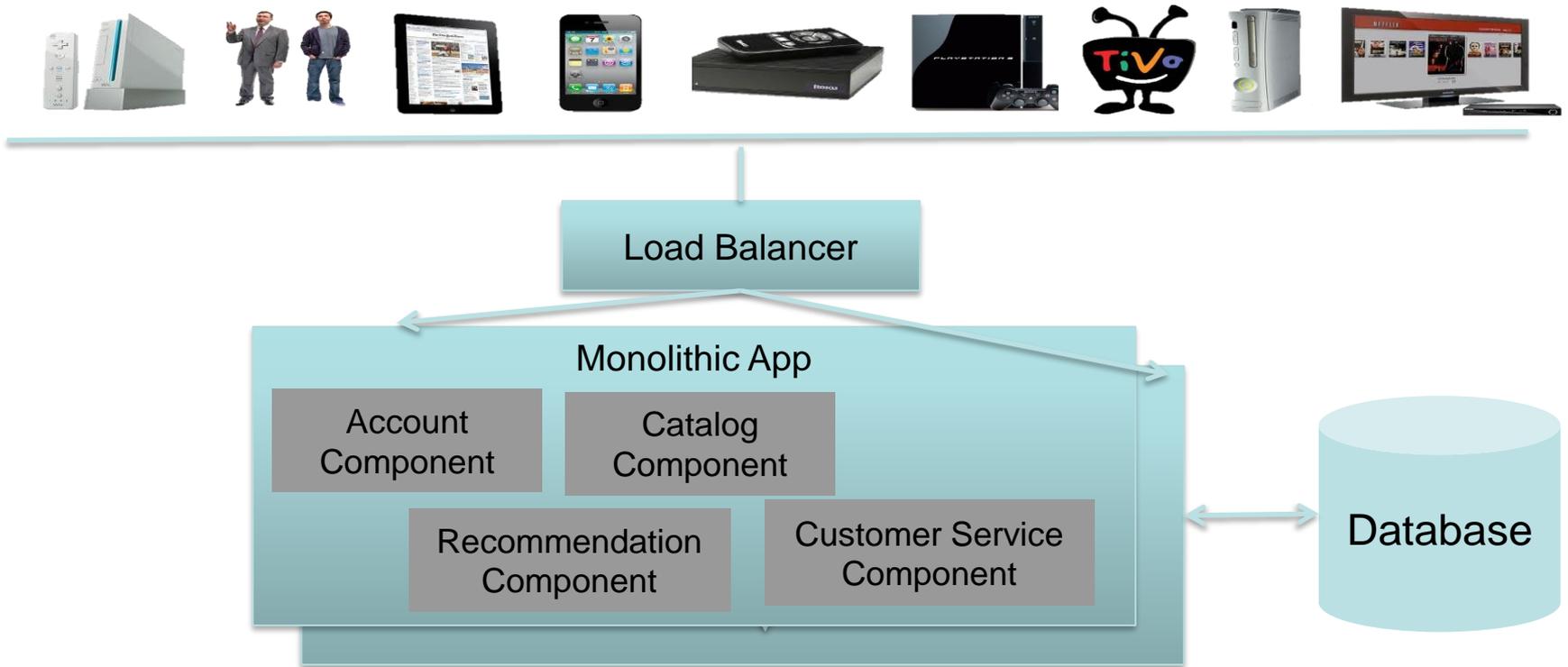


Software Systems Architecture

Monolith, SOA, Micro-services

Poștaru Andrei

Monolithic Architecture



Monolithic Architecture

- Large Codebase
- Many Components, no clear ownership
- Long deployment cycles

Monolithic Architecture

Pros

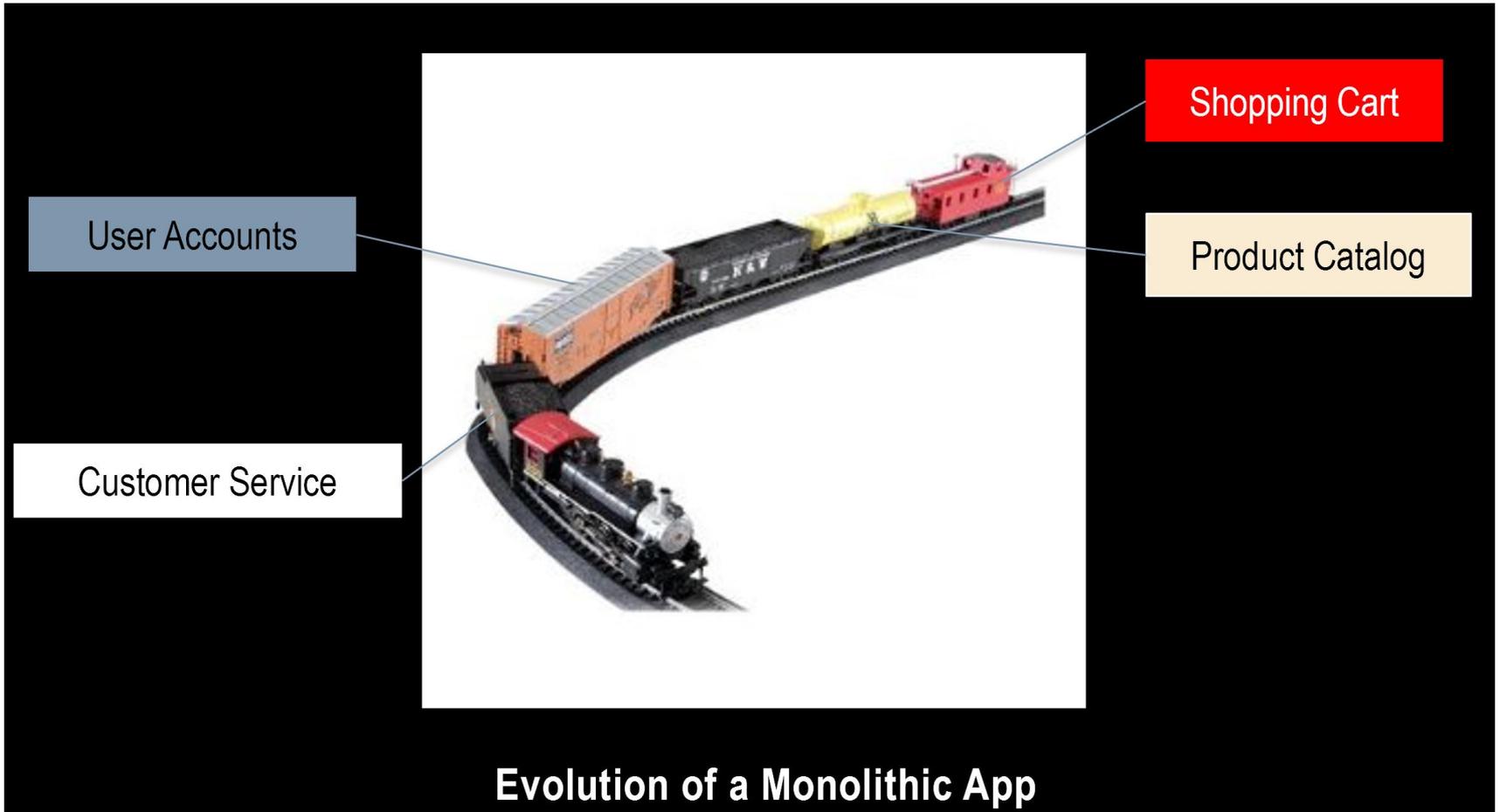
- Single codebase
 - Easy to develop/debug/deploy
 - Good IDE support
- Easy to scale horizontally (but can only scale in an “undifferentiated” manner)
- A Central Ops team can efficiently handle

Monolithic Architecture

Monolithic App – Evolution

- As codebase increases ...
 - Tends to increase “tight coupling” between components
 - Just like the cars of a train
 - All components have to be coded in the same language

Monolithic Architecture



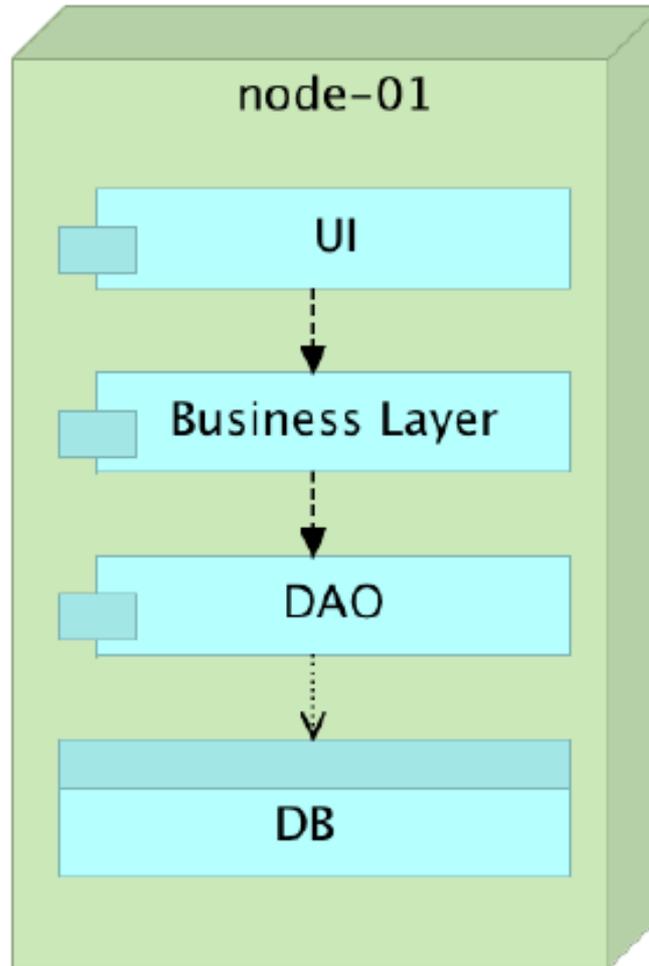
Monolithic Architecture

Monolithic App - Scaling

- Scaling is “undifferentiated”
 - Cant scale “Product Catalog” differently from “Customer Service”

