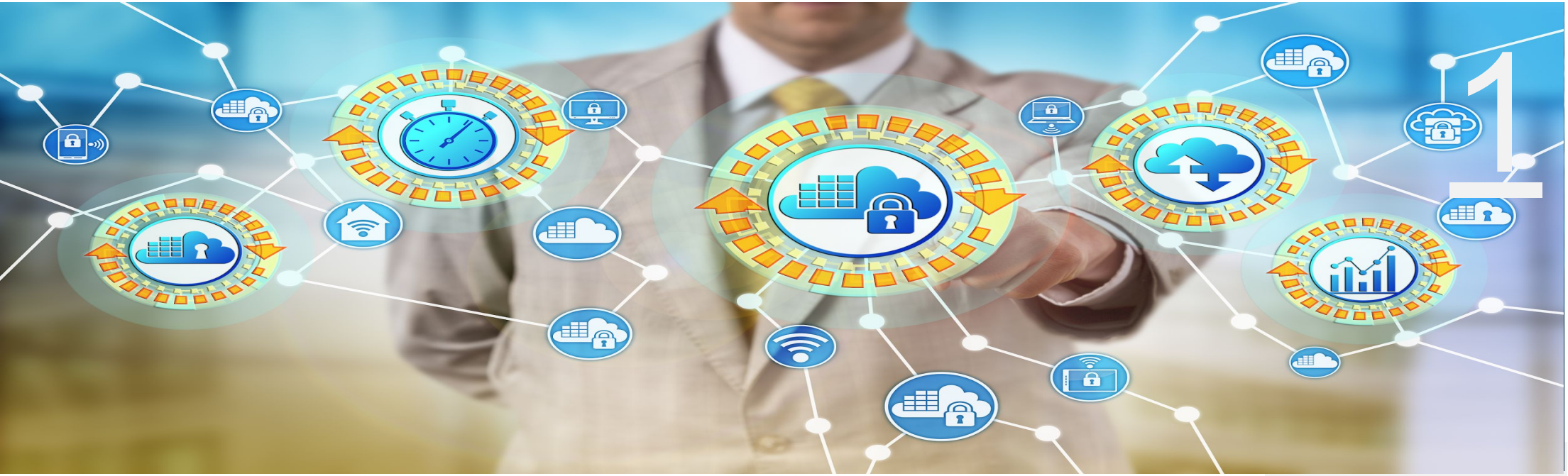
Microservice Application Architecture…



MICROSERVICE APPLICATION ARCHITECTURE

“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.”

MICROSERVICE APPLICATION ARCHITECTURE

By the end of the course you will:

- Have a comprehensive understanding of microservices application architecture
- Have a clear understanding of microservices and how to maximize them in cloud native systems.
- Will gain hands on experience working with microservices in a range of labs
- The course covers the benefits of container based microservice packaging and use of registries as well as dynamic application management using orchestration tools, such as Kubernetes.
- Attendees will examine various use cases and architecture patterns for microservice based applications throughout the course, with focused discussion on microservice communications, transactions and state management.
- Have the skills and information necessary to begin designing and working with microservice based applications.

LOGISTICS

Class Hours:

- Start time is 9am
- End time is 4:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

Lunch:

- Lunch is 11:45am to 1pm
- Yes, 1 hour and 15 minutes
- Extra time for email, phone calls, or simply a walk.

