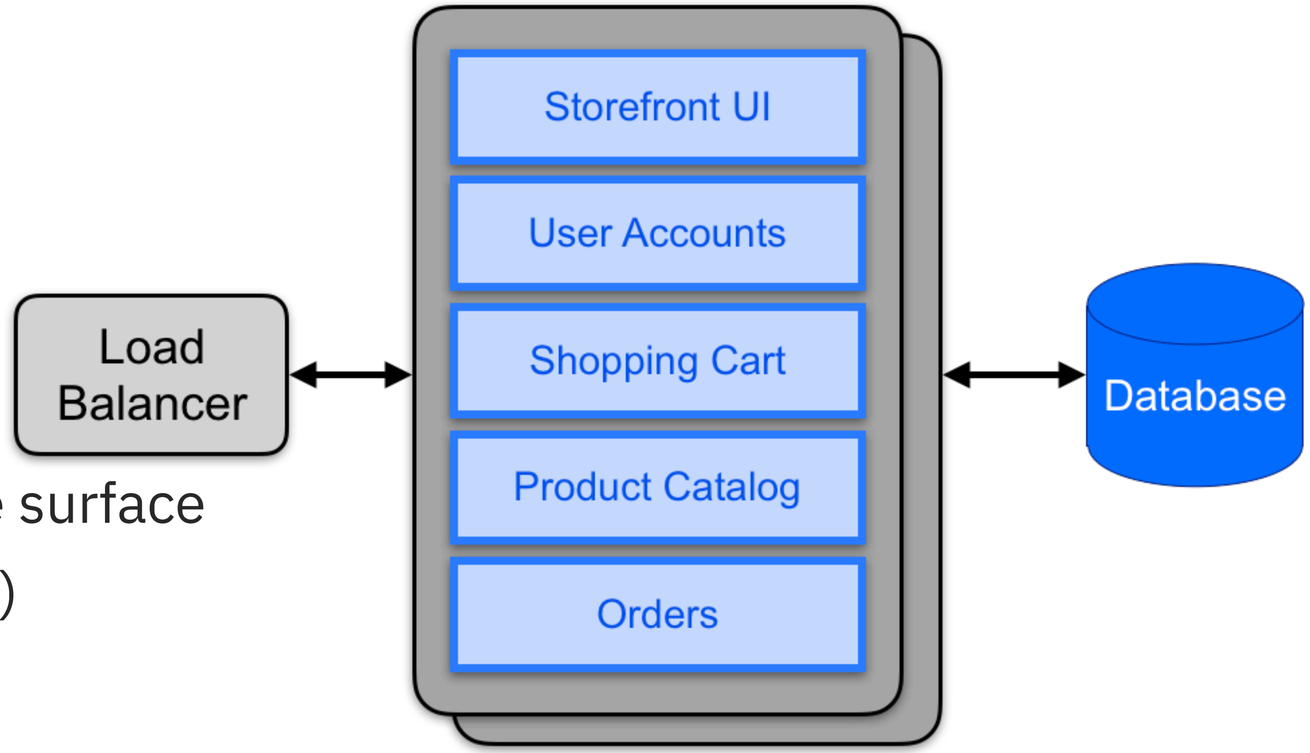
Monolithic Application Development

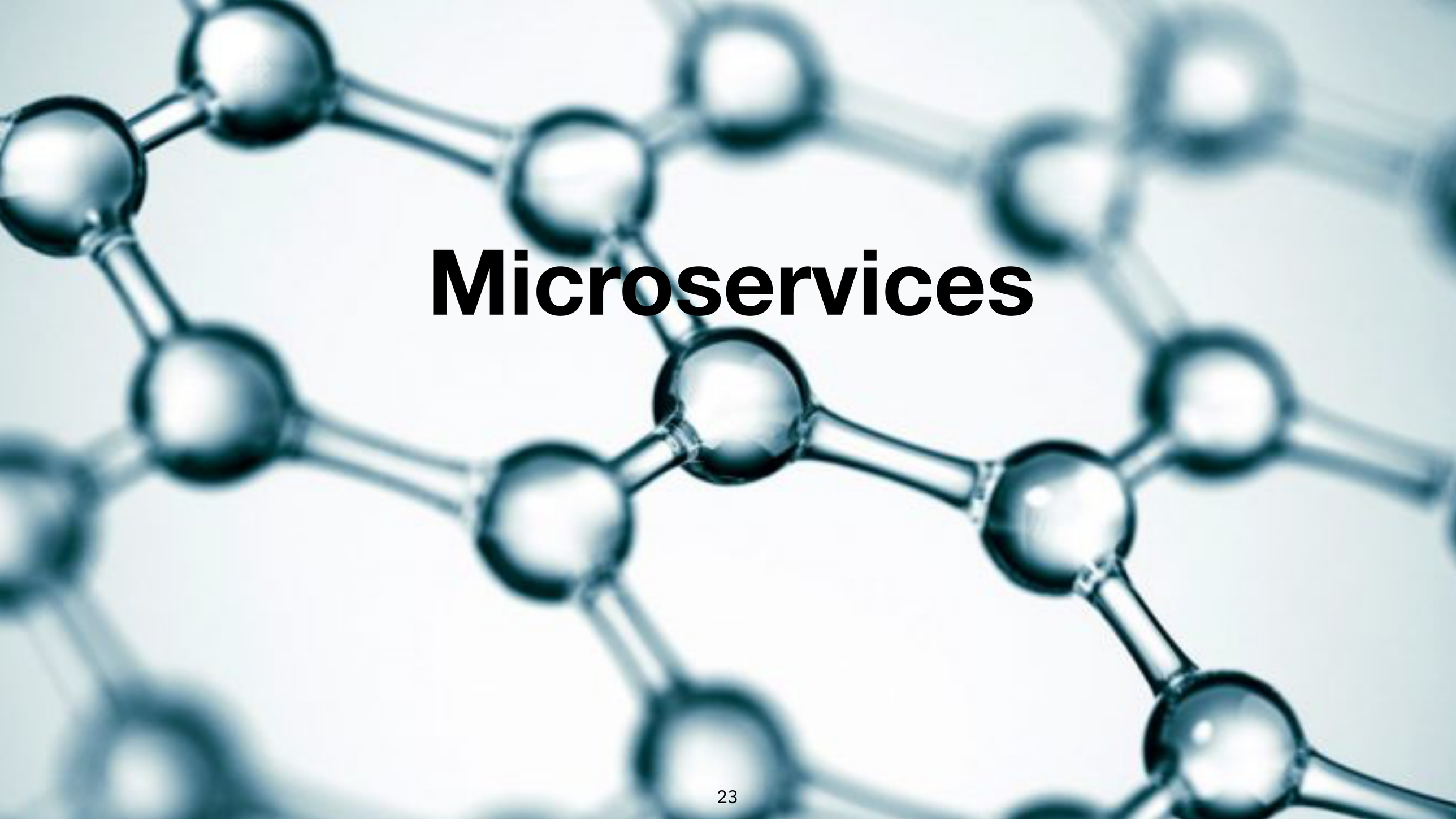
- Tightly coupled
- Mixed Concerns
- Large Deployment units
- Hard to Scale
- Long release cycles
- Slow on-boarding for new developers
- Slower feedback loop