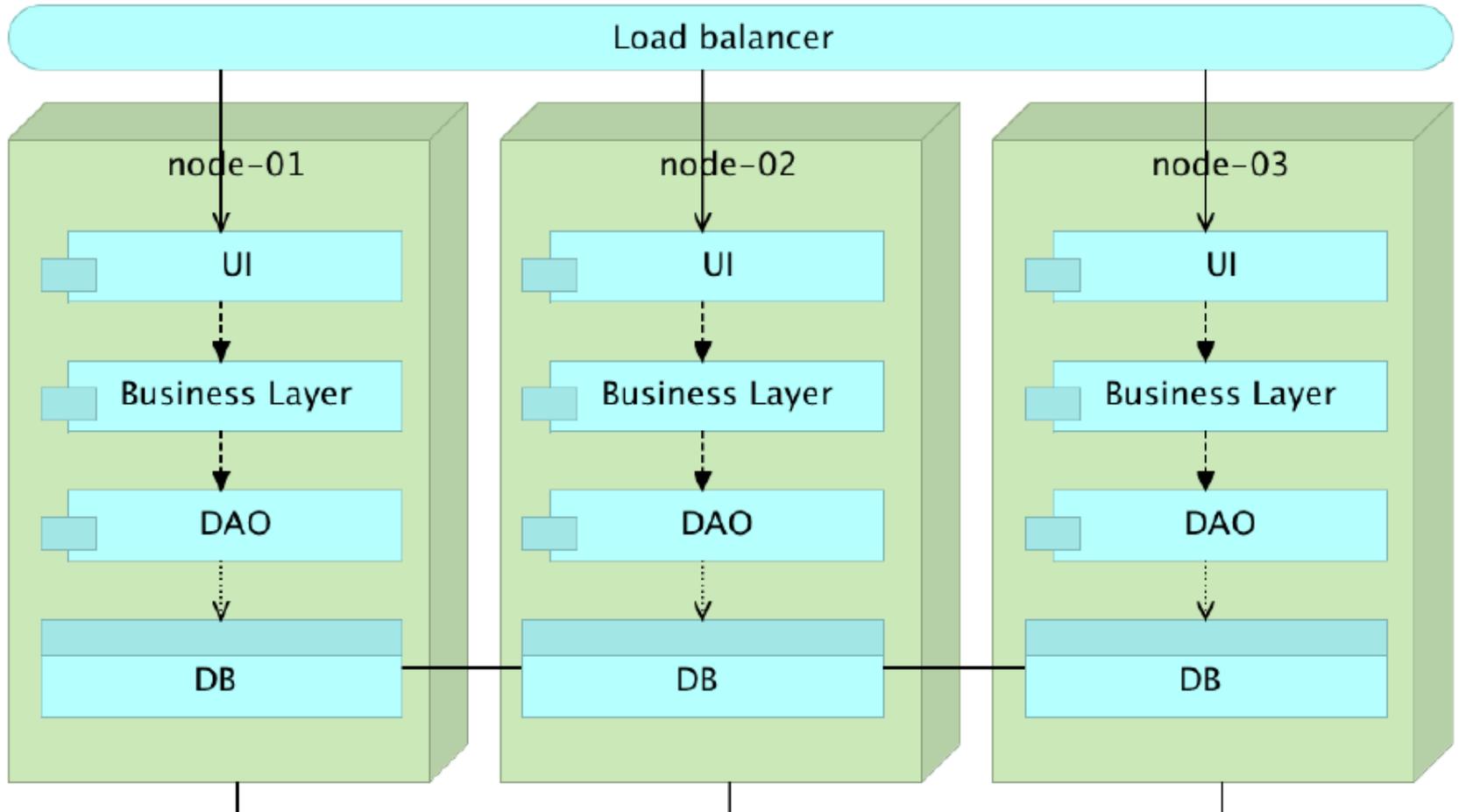


Monolithic Architecture



When an application is relatively small, splitting it into horizontal layers is a good idea. It provides a separation that makes development faster and easier as well as a separation based on type of the task code should do.

Monolithic Architecture



Scaling monolithic applications is very resource inefficient since everything needs to be duplicated on multiple nodes. There is no option to detect bottlenecks and scale or separate them from the rest of the application.

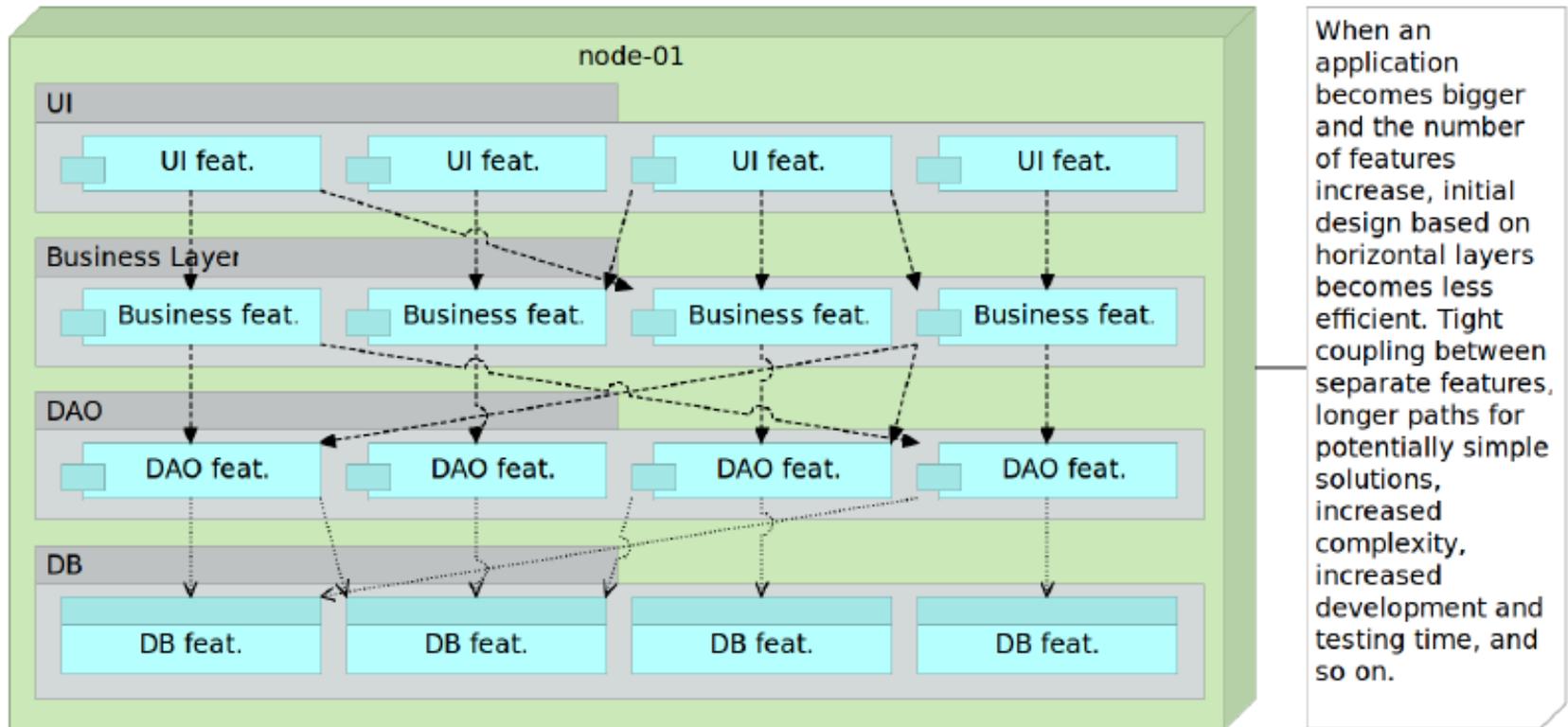
Monolithic Architecture



Monolithic Apps – Failure & Availability

Monolithic Architecture

Monolithic Application with Increased Number of Features



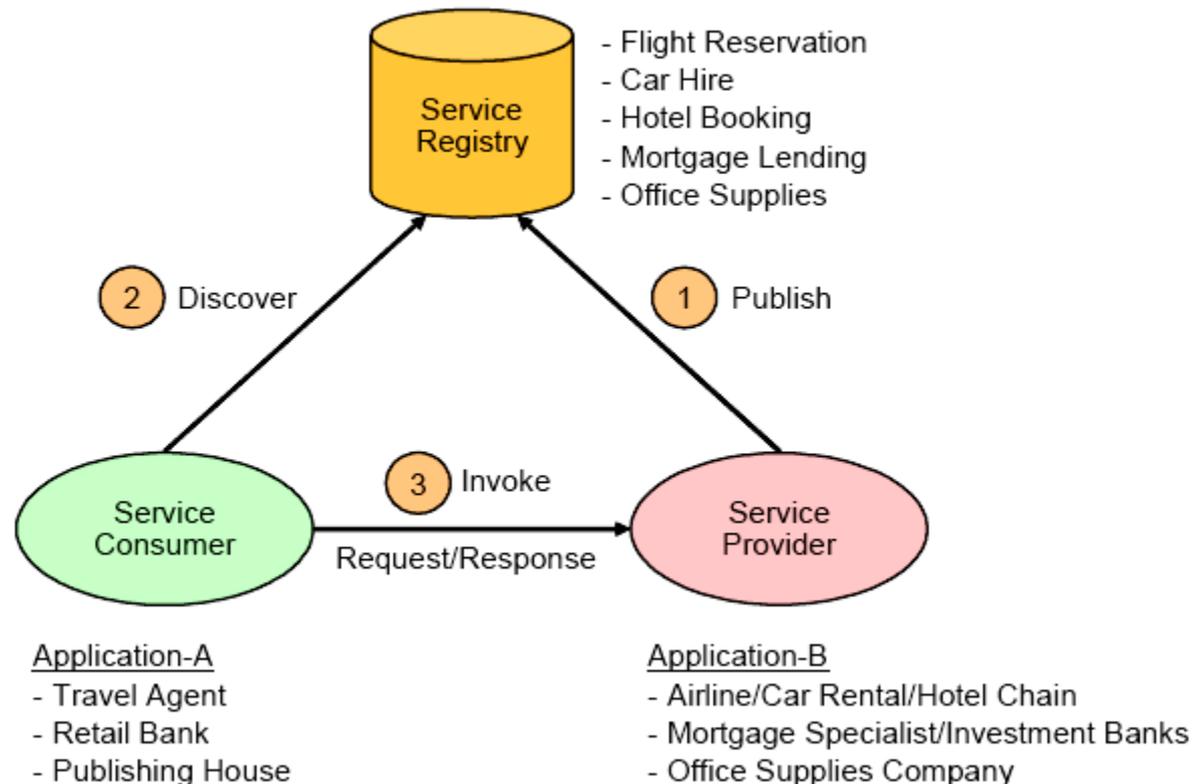
Service Oriented Architecture (SOA)