Telecommunication:

- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

Miscellaneous

- Courseware
- Bathroom
- Fire drills

Damian Igbe, PhD

- PhD in Computer Science
- Certified Kubernetes Administrator(CKA)
- AWS Certified Solutions Architect, SysOps & Developer
- Linux Systems Administrator
  > 15 years Systems experience
- Organizer of Kubernetes and CloudNative Dallas Meetup
  https://www.meetup.com/Kubernetes-and-Cloud-Native-North-Dallas/

https://www.cloudtechnologyexperts.com

damianigbe@cloudtechnologyexperts.com

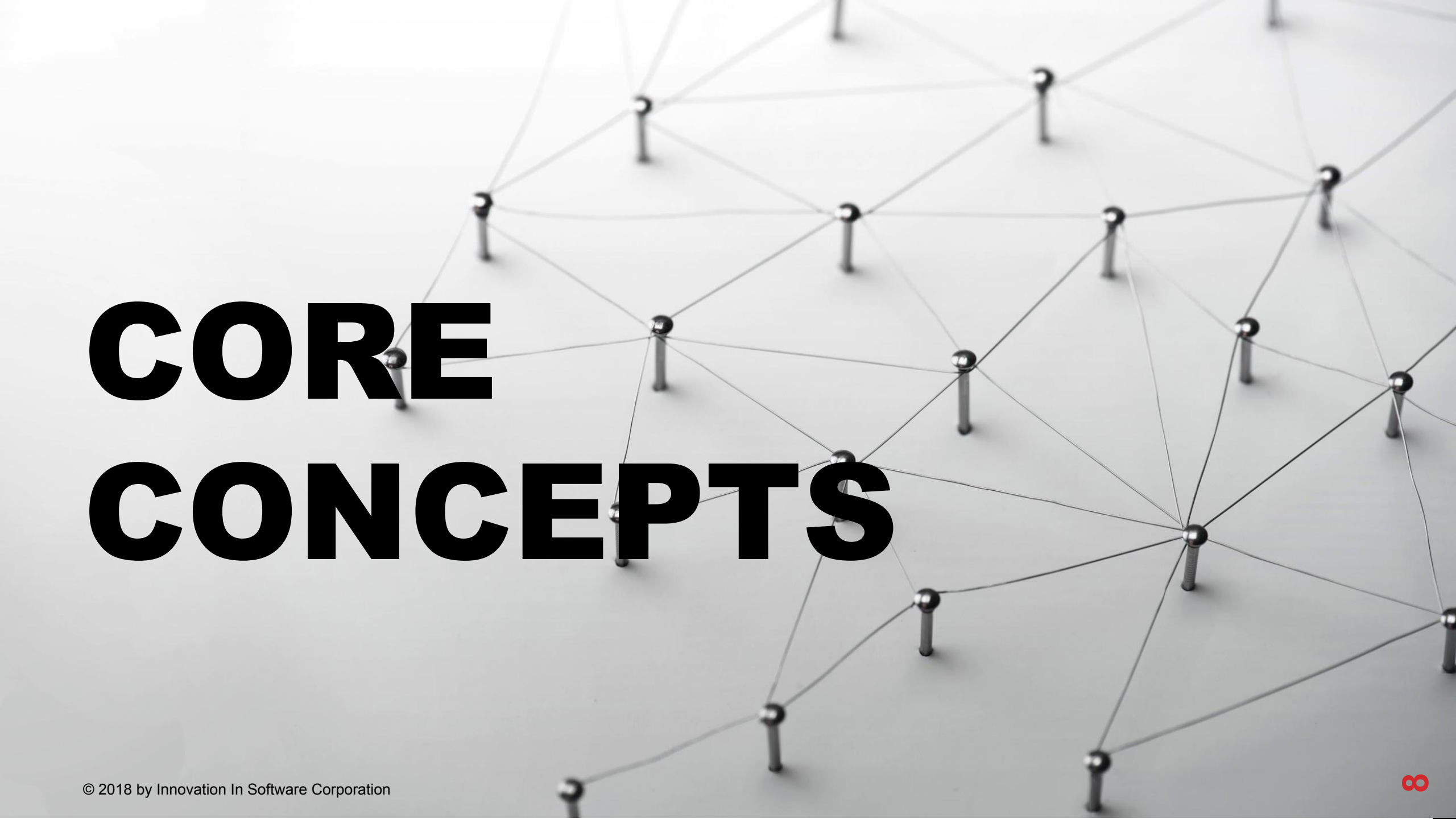
INTRODUCTION

Please tell us about yourself

- First Name
- Role
- Current project
- Languages/Areas of expertise
- Why are you taking this workshop?
- What is best learning style?