Monolithic Application Deployment

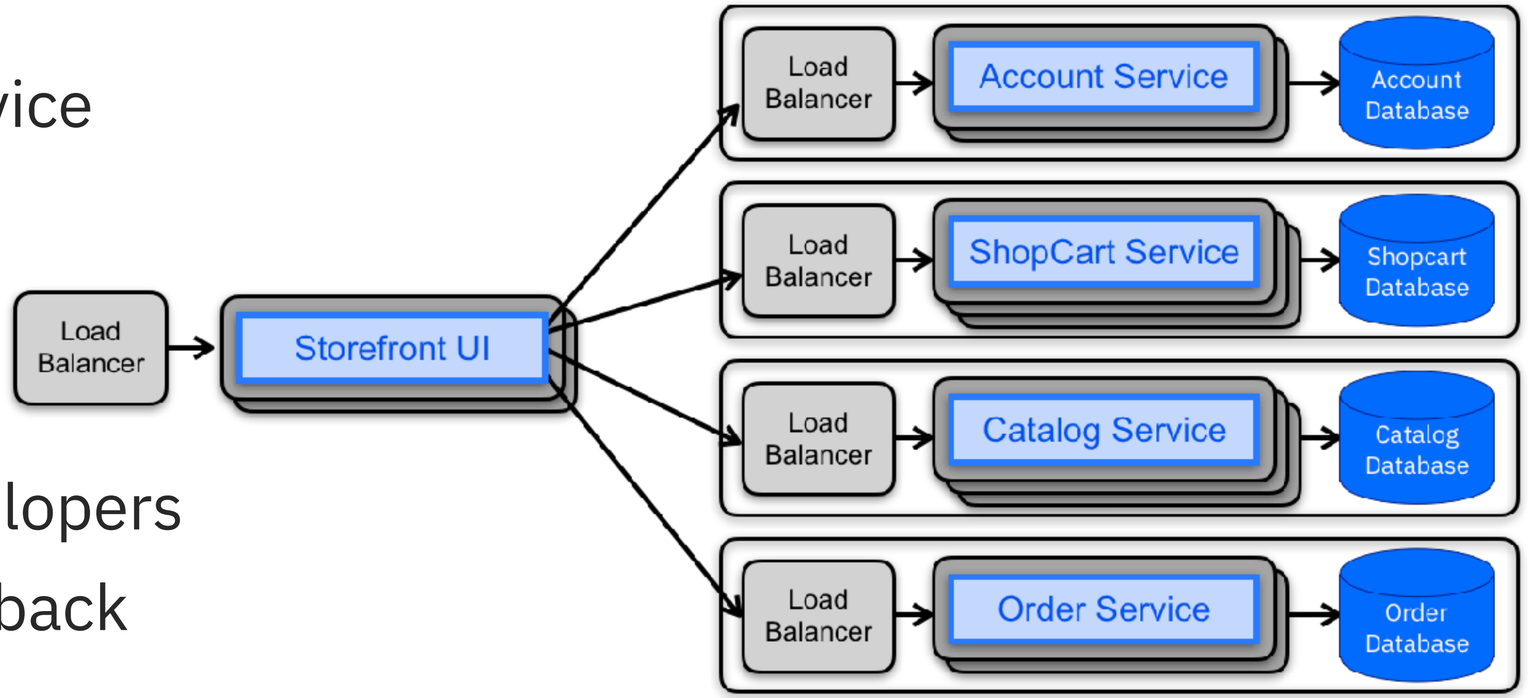
- Cloud Enabled Model (VM Centric)
  - Single monolithic image
  - Configuration: “Hand crafted”
  - Changes: Performed in Change windows
  - Deployment: All or nothing
- Implications for operations:
  - Single point of failure & large maintenance surface
  - Error prone (many different configurations)
  - Heavy on Human tasks




Microservices

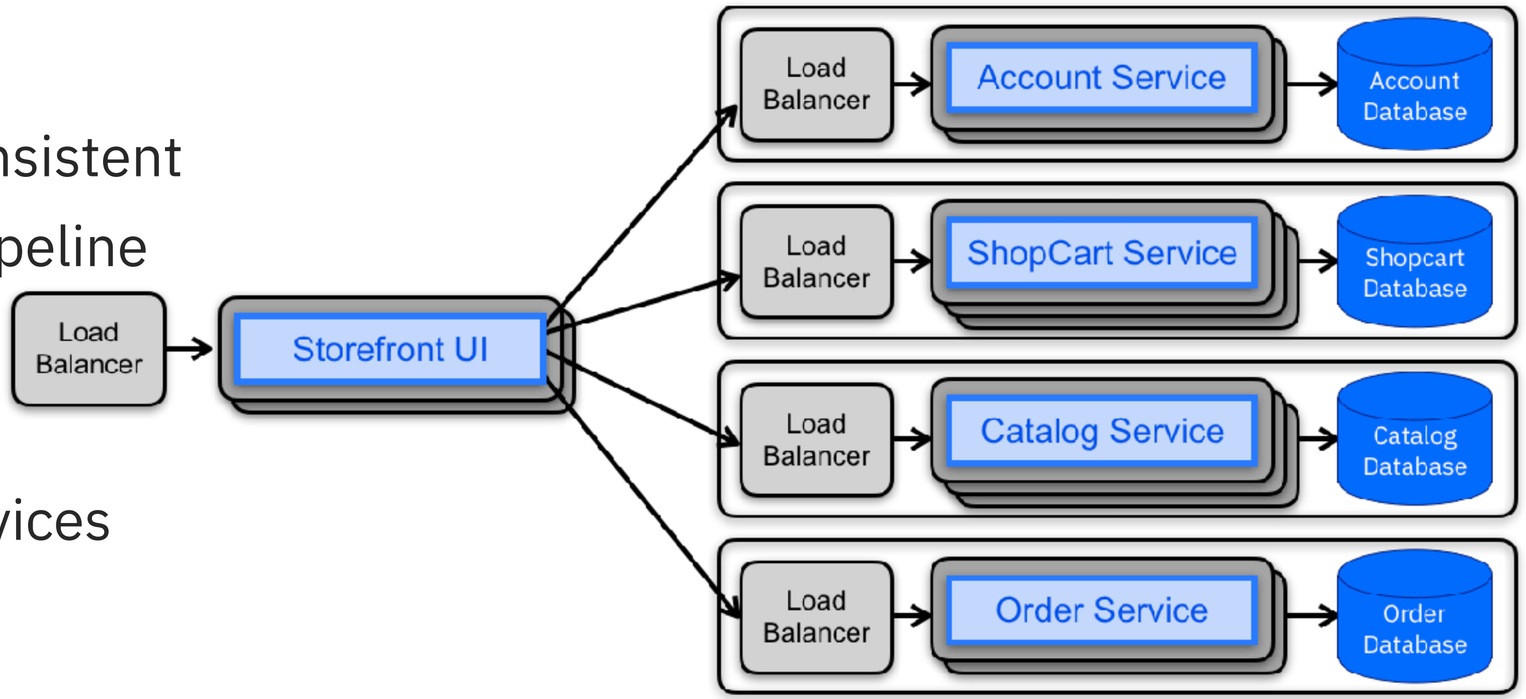
Microservice Application Development

- Loosely coupled
- Minimal responsibility per service
- Small Deployment units
- Easy to Scale
- Short release cycles
- Fast on-boarding for new developers
- Develop quickly with fast feedback



Microservice Application Deployment

- Cloud Native Model
  - Multiple microservices
  - Configuration: Automated and consistent
  - Changes: Performed in DevOps Pipeline
  - Deployment: Only what changes
- Advantages for operations
  - Resiliency through redundant services
  - Consistent configuration
  - Automated massive deployment



A Microservice Should Have

High Cohesion (Bounded Context around a Business Domain)

- Does stuff that needs to change together occur together?

Low Coupling (Shared Nothing with Technology Agnostic API)

- Do you avoid making otherwise independent concerns dependent?