Service

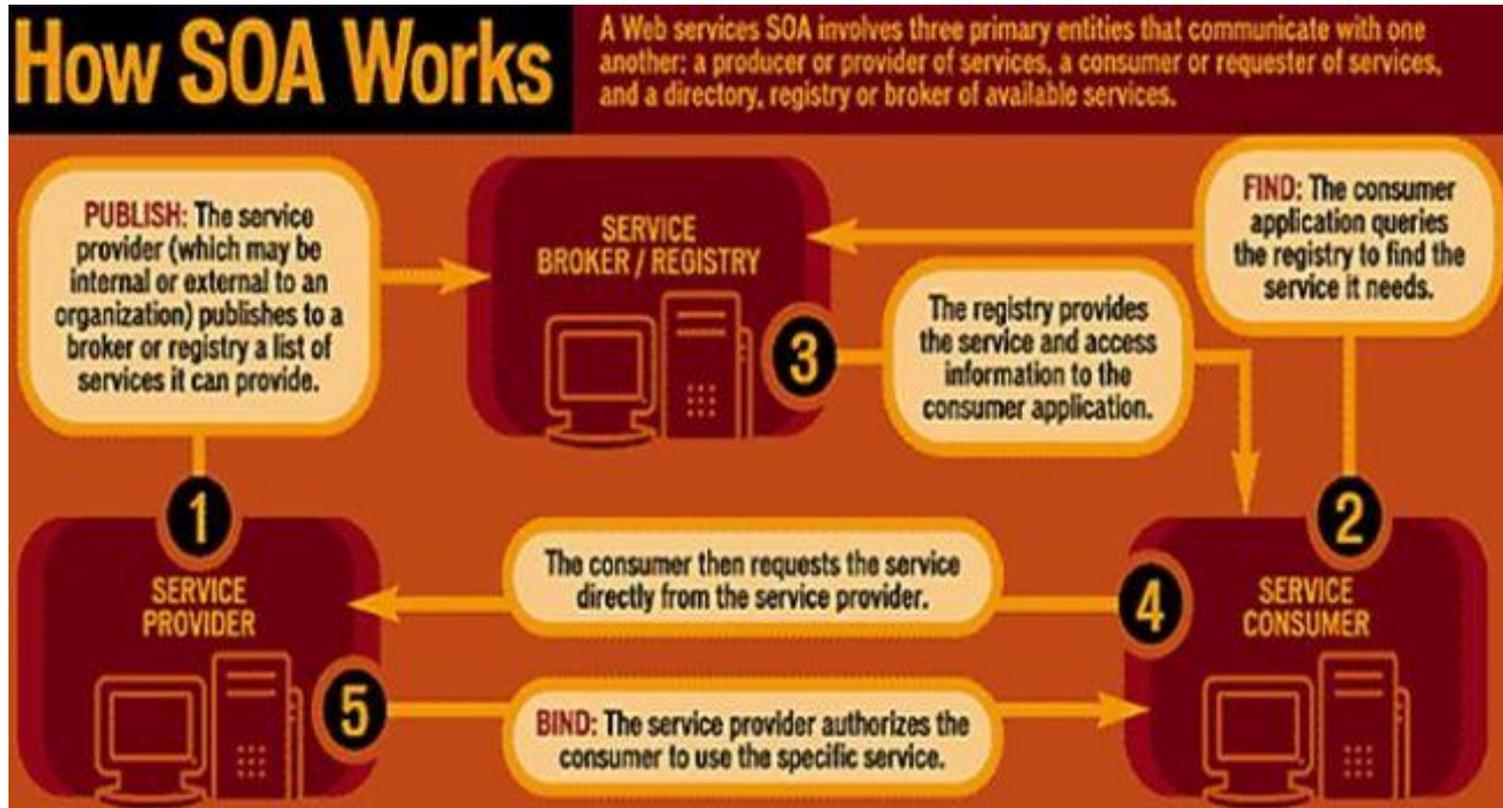
- Code solving a domain-specific problem
- Built by small team using a particular technology stack
- Exposes features to caller via a well-defined API contract
- Degrades gracefully when dependent services fail
- Can be upgraded independently of calling services
- Dividing a big service into smaller services is often referred to as a *microservices architecture*

Service Oriented Architecture (SOA)

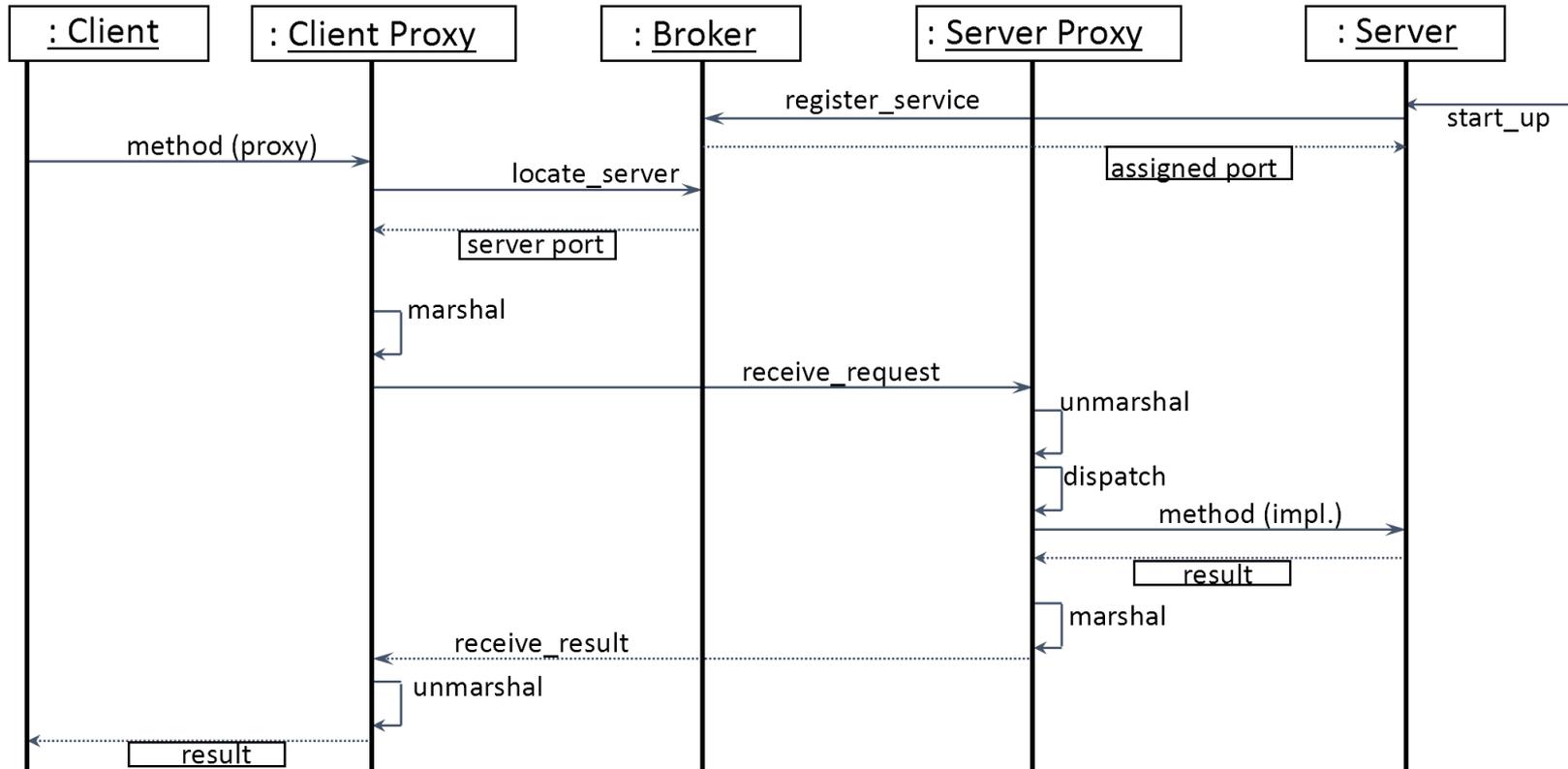
SOA Components and Operations



How SOA Works?



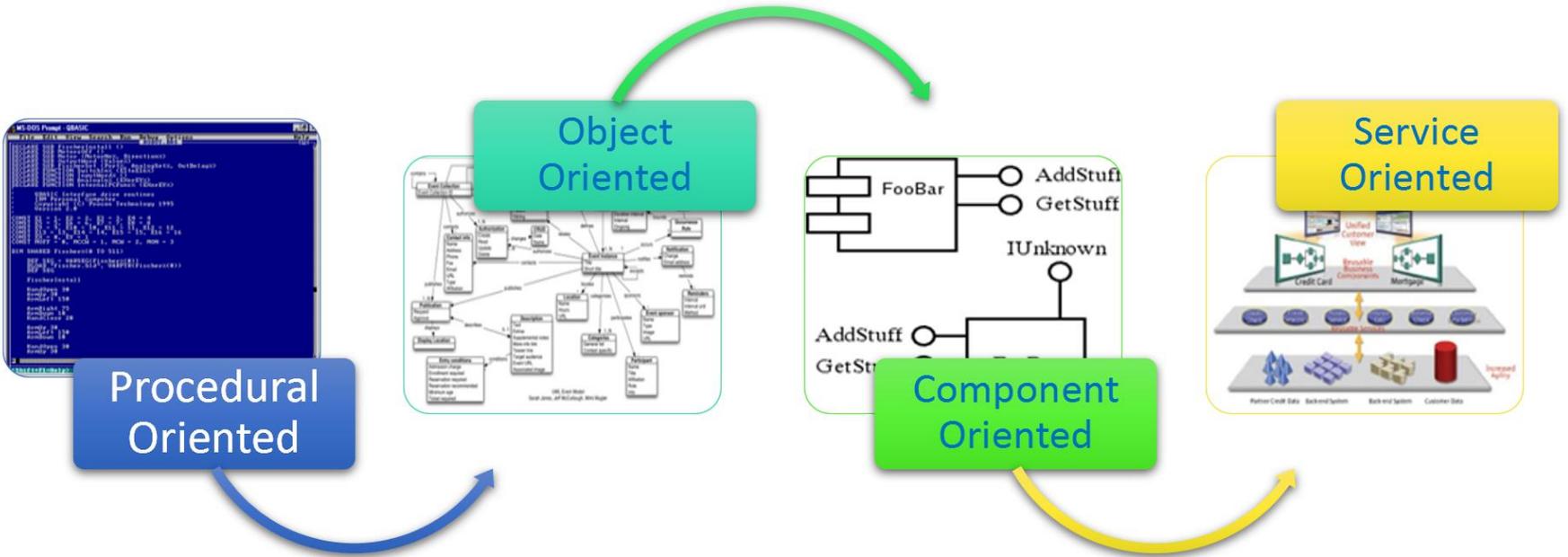
Broker Design Pattern Dynamics



Broker tools provide the generation of necessary client & server proxies from higher level interface definitions



How did it come to SOA?