QUESTIONS?

Does anyone have any questions before we begin?



CORE CONCEPT$

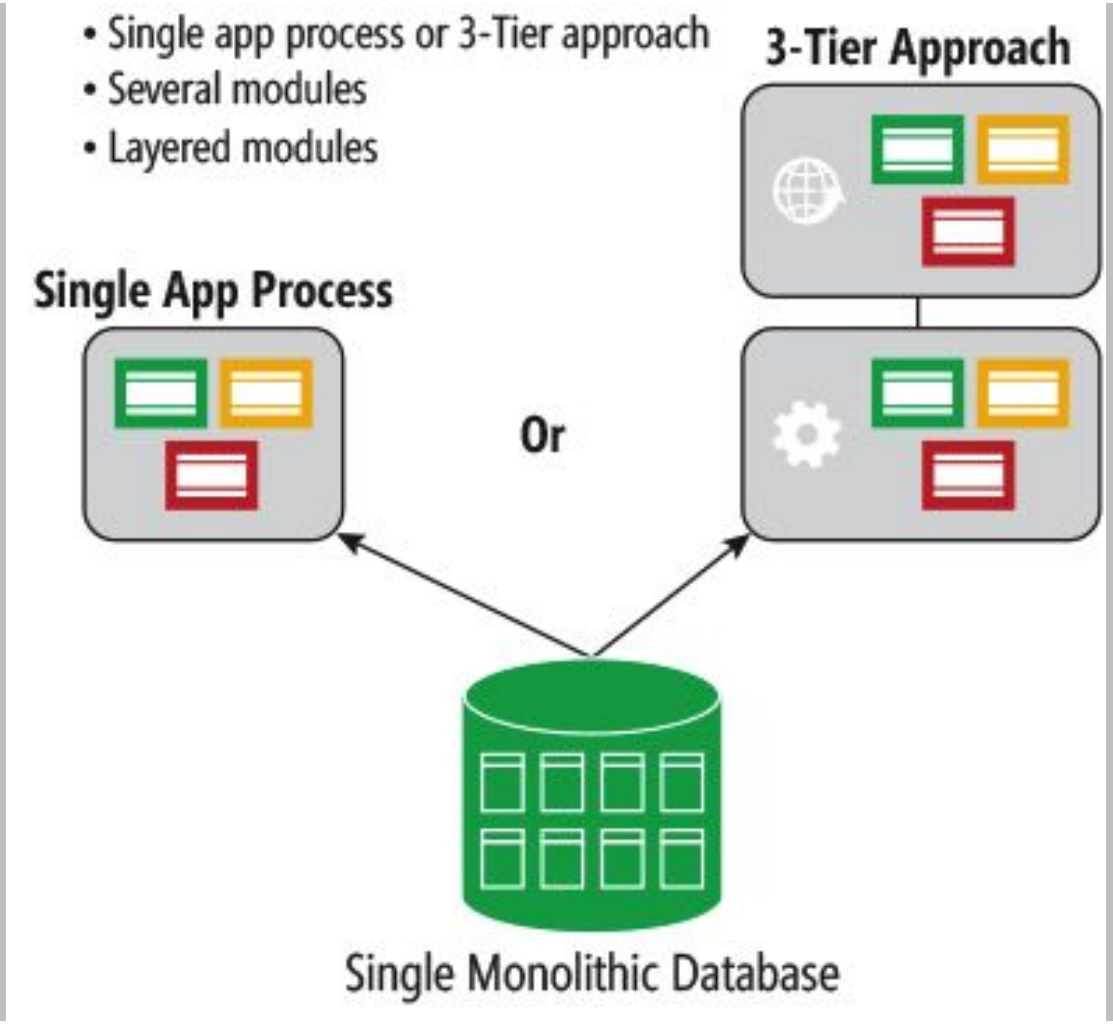
MONOLITHIC OVERVIEW

There is no single definition of a monolithic application

- A single-tiered software application.
- User interface, business logic, and data access are combined into a single program from a single platform.
- Self-contained, and independent from other computing applications.