Low Time to Comprehension (Small and Single Responsibility)

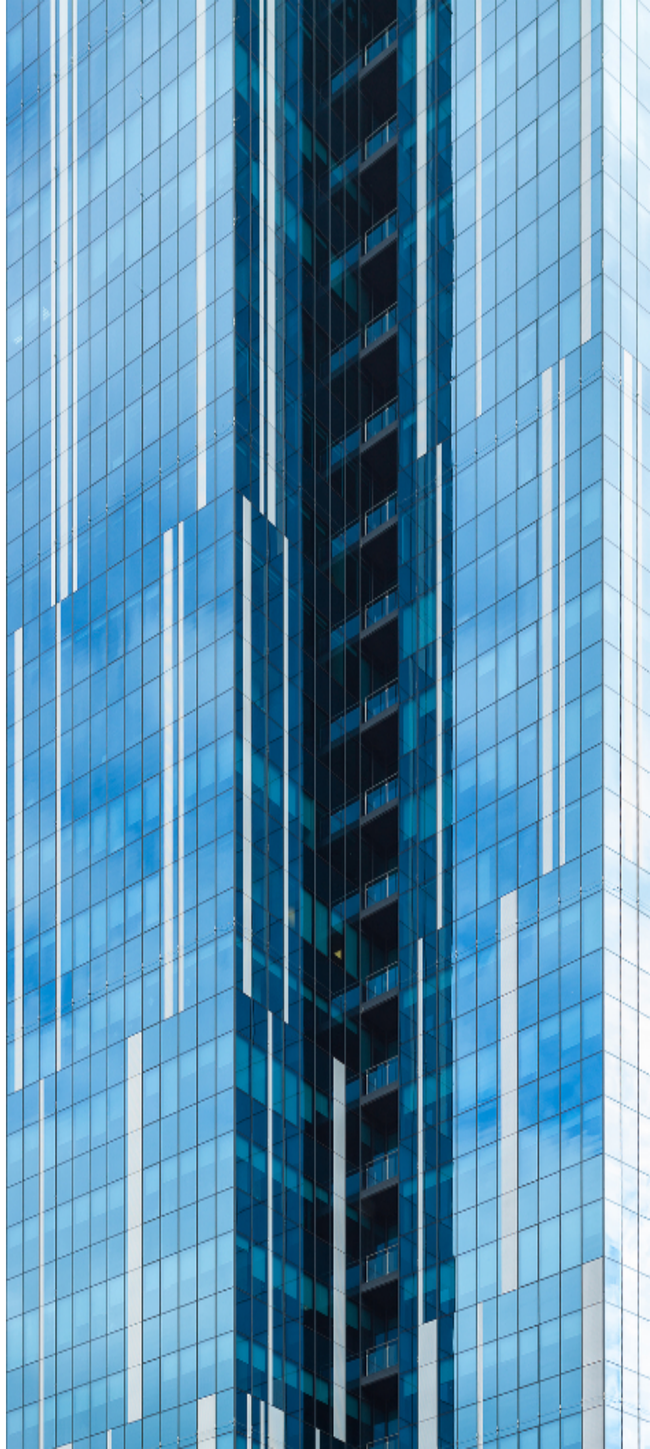
- Small enough for one person to understand quickly

Monolithic vs Microservices Architectures

Category	Monolithic architecture	Microservices architecture
Architecture	Built as a single logical executable	Built as a suite of small services
Modularity	Based on language features	Based on business capabilities
Agility	Changes to the system involve building and deploying a new version of the entire application	Changes can be applied to each service independently
Scaling	Entire application scaled when only one part is the bottleneck	Each service scaled independently when needed
Implementation	Typically entirely developed in one programming language	Each service can be developed in a different programming language
Maintainability	Large code base is intimidating to new developers	Smaller code bases easier to manage
Deployment	Complex deployments with maintenance windows and scheduled downtimes	Simple deployment as each service can be deployed individually, with minimal if not zero downtime

What makes a good Microservice?

- Microservices should have minimal outside dependancies
- Business Domains usually have well established boundaries
- Microservices should be broken up by Business Domains
  - With well defined interface
  - Limiting dependancies as much as possible

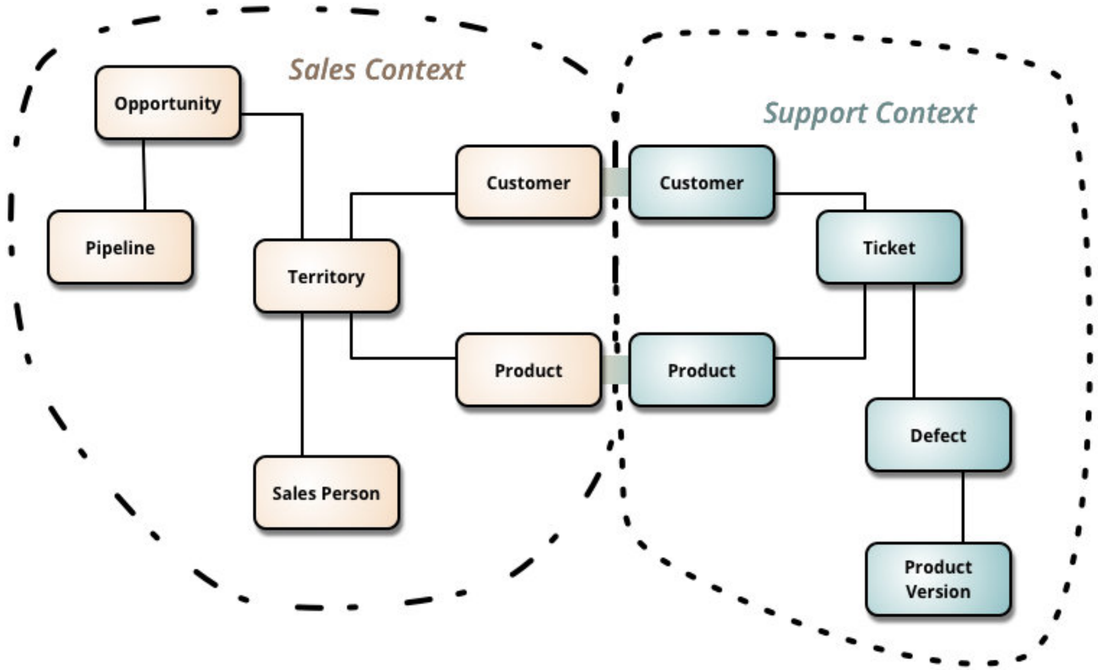


“Gather together those things that change for the same reason, and separate those things that change for different reasons.”

– Single Responsibility Principle by Robert C. Martin”

Domain Driven Design

- DDD is about designing software based on models of the underlying domain
- **Bounded Context** is a central pattern in Domain-Driven Design
- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



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

Microservices: Designing for Failure

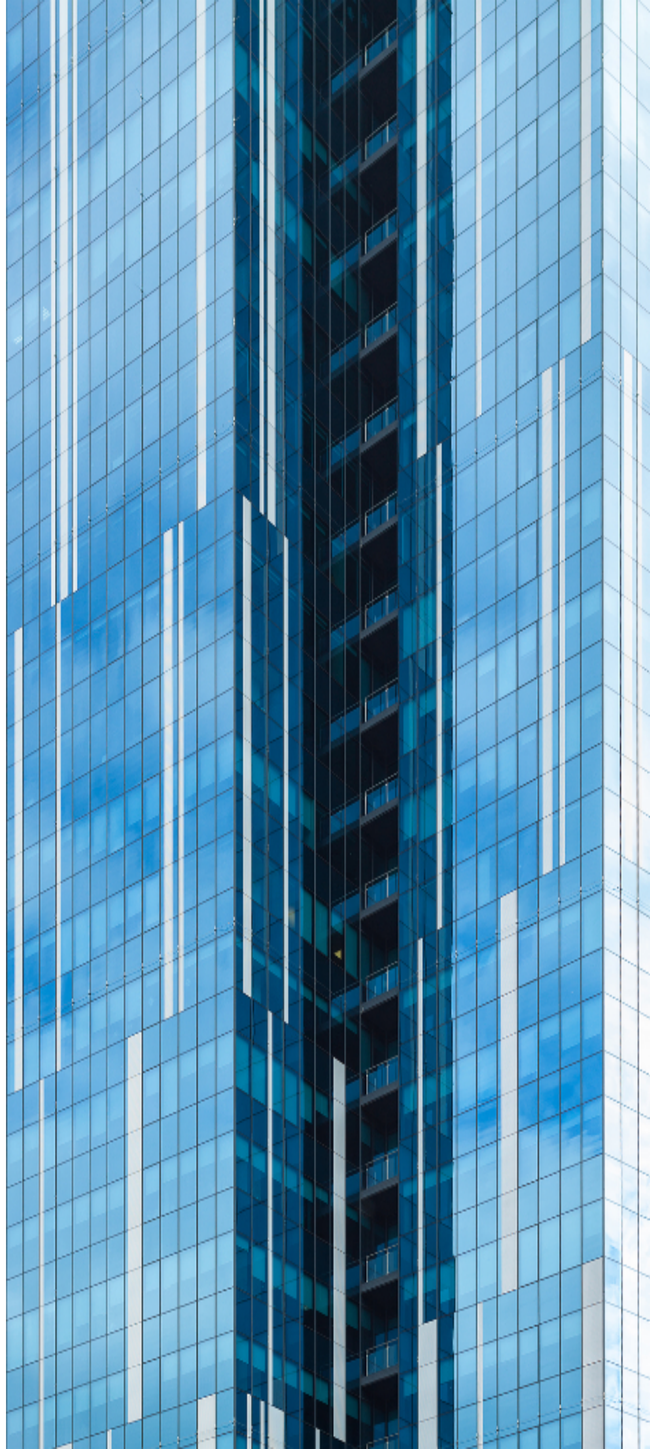

“The cloud is in a continuous state of partial failure”

“The cloud is in a continuous state of partial failure”

Does it matter if the customer never sees the failure?