Microservices



MONOLITHS

Hard to deliver, even harder to test and impossible to maintain

Microservices



Microservices

1990s and earlier

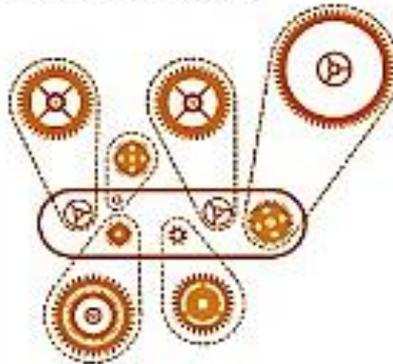
Coupling

Pre-SOA (monolithic)
Tight coupling



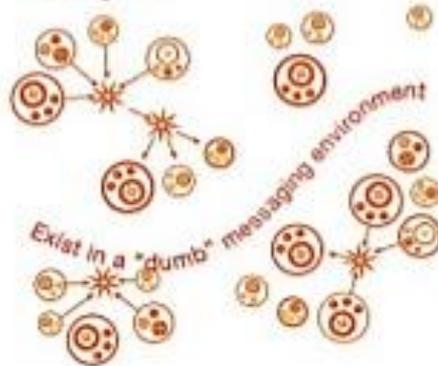
2000s

Traditional SOA
Looser coupling

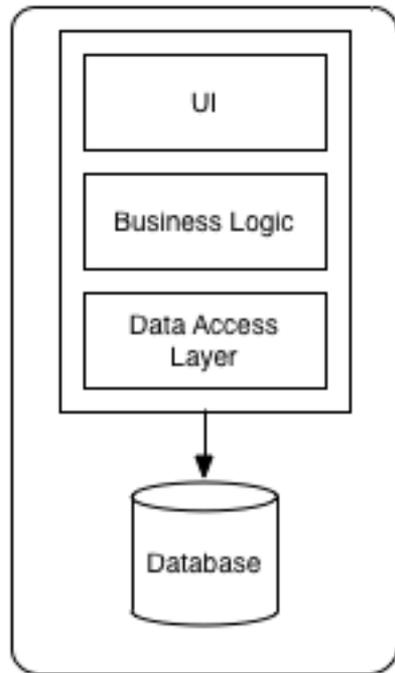


2010s

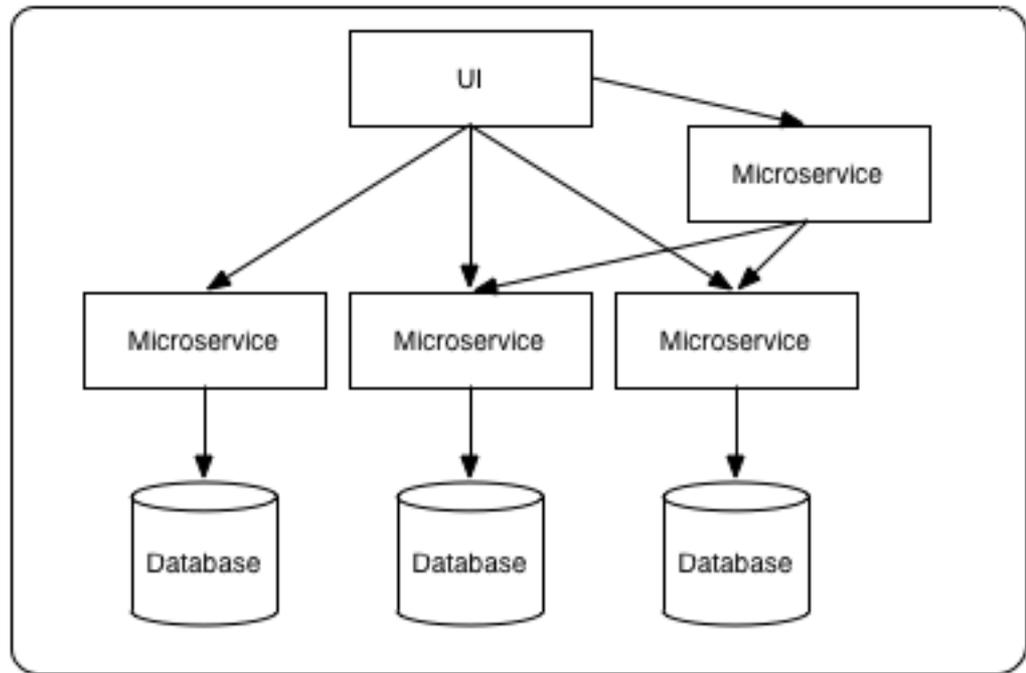
Microservices
Decoupled



Microservices



Monolithic Architecture



Microservices Architecture

Microservices

TIPPING POINT



Organizational Growth

&



Disverse Functionality

&



Bottleneck in
Monolithic stack

Microservices

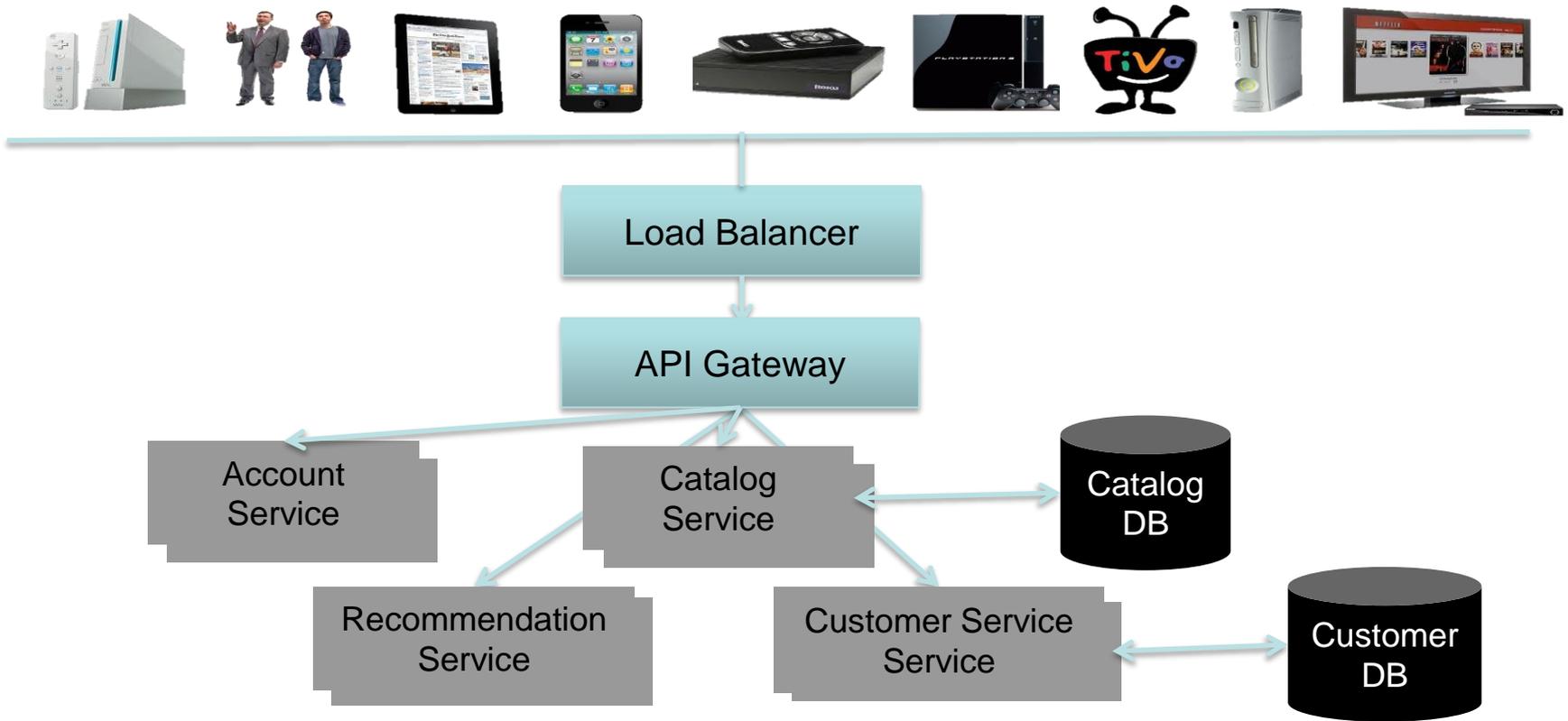
Characteristics

- Many smaller (fine grained), clearly scoped services
 - Single Responsibility Principle
 - Domain Driven Development
 - Bounded Context
 - Independently Managed
- Clear ownership for each service
 - Typically need/adopt the “DevOps” model

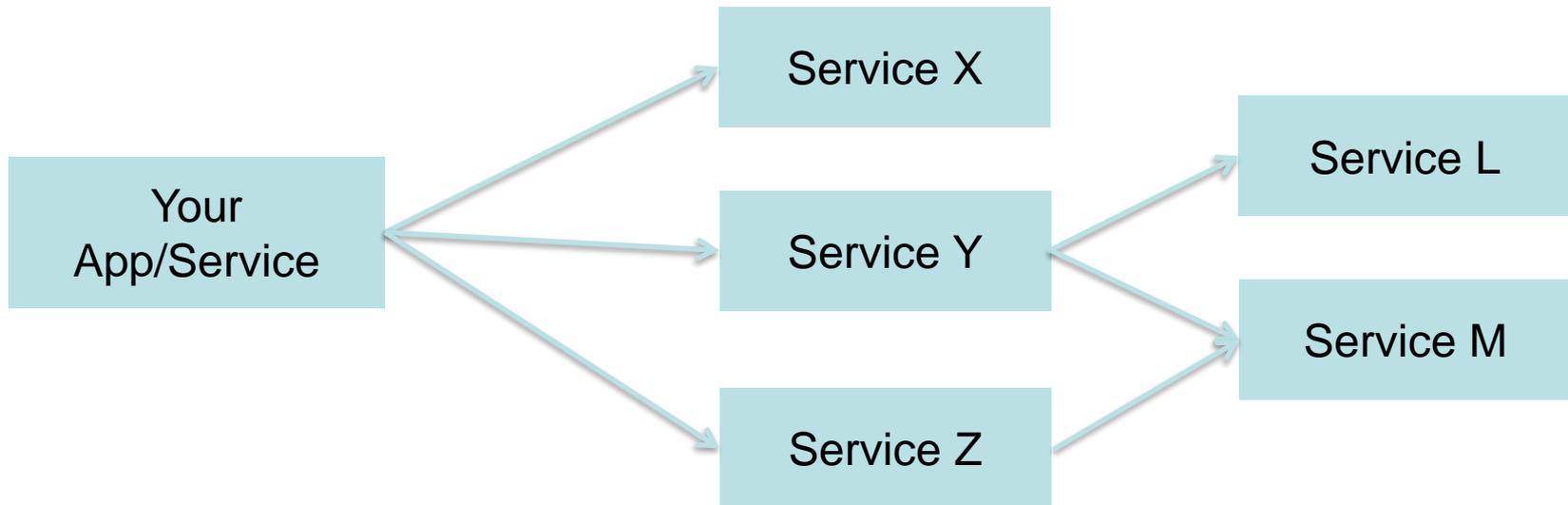
Comparing Monolithic to MicroServices



Microservices Architecture



Concept -> Service Dependency Graph



Why?