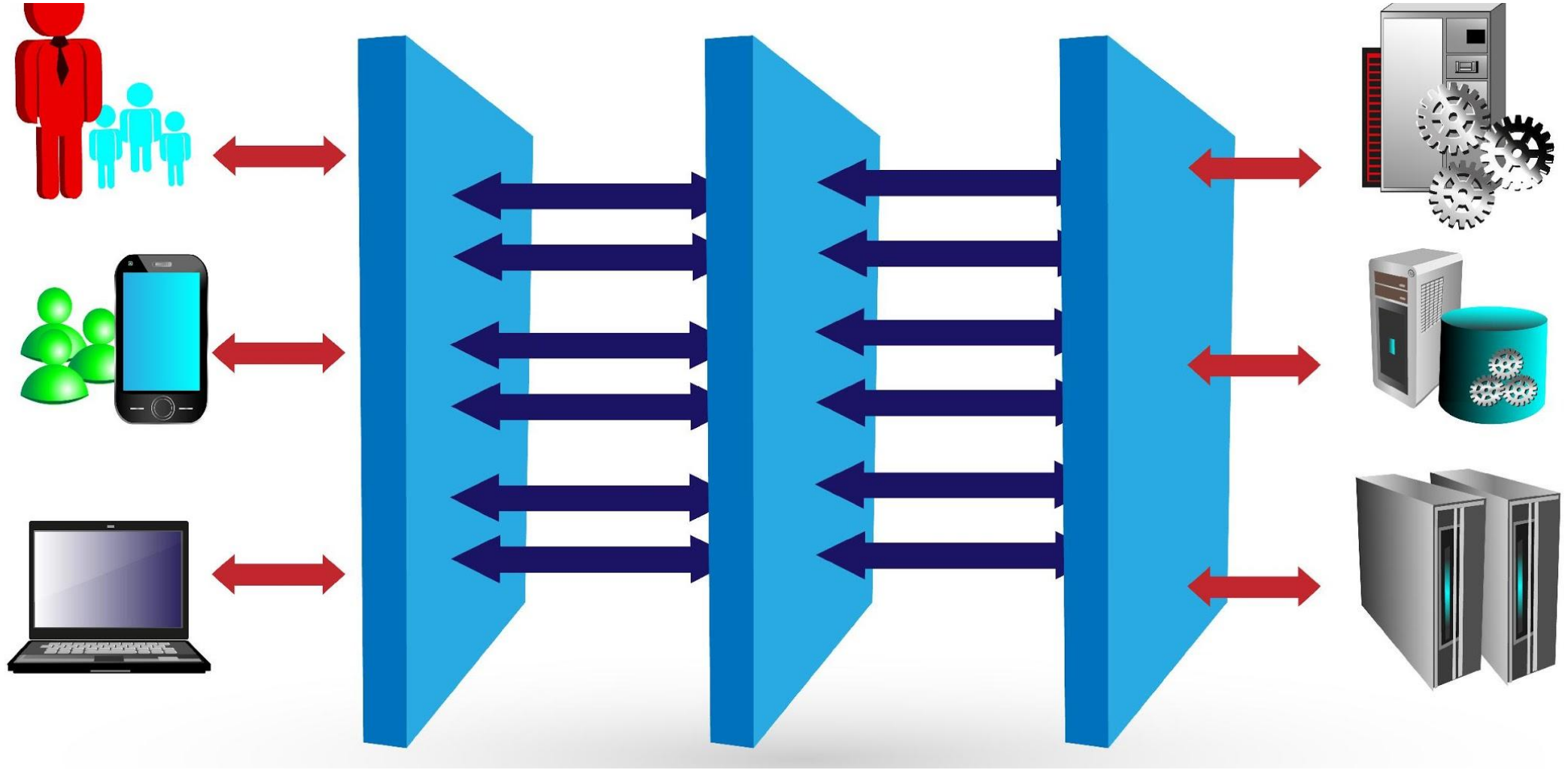
Each of these types of applications have some degree of a monolithic nature