Microservice Designs Must:

- Design for failure
  - How to avoid —> How to identify & what to do about it
  - Pure operational concern —> developer concern
- Plan to be throttled
- Plan to retry (with exponential backoff)
- Degrade gracefully
- Cache when appropriate



Retry Pattern

- Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation.
- Exponentially back-off delaying longer with each retry

1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

Retry Pattern Example in Python

```
def retry(ExceptionToCheck, tries=4, delay=3, backoff=2, logger=None):
    """ Retry calling the decorated function using an exponential backoff. """
    def decorator(f):

        @wraps(f)
        def wrapper(*args, **kwargs):
            max_tries, max_delay = tries, delay
            while max_tries > 1:
                try:
                    return f(*args, **kwargs)
                except ExceptionToCheck, e:
                    msg = "%s, Retrying in %d seconds..." % (str(e), max_delay)
                    if logger:
                        logger.warning(msg)
                    else:
                        print msg
                    time.sleep(max_delay)  # wait before trying again
                    max_tries -= 1         # decrement retry count
                    max_delay *= backoff   # exponential delay more and more
            return f(*args, **kwargs)

        return wrapper  # true decorator

    return decorator
```

Retry Pattern Example in Python

```
def retry(ExceptionToCheck, tries=4, delay=3, backoff=2, logger=None):
    """ Retry calling the decorated function using an exponential backoff. """
    def decorator(f):

        @wraps(f)
        def wrapper(*args, **kwargs):
            max_tries, max_delay = tries, delay
            while max_tries > 1:
                try:
                    return f(*args, **kwargs)
                except ExceptionToCheck, e:
                    msg = "%s, Retrying in %d seconds..." % (str(e), max_delay)
                    if logger:
                        logger.warning(msg)
                    else:
                        print msg
                    time.sleep(max_delay)  # wait before trying again
                    max_tries -= 1         # decrement retry count
                    max_delay *= backoff   # exponential delay more and more
            return f(*args, **kwargs)

        return wrapper  # true decorator

    return decorator
```

Heart of the retry logic

Using Python retry package

- For example if are getting: `HTTP 429 Client Error: Too Many Requests` with Cloudant, you can use the retry package

```
from retry import retry

    @retry(HTTPError, delay=1, backoff=2, tries=5)
    def delete(self):
        """ Removes a document from the database """
        try:
            document = self.database[self.id]
        except KeyError:
            document = None
        if document:
            document.delete()
```

This will catch any `HTTPError` and delay 1 second (`delay=1`), then 2 seconds, then 4 seconds, etc. (`backoff=2` by 2 seconds each time) until 5 tries are reached (`tries=5`)

Using Python retry package

- For example if are getting: `HTTP 429 Client Er` `ests` with Cloudant, you can use the retry package

Retry Decorator add to delete() function

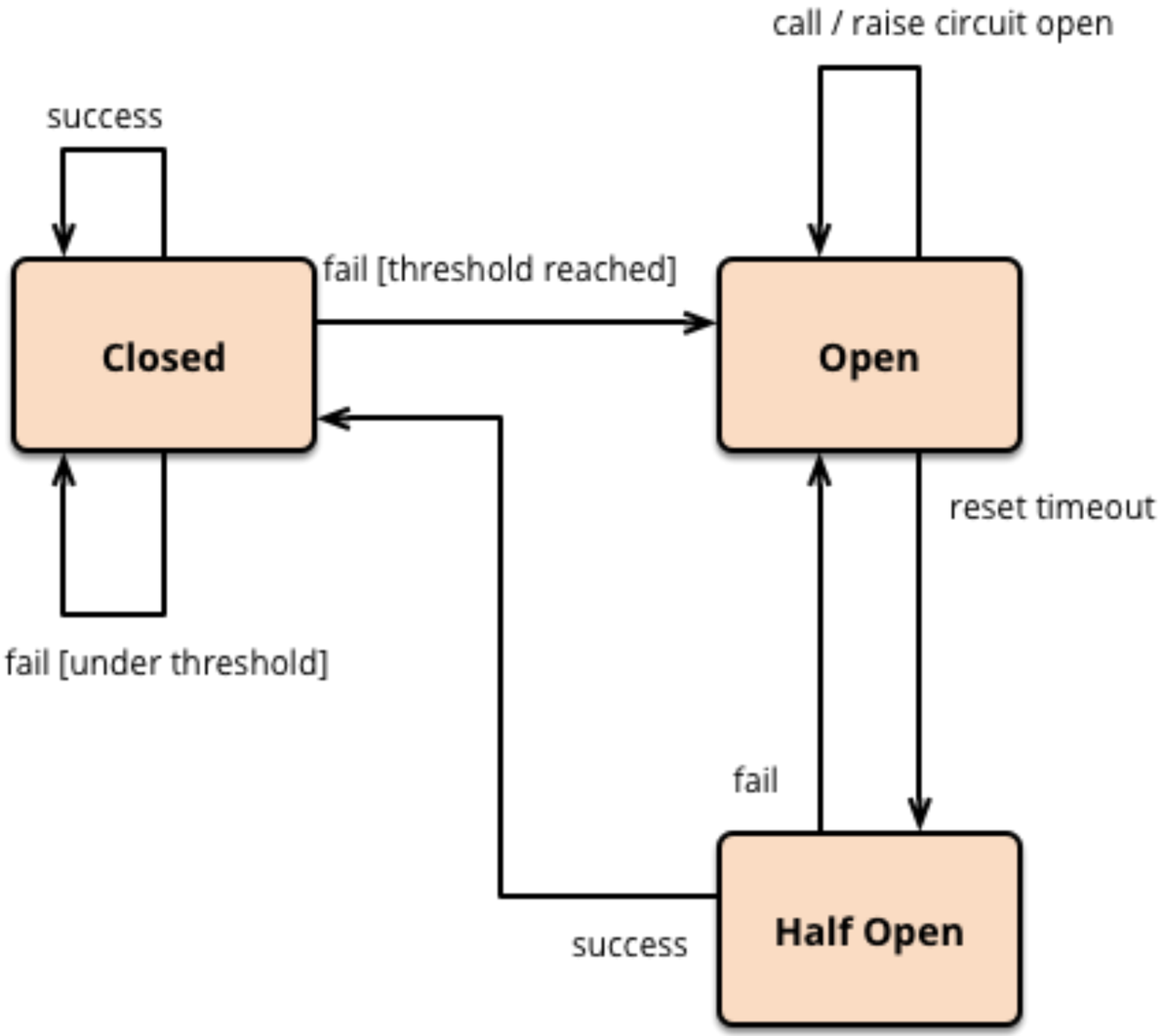
```
from retry import retry

    @retry(HTTPError, delay=1, backoff=2, tries=5)
    def delete(self):
        """ Removes a document from the database """
        try:
            document = self.database[self.id]
        except KeyError:
            document = None
        if document:
            document.delete()
```

This will catch any `HTTPError` and delay 1 second (`delay=1`), then 2 seconds, then 4 seconds, etc. (`backoff=2` by 2 seconds each time) until 5 tries are reached (`tries=5`)

Circuit Breaker Pattern

- You wrap a protected function call in a circuit breaker object, which monitors for failures.
- Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.
- Usually you'll also want some kind of monitor alert if the circuit breaker trips.



https://martinfowler.com/bliki/CircuitBreaker.html

Circuit Breaker Package in Python

PyPi has a library called circuitbreaker that you can install

```
$ pip install circuitbreaker
```