- Faster and simpler deployments and rollbacks
 - Independent Speed of Delivery (by different teams)
- Right framework/tool/language for each domain
 - Recommendation component using Python?,
Catalog Service in Java ..
- Greater Resiliency
 - Fault Isolation
- Better Availability
 - If architected right 😊

Challenges



dreamstime.com



Challenges

Pain points

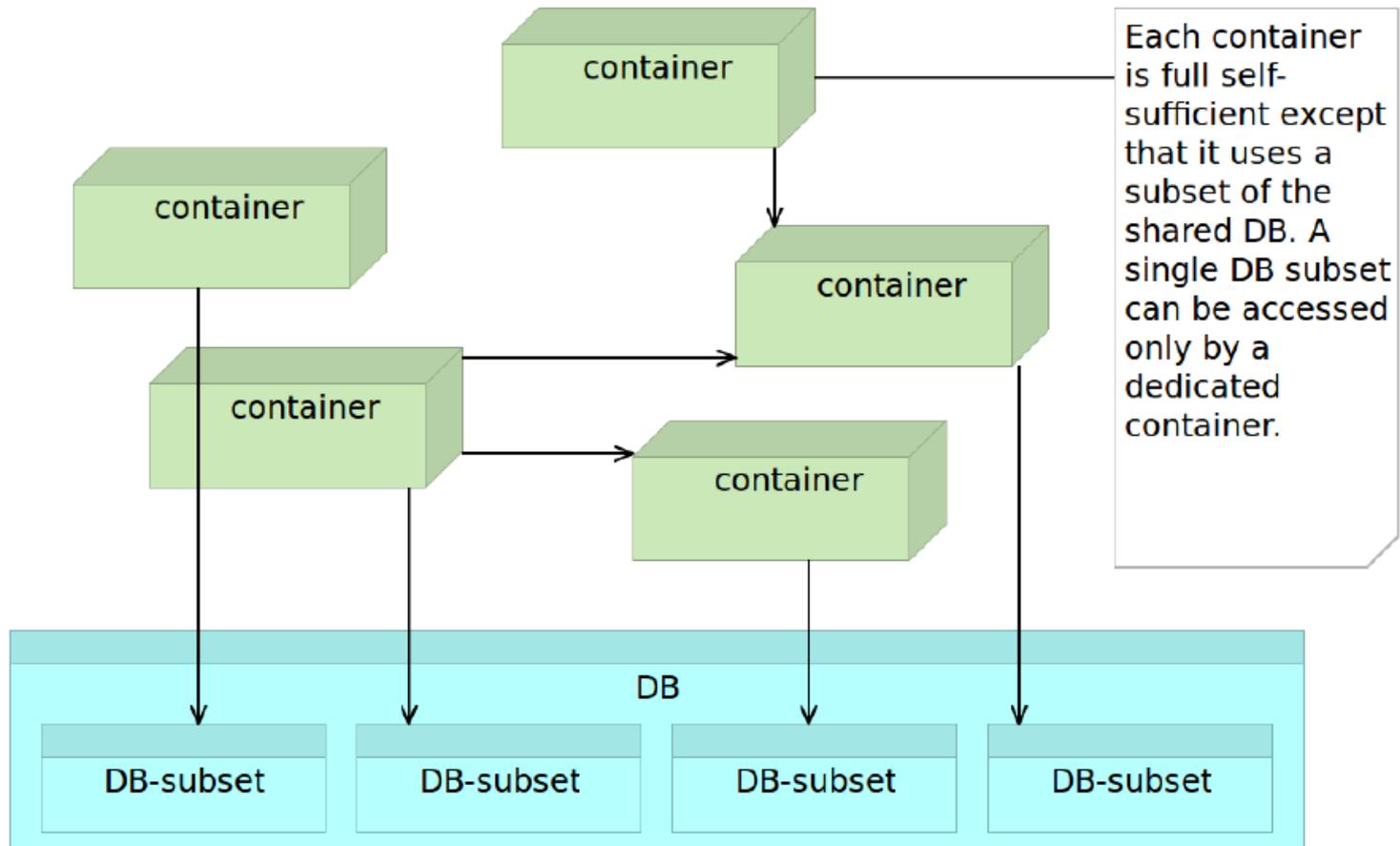
- More services means more network communication
 - Decreases overall performance due to network hops & (de)serialization
 - Requires more failure (timeout) recovery code
- Hard to test in isolation without dependent services
- Hard to debug/monitor across services
- New service versions must support old & new API contracts simultaneously because client services don't upgrade at the same time
- Developers trade short-term pain for long-term gain

Challenges

Pain points

- Distributed Systems are inherently Complex
 - N/W Latency, Fault Tolerance, Retry storms ..
- Operational Overhead
 - TIP: Embrace DevOps Model

Microservices Accessing the Shared Database

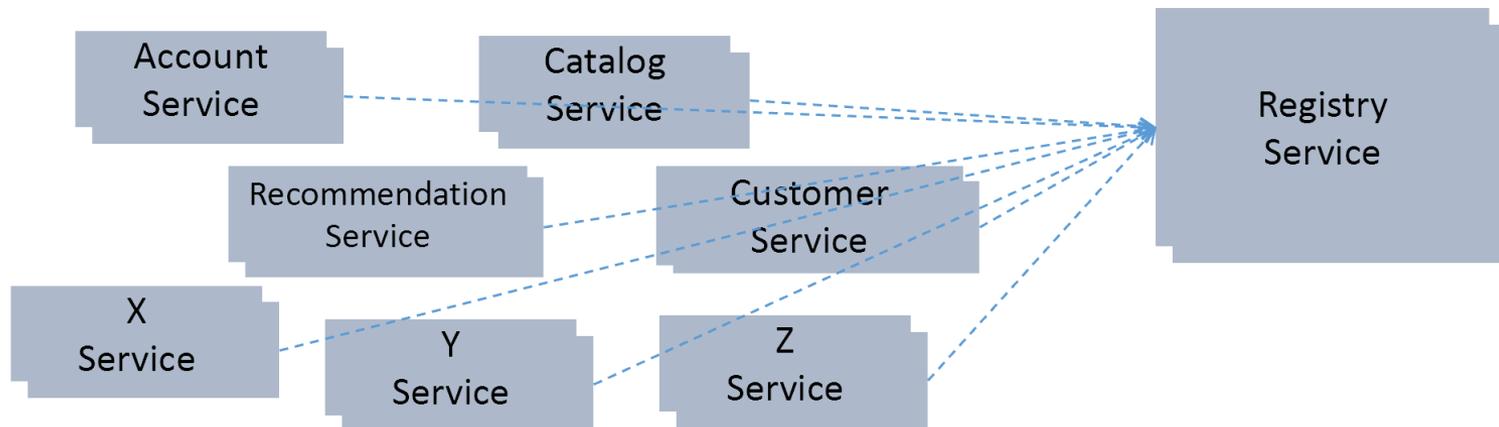


Microservices Characteristics

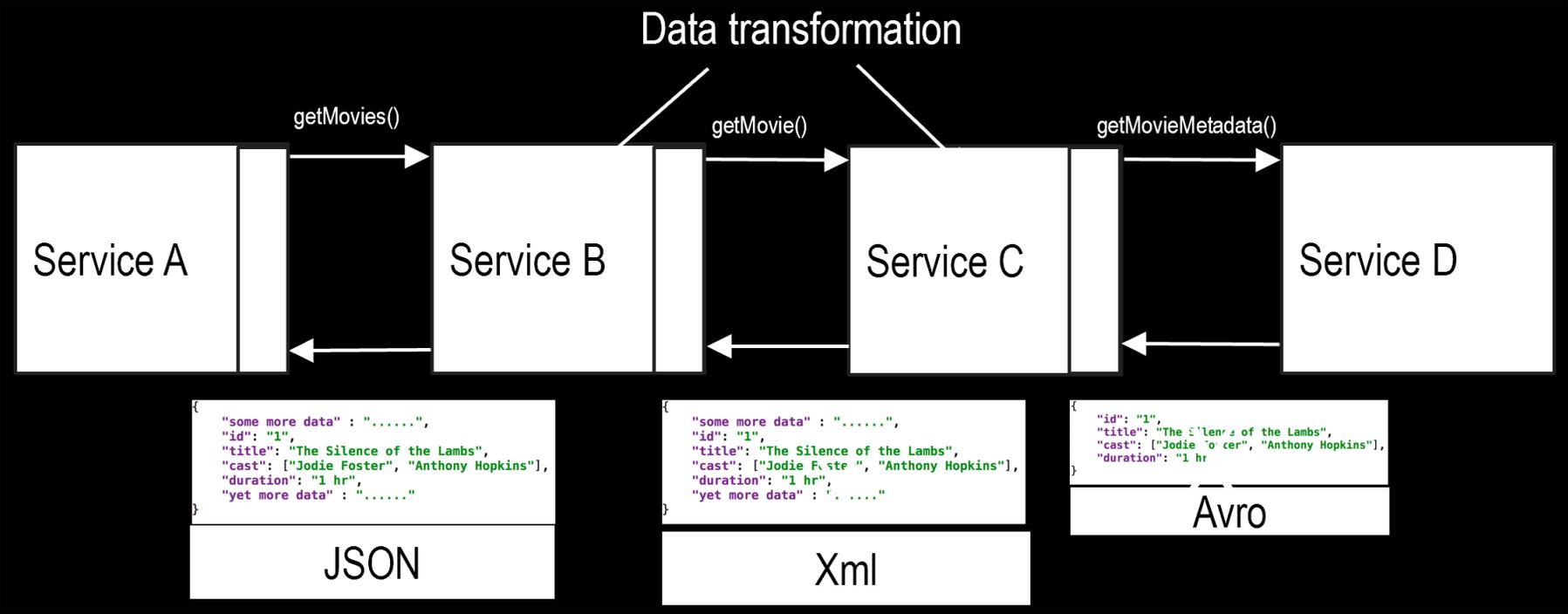
- Many smaller (fine grained), clearly scoped services
 - Single Responsibility Principle
 - Independently Managed
- Clear ownership for each service
 - Typically need/adopt the “DevOps” model