- Enterprise Applications
- Multi-Tier Applications
- Java EE Applications
- SOA Architectures
- Microservices



MONOLITHIC OVERVIEW (CONT'D)

Traditional Application Architectures



MONOLITHIC OVERVIEW (CONT'D)

PROS & CONS OF MONOLITHIC ARCHITECTURES

PROS & CONS OF MONOLITHIC ARCHITECTURES

Pro's

- Easy to develop
- Easy to understand
- Easy to deploy
- Easy to monitor
- Easier to manage transactional integrity
- Single release schedule

Con's

- Difficult to maintain
- Difficult to refactor
- Difficult to scale
- Difficult / impossible to understand
- Requires large team
- Single release schedule

MONOLITHIC BENEFITS

Fewer Cross-cutting Concerns

- Most applications have a large number of cross-cutting concerns (logging, rate limiting, security etc.).
- Can be easy to connect components to cross-cutting concerns.

MONOLITHIC BENEFITS

Less Operational Overhead

- Fewer deployments, and are generally less complex.
- A self contained application is easier to configure, test, and monitor.

MONOLITHIC BENEFITS

Performance

- In-memory access is faster than network communications.
- Easier to manage transactional integrity.

MONOLITHIC BENEFITS

Single Release Schedule

- Entire application is tied to the same release schedule making release management easier.
- Operational effort is manageable.

MONOLITHIC CHALLENGES

MONOLITHIC CHALLENGES

Single Release Schedule

- Entire application is tied to the same release schedule.
- New features cannot be deployed independently.

MONOLITHIC CHALLENGES

Communication Bottlenecks

- In-process communication might require complex threading to overcome bottlenecks.
- Network traffic bottlenecks becomes significant challenge due to limited scaling strategy options (Horizontal vs. Vertical)

MONOLITHIC CHALLENGES

Narrow Technology

- Technology options for monolith are typically dictated for the entire application.
- Application cannot use true polyglot solutions.

MONOLITHIC CHALLENGES

Costly Resource Management

- Difficult to scale as the application is tied to the same scaling strategy (Horizontal vs. Vertical)
- Resources (infrastructure, Network, etc.) will be more costly due to limited scaling strategy options.
- Application services tend to have very different resource requirements (HD, CPU, Memory)

MONOLITHIC CHALLENGES

Lack of Agility

- Small code changes require extensive packaging and testing.
- Slow to deliver features and enhancements.
- Difficult integration with new technology, processes (Mobile, Cloud, Continuous DevOps)
- Monoliths were designed assuming infrequent deployment.

MONOLITH -> SOA -> MICROSERVICES

MICROSERVICE OVERVIEW

“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.”

MICROSERVICE OVERVIEW (CONT'D)

- Microservices are a set of small, independent, composable services.
- Each service can be accessed with RESTful APIs.
- This architectural style de-constructs business processes to their most basic level by creating small, separate processes that then replace large, single applications.