Just import the decorator function and use it to wrap your function

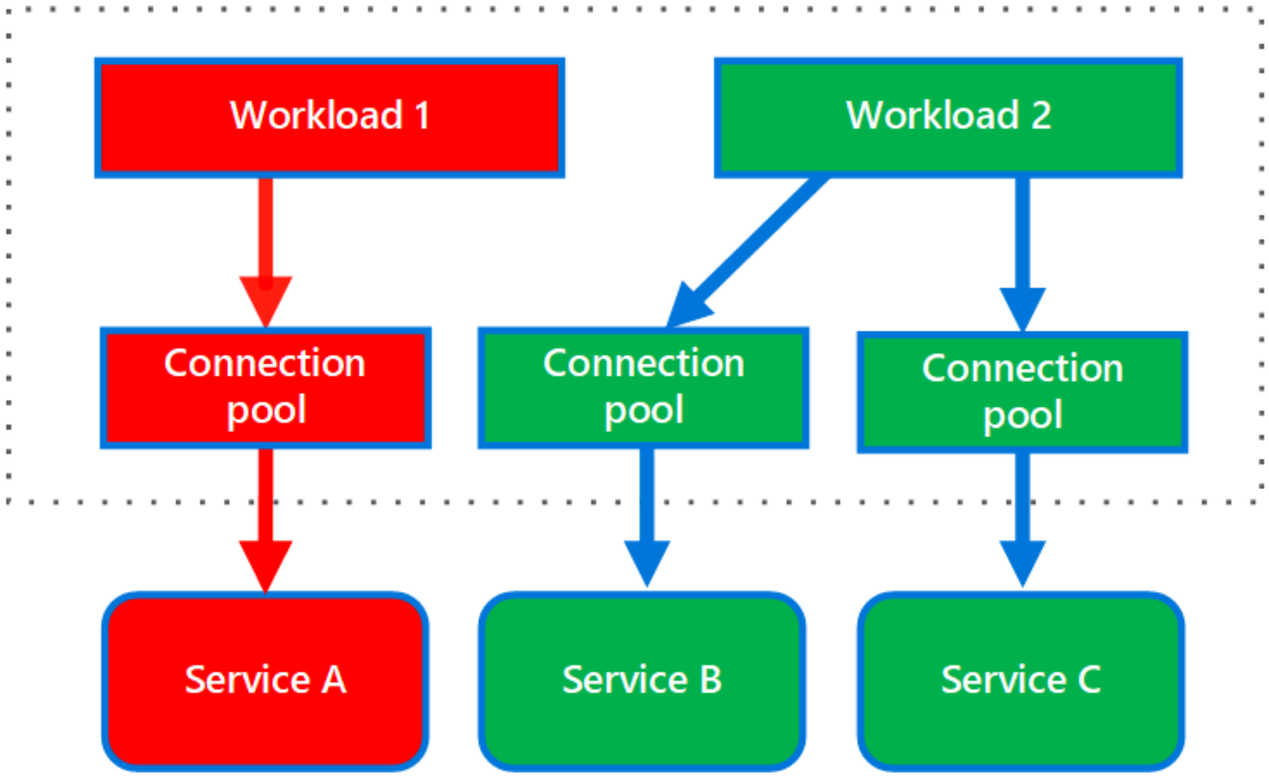
```
from circuitbreaker import circuit

@circuit(failure_threshold=10, expected_exception=ConnectionError)
def external_call():
    ...
```


https://pypi.org/project/circuitbreaker/

Bulkhead Pattern

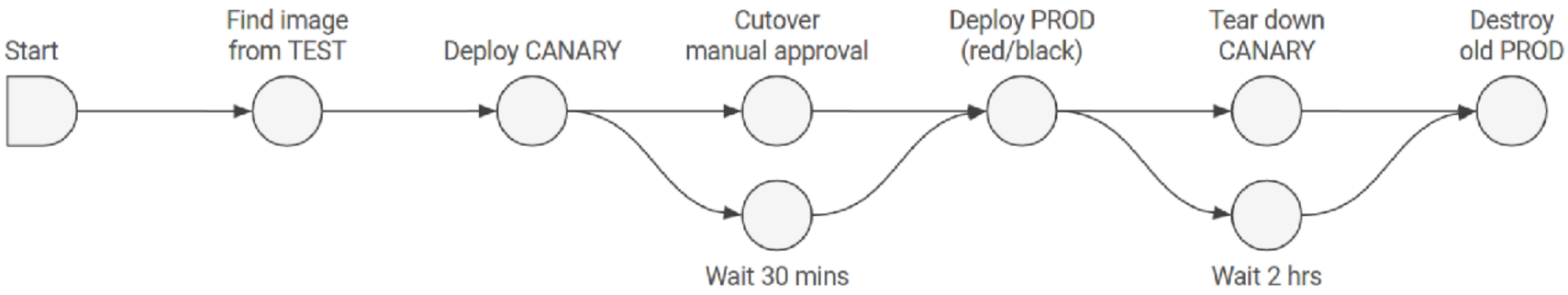
- Isolates consumers and services from cascading failures.
  - An issue affecting a consumer or service can be isolated within its own bulkhead, preventing the entire solution from failing.
- This pattern is named Bulkhead because it resembles the sectioned partitions of a ship's hull
  - If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.
- Allows you to preserve some functionality in the event of a service failure. Other services and features of the application will continue to work.