Service Discovery

- 100s of MicroServices
 - Need a Service Metadata Registry (Discovery Service)



Data Serialization Overhead



Best Practice -> Isolation/Access

- TIP: In AWS, use Security Groups to isolate/restrict access to your MicroServices

Edit inbound rules [X]

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ	
HTTP ⓘ	TCP	80	Anywhere ⓘ 0.0.0.0/0	⊗
HTTPS ⓘ	TCP	443	Anywhere ⓘ 0.0.0.0/0	⊗

Best Practice -> Loadbalancers

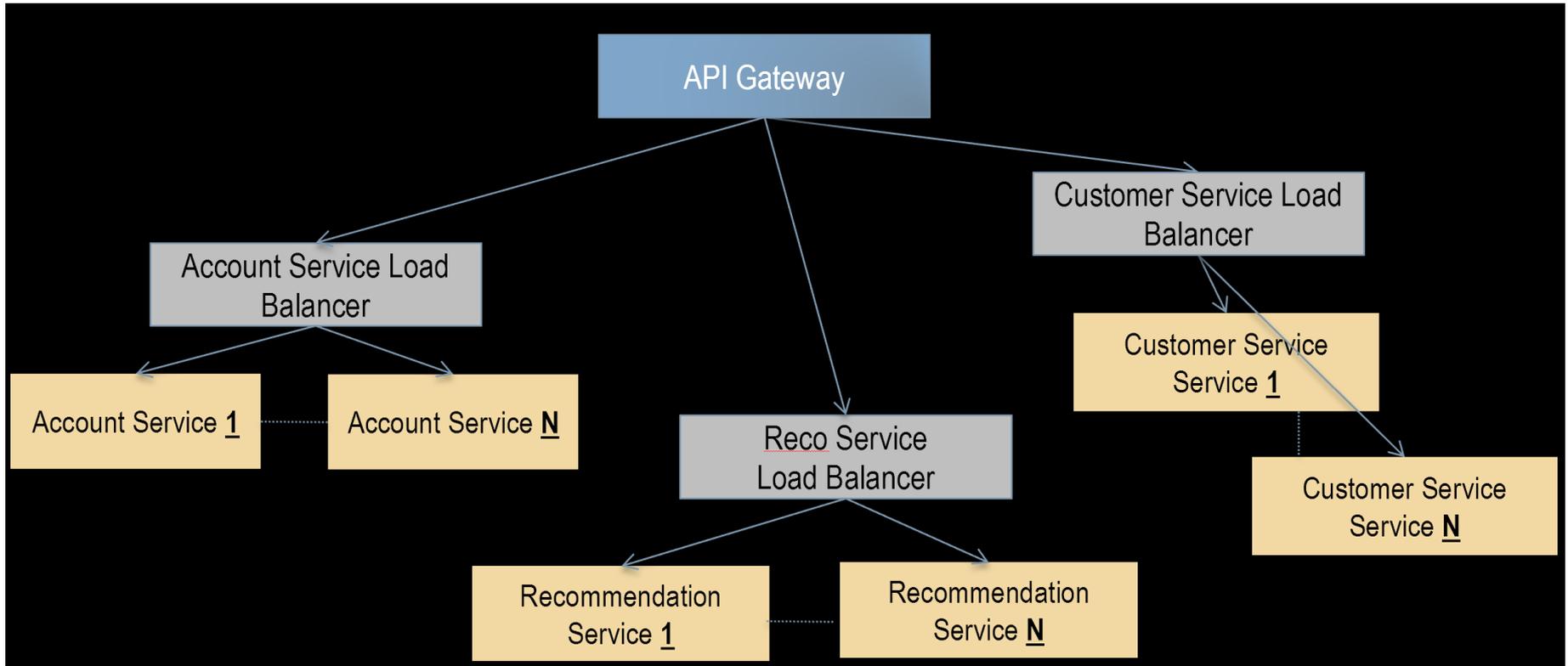
Choice

Central Loadbalancer? (H/W or S/W)

OR

2. Client based S/W Loadbalancer?

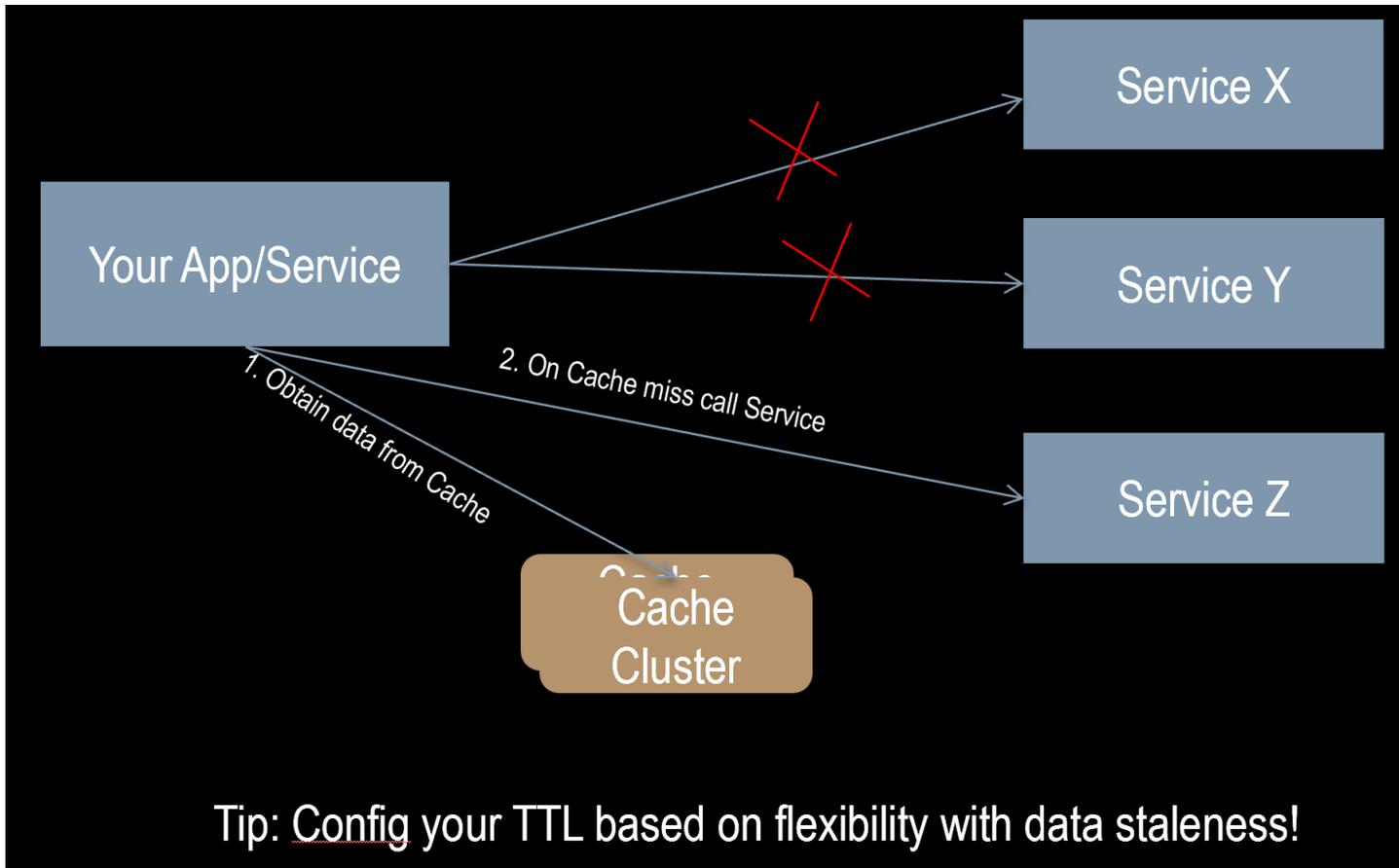
Central (Proxy) Loadbalancer



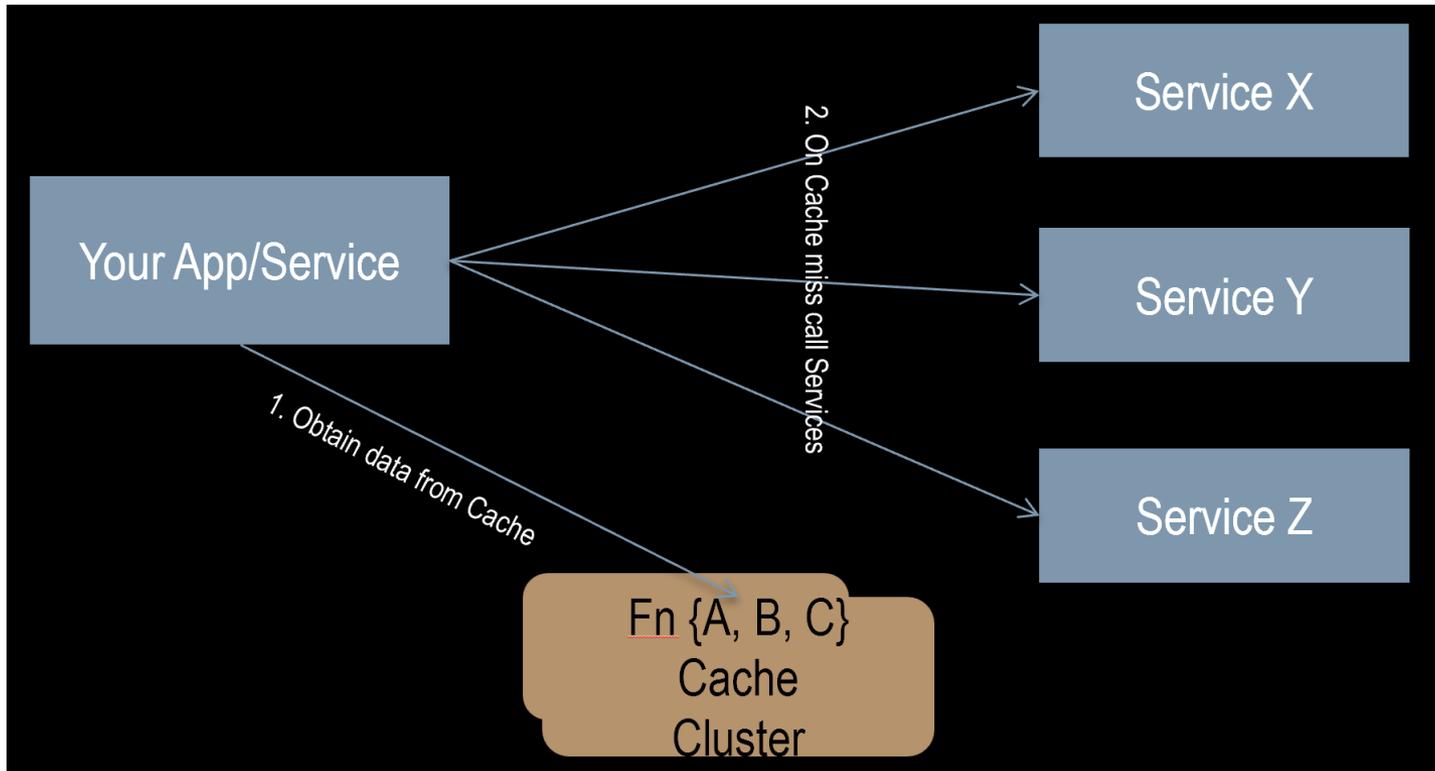
Best Practices

- Dependency Calls
 - Guard your dependency calls
 - Cache your dependency call results
 - Consider Batching your dependency calls
 - Increase throughput via Async patterns