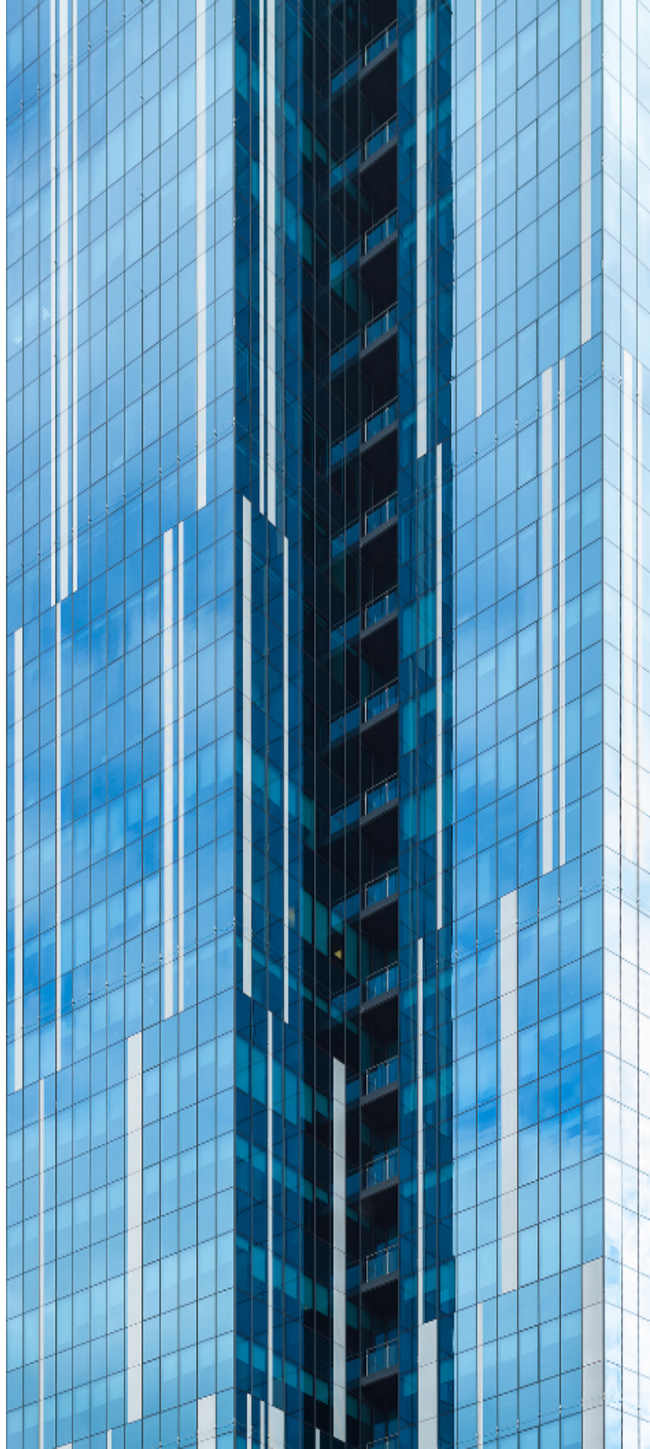


https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead

Canary Testing

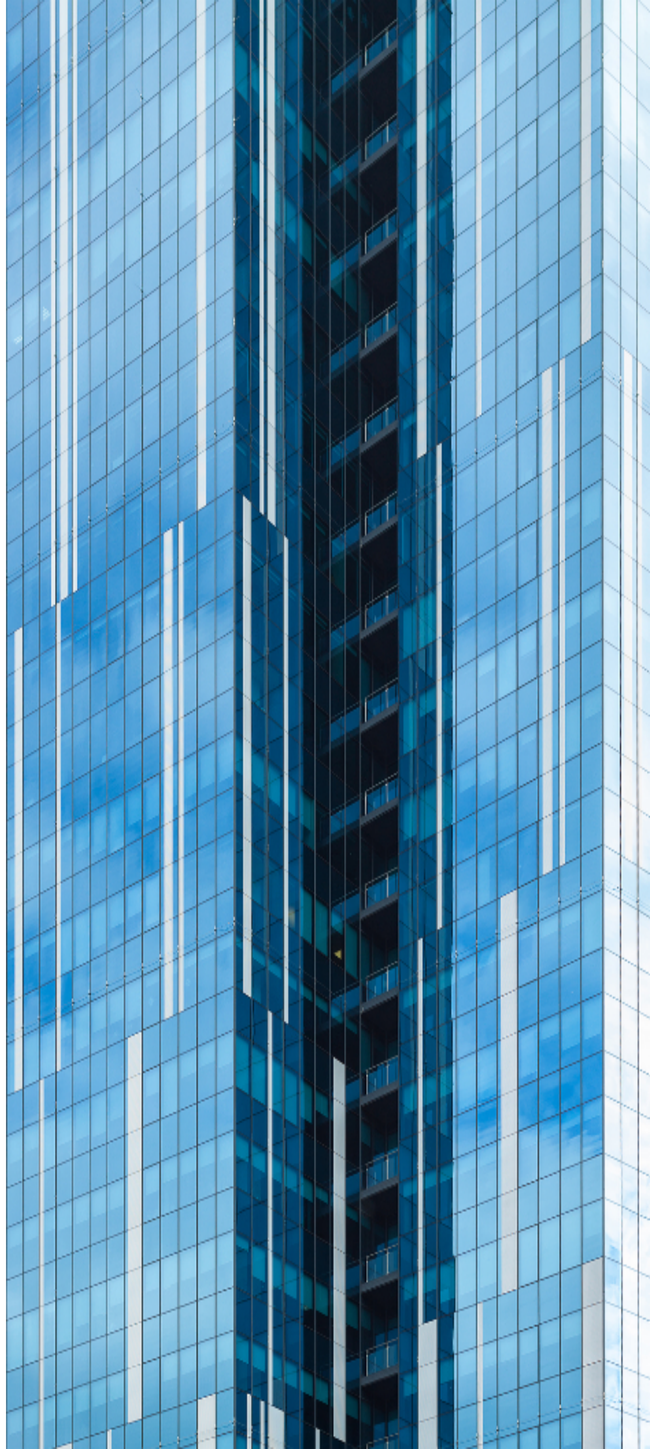
- Mitigates the risks of changes to applications in production
- Features are activated only for a small percentage of users, and the application performance and adoption results are measured
  - If those results indicate that the change is good, then it is ramped up to the rest of the user population
  - If the change is not good, it is rolled back





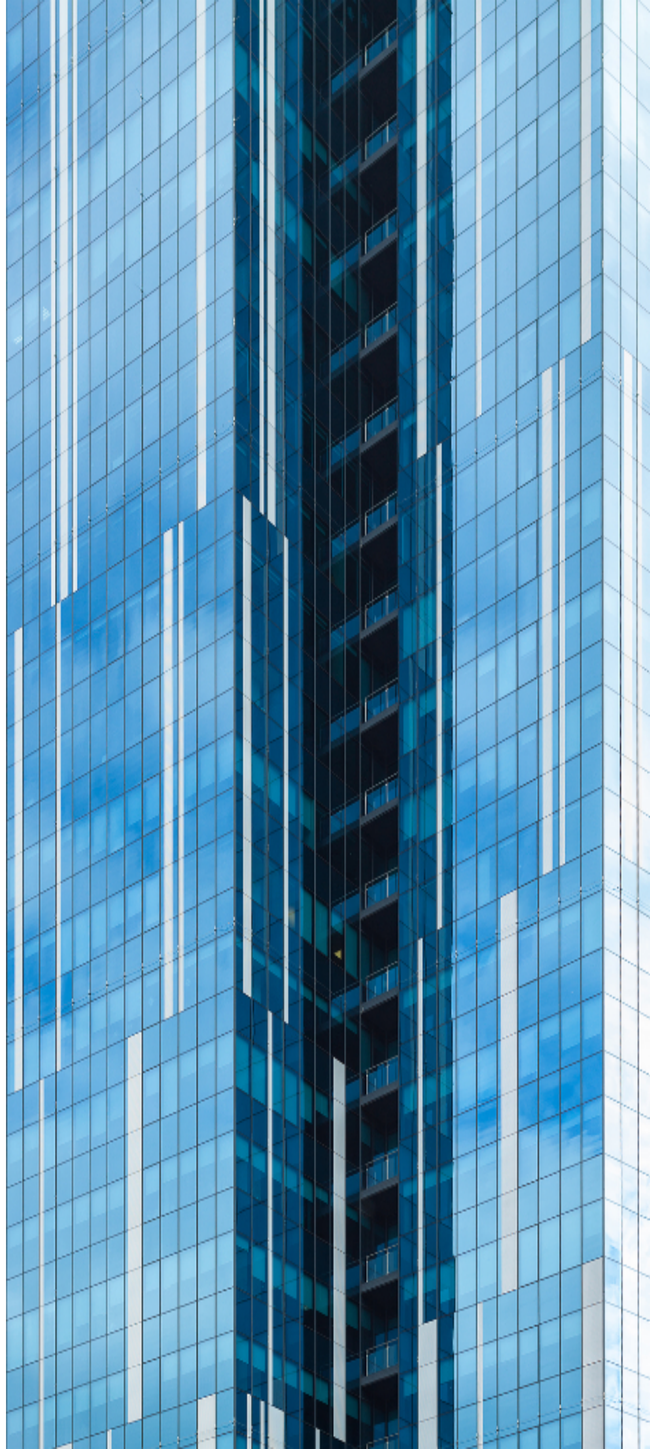
A/B Testing

- Some set of users are directed to one implementation of a feature, let’s call it the A version, while a different set of users are directed to a different implementation of the feature, let’s call that the B version
- This allows DevOps teams to evaluate different implementation options for a feature, and pick the one that works the best in the field, by measuring actual usage
- The data from usage analytics is then used to influence the priorities of the remaining stories in the backlog.



Feature Flags

- With Continuous Delivery, code can be tested and delivered into production with “Feature Flags” around the new code
- Using these techniques, those features can be made visible to specific audiences to allow testing of those features in a production environment
- The act of going Live can happen via a Release Event on a later date when the Feature Flags are turned on in a coordinated way across multiple components, and the new feature is made publicly visible.