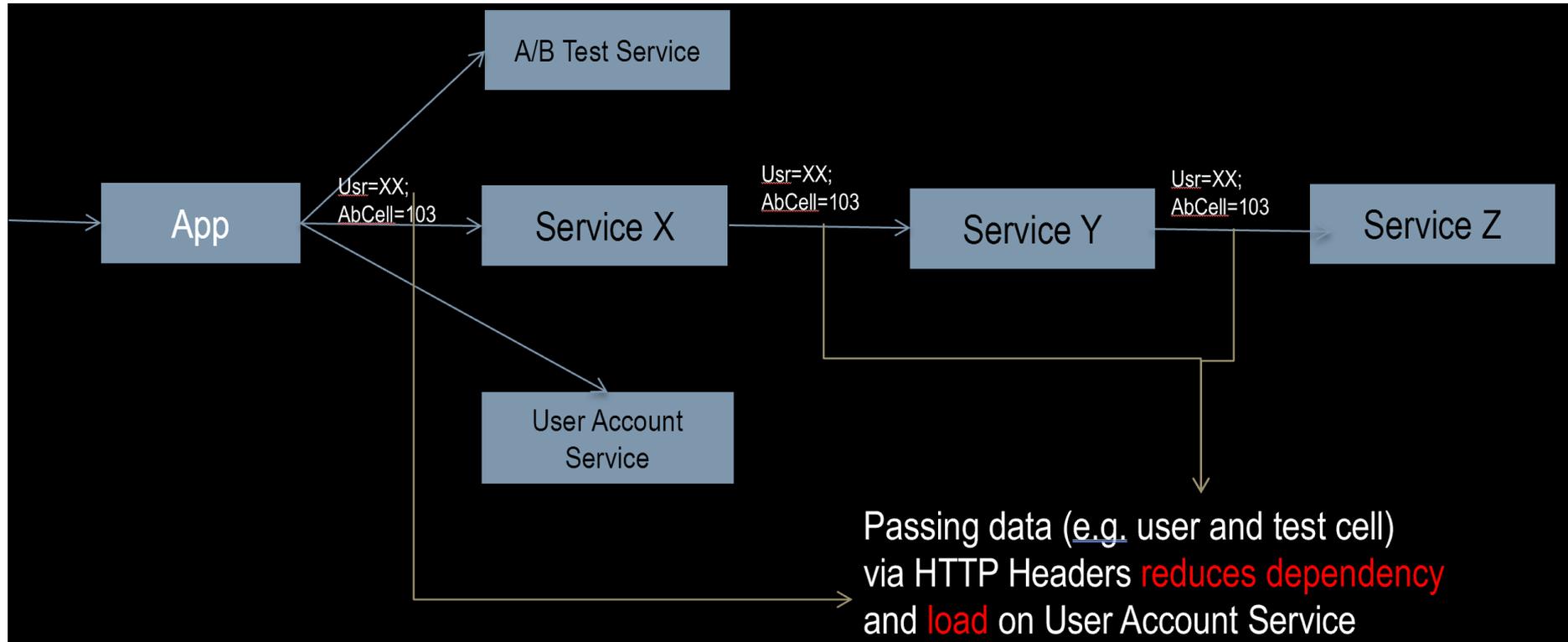
Server Caching



Composite Caching



Tip: Pass data via Headers



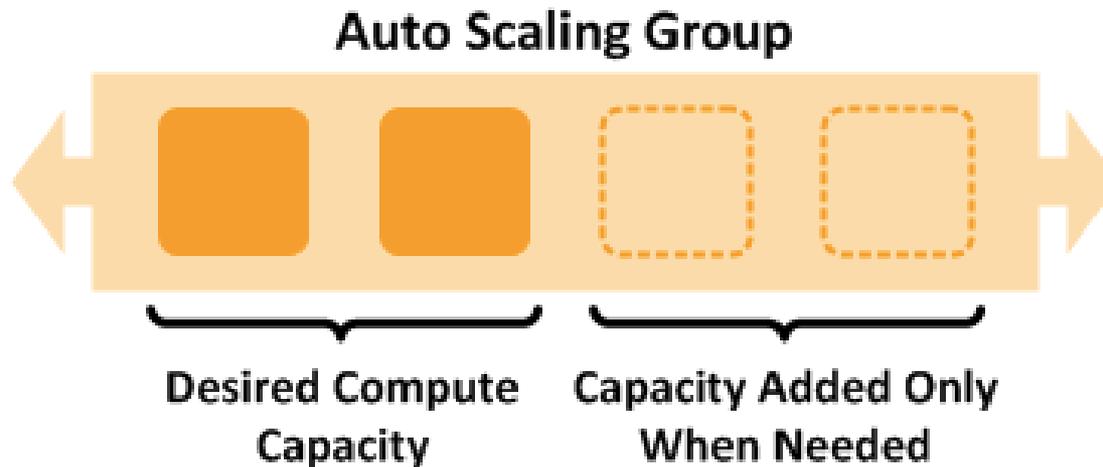
Best Practices

- Test Services for Resiliency
 - Latency/Error tests
 - Dependency Service Unavailability
 - Network Errors

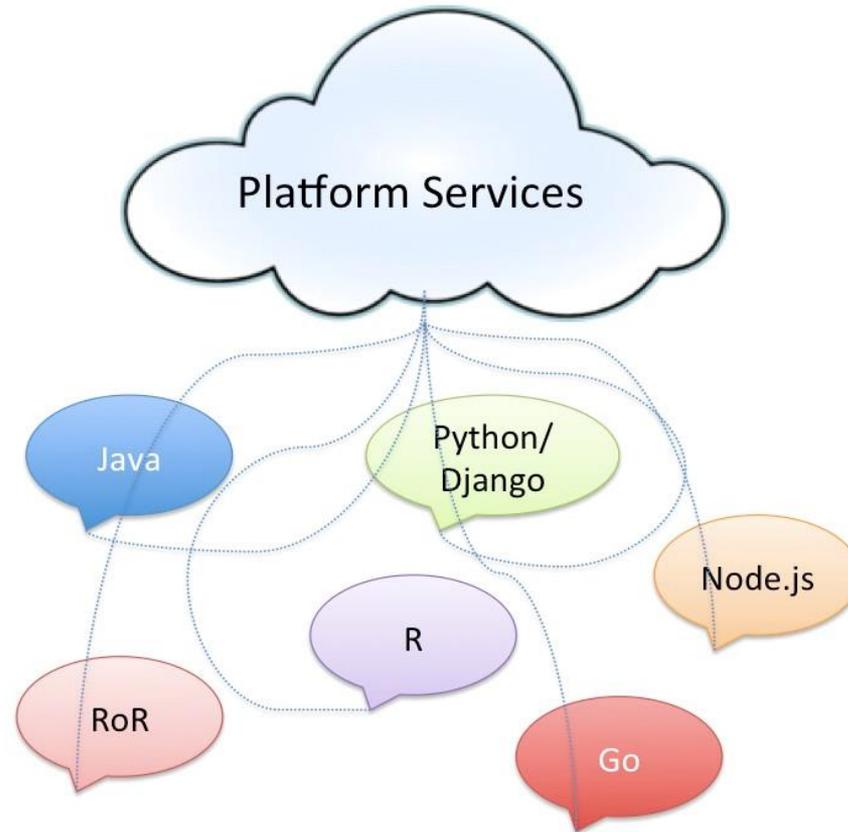
Auto Scaling

Use AWS Auto Scaling Groups to automatically scale your microservices

RPS or CPU/LoadAverage via CloudWatch are typical metrics used to scale



Homogeneity in A Polyglot Ecosystem

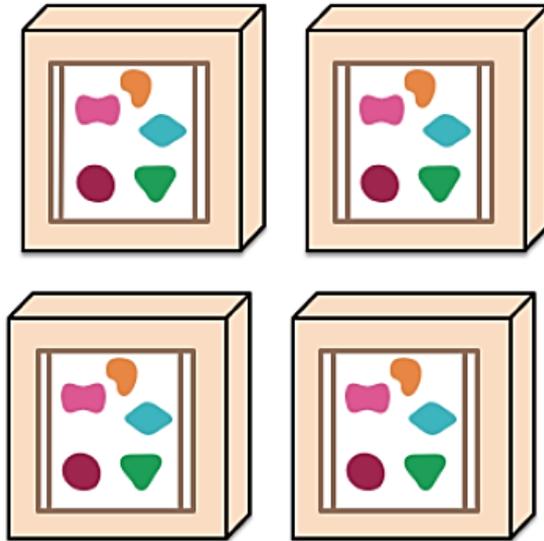


Microservices. Scalability

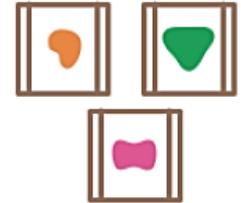
A monolithic application puts all its functionality into a single process...