Practice:

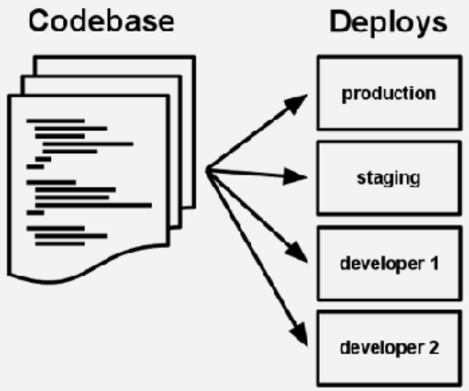
12-factor Methodology

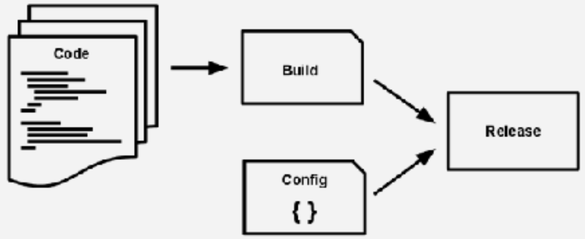
The methodology was drafted by developers at Heroku and was first presented by Adam Wiggins circa 2011

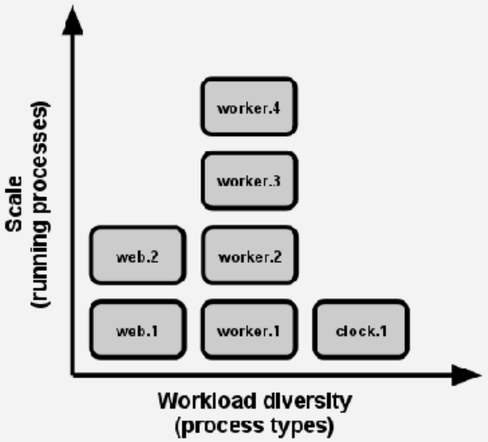
The Twelve-Factor App

- The twelve-factor app is a methodology for building software-as-a-service apps that:
  - Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
  - Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
  - Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
  - **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
  - And can **scale up** without significant changes to tooling, architecture, or development practices.

12-factor Methodology







I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

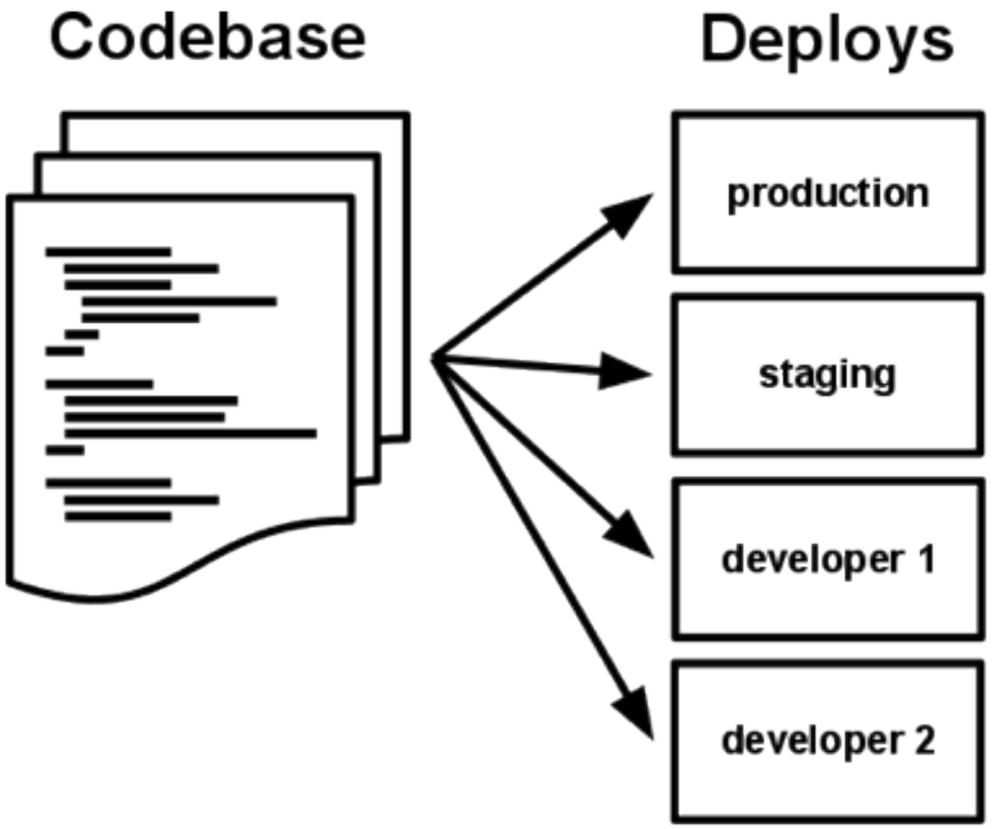
XII. Admin processes
Run admin/management tasks as one-off processes

https://12factor.net/

I. Codebase

One codebase tracked in revision control, many deploys

- **There is always a one-to-one correlation between the codebase and the app**
- If there are multiple codebases, it's not an app – it's a distributed system.
  - Each component in a distributed system is an app, and each can individually comply with twelve-factor.
- Multiple apps sharing the same code is a violation of twelve-factor.
  - The solution here is to factor shared code into libraries which can be included through the dependency manager.



II. Dependencies

Explicitly declare and isolate dependencies

- Most programming languages offer a packaging system for distributing support libraries, such as Rubygems for Ruby, or PyPi for Python, or Maven for Java
  - Libraries installed through a packaging system can be installed system-wide (known as “site packages”) or scoped into the directory containing the app (known as “vendoring” or “bundling”)
- **A twelve-factor app never relies on implicit existence of system-wide packages**
  - It declares all dependencies, completely and exactly, via a dependency declaration manifest
  - Furthermore, it uses a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development

III. Config

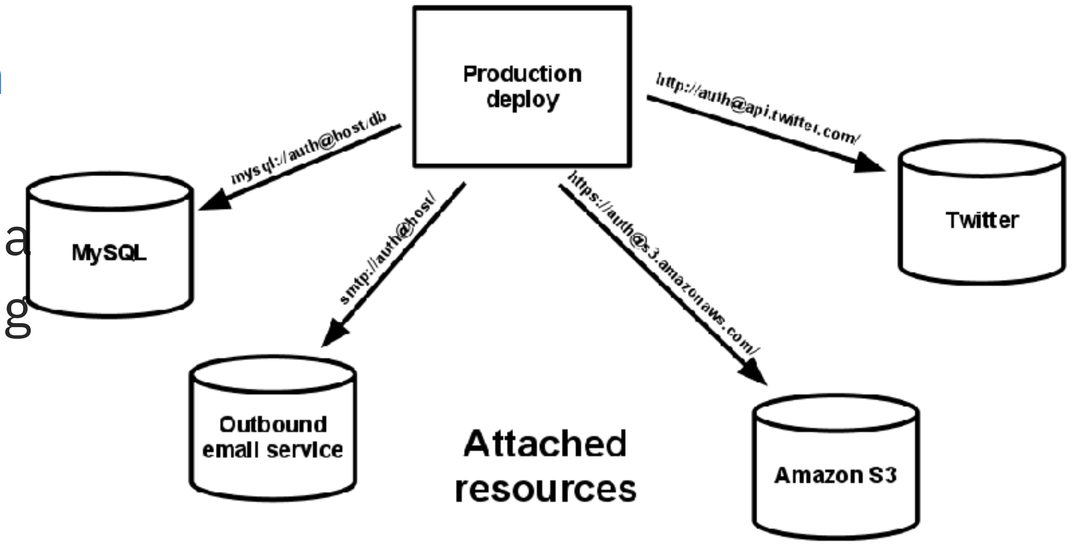
Store config in the environment

- 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 or Twitter
  - Per-deploy values such as the canonical hostname for the deploy
- Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which **requires strict separation of config from code**. Config varies substantially across deploys, code does not.