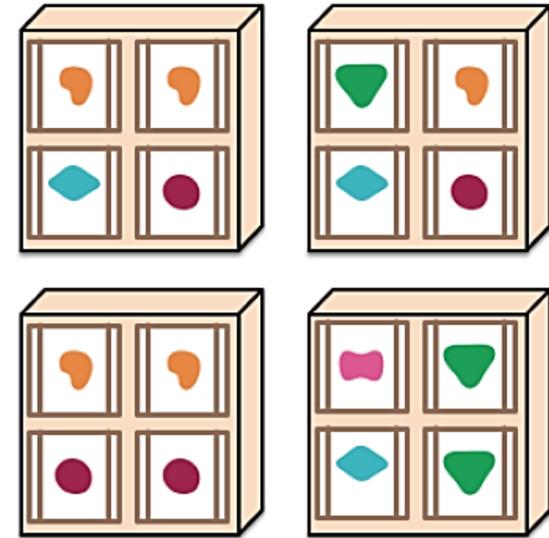
... and scales by replicating the monolith on multiple servers



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



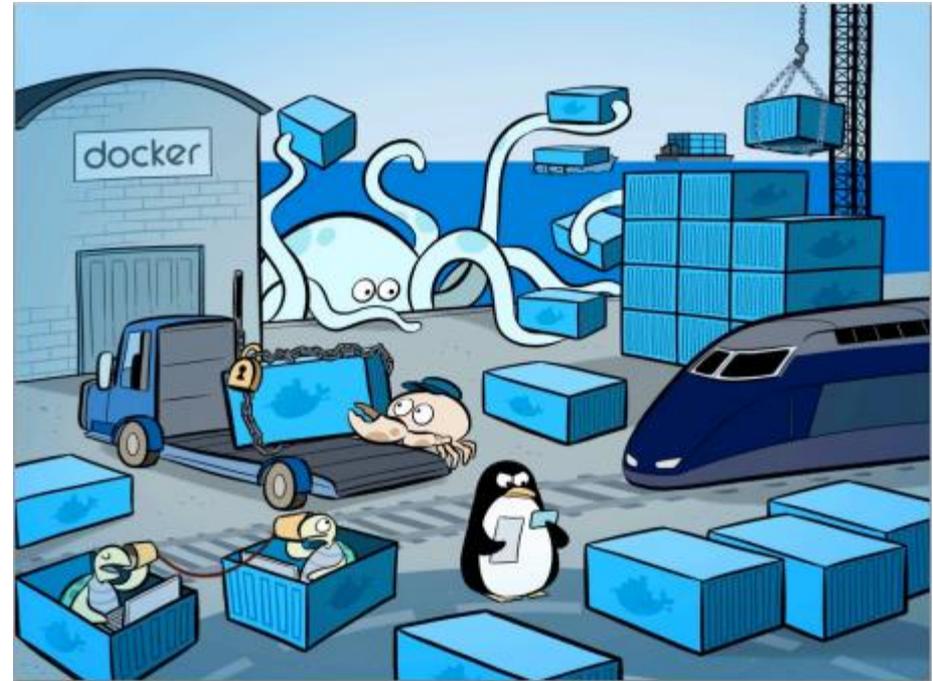
... and scales by distributing these services across servers, replicating as needed.



Docker: Containerization for Software

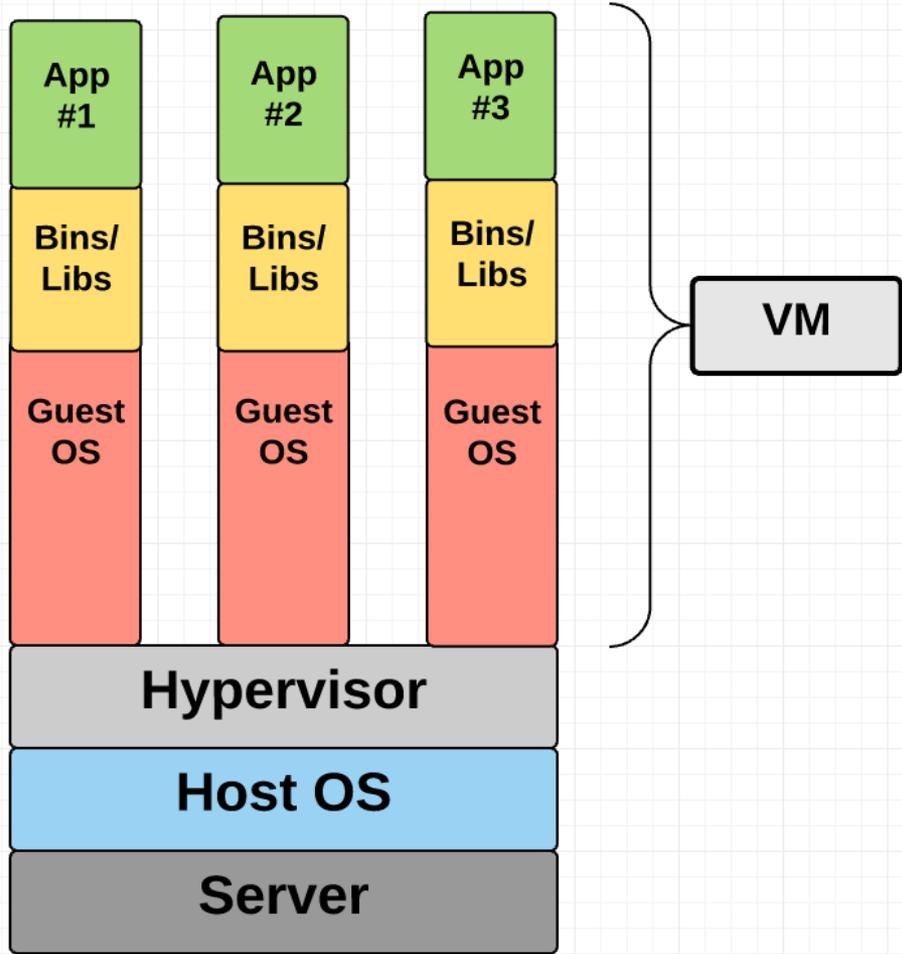


Docker

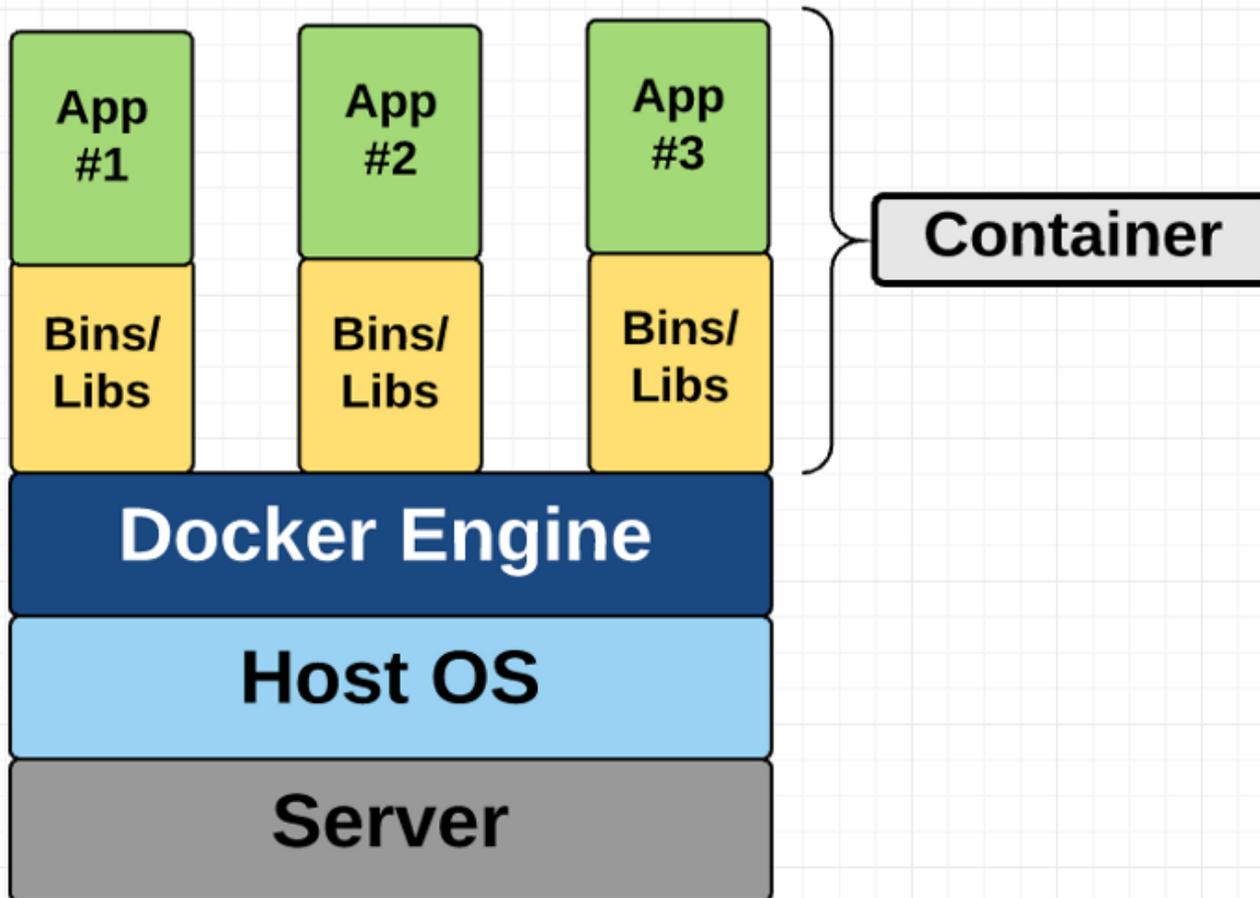


“Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications”

Virtual Machine



Container



So why Docker?

- Containers are far from new;
 - Google has been using their own container technology for years.
 - Others Linux container technologies include
 - Solaris Zones,
 - BSD jails, and
 - LXC, which have been around for many years.
- Docker is an open-source project based on Linux containers. It uses Linux Kernel features.

Docker Benefits

1. Local development environments can be set up that are exact replicas of a live environment/server.
2. It simplifies collaboration by allowing anyone to work on the same project with the same settings, irrespective of the local host environment.
3. Multiple development environments can be run from the same host each one having different configurations, operating systems, and software.
4. Projects can be tested on different servers.
5. It gives you instant application portability. Build, ship, and run any application as a portable container that can run almost anywhere.

Docker Benefits

- **Ease of use.** It allows anyone to package an application on their laptop, which in turn can run unmodified anywhere
 - The mantra is: “build once, run anywhere.”
- **Speed.** Docker containers are very lightweight and fast. Since containers are just sandboxed environments running on the kernel, they take up fewer resources. You can create and run a Docker container in seconds, compared to VMs which might take longer because they have to boot up a full virtual operating system every time.
- **Docker Hub.** Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an “app store for Docker images.” Docker Hub has tens of thousands of public images created by the community that are readily available for use.
- **Modularity and Scalability.** Docker makes it easy to break out your application’s functionality into individual containers. With Docker, it’s become easier to link containers together to create your application, making it easy to scale or update components independently in the future.

VM vs. Docker



Size	A grey elephant icon, representing large size.	A white mouse icon, representing small size.
Startup	A green turtle icon, representing slow startup.	A blue penguin icon sitting on a nest, representing fast startup.
Integration	A red angry face emoji with furrowed brows and a grimace, representing poor integration.	A yellow thumbs up emoji with a smiling face, representing good integration.