IV. Backing services

Treat backing services as attached resources

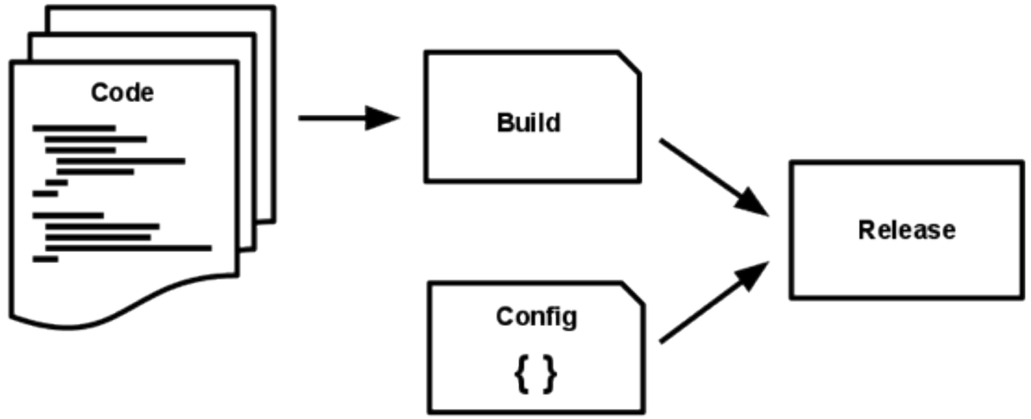
- A backing service is any service the app consumes over the network as part of its normal operation
- **The code for a twelve-factor app makes no distinction between local and third party services**
  - To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config
  - A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code



V. Build, release, run

Strictly separate build and run stages

- A codebase is transformed into a (non-development) deploy through three stages:
  - The **build** stage is a transform which converts a code repo into an executable bundle known as a build.
  - The **release** stage takes the build produced by the build stage and combines it with the deploy's current config.
  - The **run** stage (also known as "runtime") runs the app in the execution environment, by launching some set of the app's processes against a selected release.



- **The twelve-factor app uses strict separation between the build, release, and run stages**
  - For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.

VI. Processes

Execute the app as one or more stateless processes

- The app is executed in the execution environment as one or more processes
- In the simplest case, the code is a stand-alone script, the execution environment is a developer's local laptop with an installed language runtime, and the process is launched via the command line (for example, python my_script.py)
- On the other end of the spectrum, a production deploy of a sophisticated app may use many process types, instantiated into zero or more running 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.

VII. Port binding

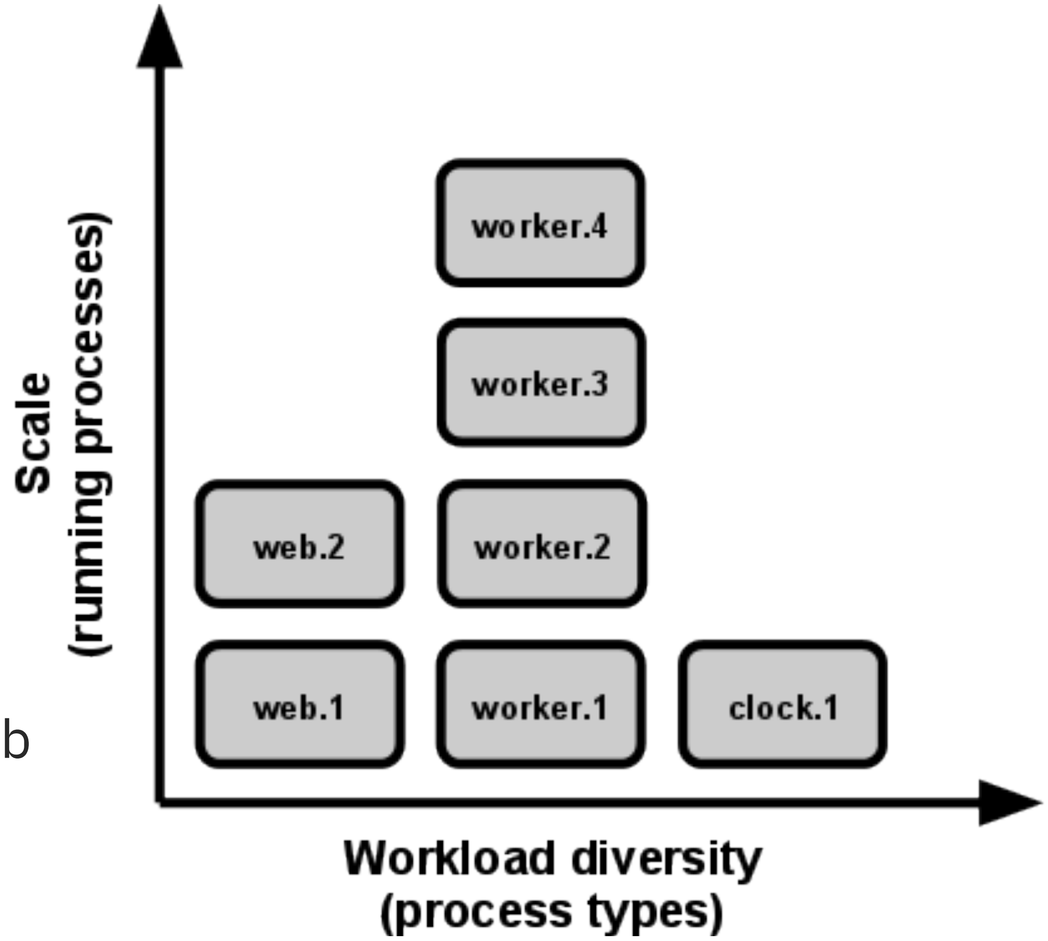
Export services via port binding

- **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.
- In a local development environment, the developer visits a service URL like `http://localhost:5000/` to access the service exported by their app
- In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

VIII. Concurrency

Scale out via the process model

- **In the twelve-factor app, processes are a first class citizen**
- Processes in the twelve-factor app take strong cues from the unix process model for running service daemons
- Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a process type
  - For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.



IX. 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
  - 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.
  - For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting..

X. Dev/prod parity

Keep development, staging, and production as similar as possible

- Historically, there have been substantial gaps between development (a developer making live edits to a local deploy of the app) and production (a running deploy of the app accessed by end users). These gaps manifest in three areas:
  - 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.

X. Dev/prod parity (continued)

Keep development, staging, and production as similar as possible

- **The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.**
- Looking at the three gaps described on the previous chart:
  - 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.

XI. Logs

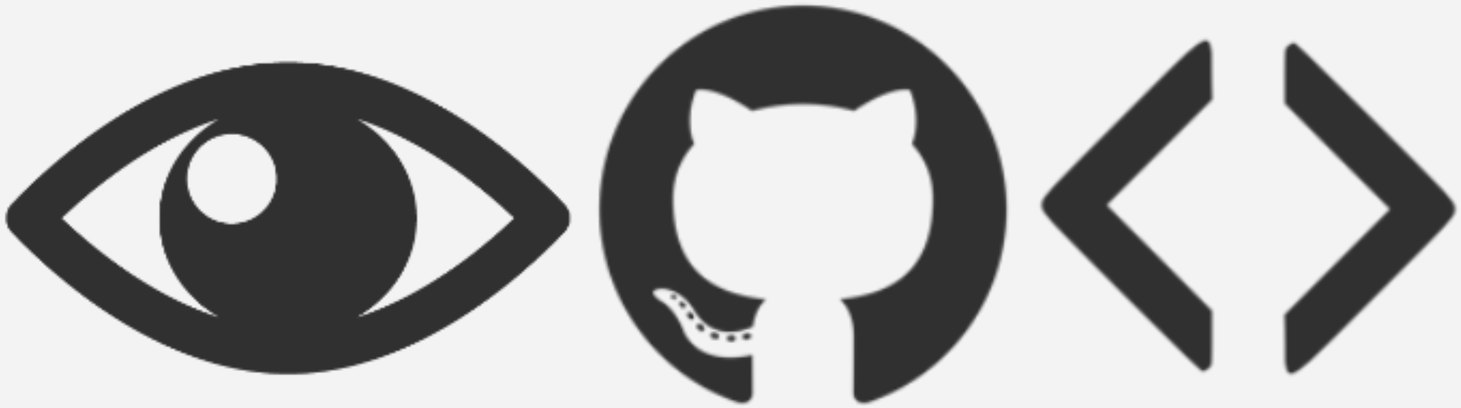
Treat logs as event streams

- Logs provide visibility into the behavior of a running app. In server-based environments they are commonly written to a file on disk (a “logfile”); but this is only an output format.
- **A twelve-factor app never concerns itself with routing or storage of its output stream**
  - It should not attempt to write to or manage logfiles
  - Instead, each running process writes its event stream, unbuffered, to stdout
  - During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior

XII. Admin processes

Run admin/management tasks as one-off processes

- **One-off admin processes should be run in an identical environment as the regular long-running processes of the app**, such as:
  - 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)
- They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.



Let's look at some 12-Factor Code!