MICROSERVICE OVERVIEW (CONT'D)

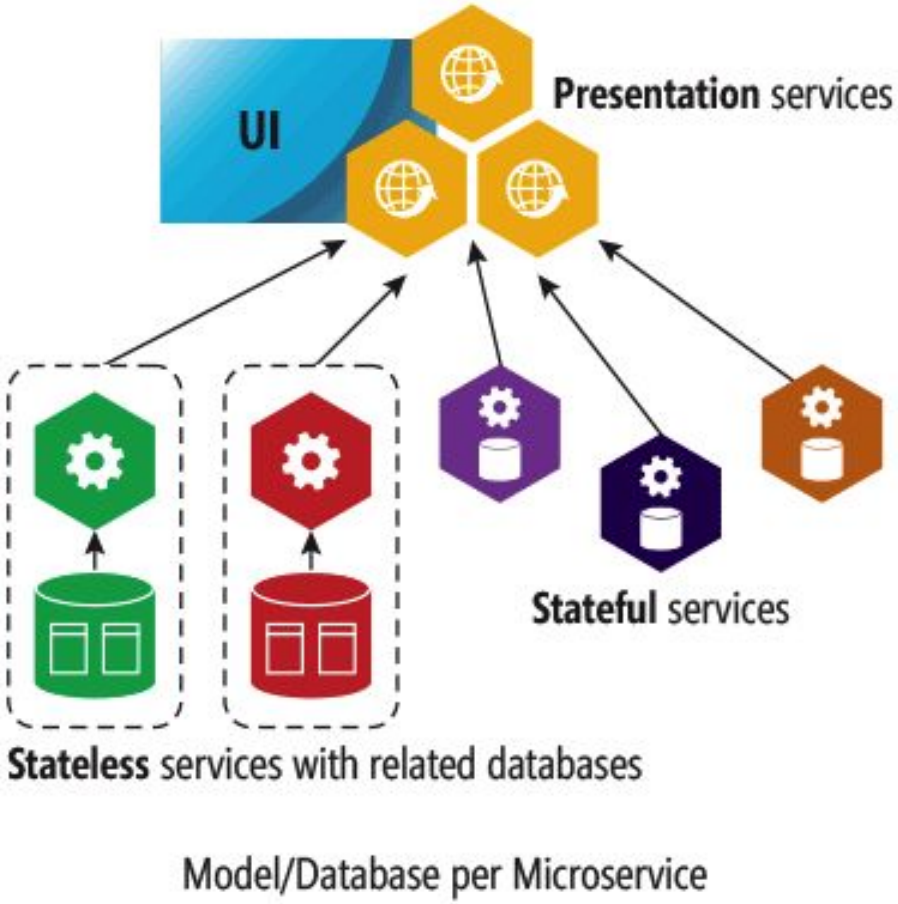
- Breaking a system into smaller parts allows each service to focus on a single business capability.
- Microservices cannot live alone.
- They are a part of the larger system.

MICROSERVICE OVERVIEW (CONT'D)

- They work alongside other services to accomplish the tasks normally handled by one large standalone monolith application.
- They communicate with each other via remote procedure calls (RPCs) which are designed to look and behave like local procedure calls.
- Microservices make applications easier to develop and maintain.

MICROSERVICES APPROACH

Transition from monolith to a microservices approach.



12-FACTOR APPLICATIONS

12-FACTOR APPLICATIONS

In today's contemporary age, software is usually distributed as web apps or software-as-a-service (SaaS).

12-FACTOR APPLICATIONS (CONT'D)

The 12-factor methodologies were established, were:

- **Uses declarative formats to cut down the cost and time for developers for setting up Automation.**
- **Partake a clean contract underlying operating system, which offers maximum portability amongst implementing environments.**

12-FACTOR APPLICATIONS (CONT'D)

The 12-factor methodologies were established, were:

- **It forsakes the need for servers and system admins as they are suitable for deployment on modern cloud platforms.**
- **Gives maximum agility for continuous deployment by minimizing divergence between development and production.**

12-FACTOR APPLICATIONS (CONT'D)

The 12-factor methodologies were established, were:

- **It can scale up without any noteworthy changes to tooling, architecture or development practices.**
- **The 12 factor methodology is language independent.**
- **Can be applied to application written in any programming language using any combination of backing services (database, queue, memory cache, etc.).**

12-FACTOR APPLICATION CHALLENGES

12-FACTOR APPLICATION

Challenges

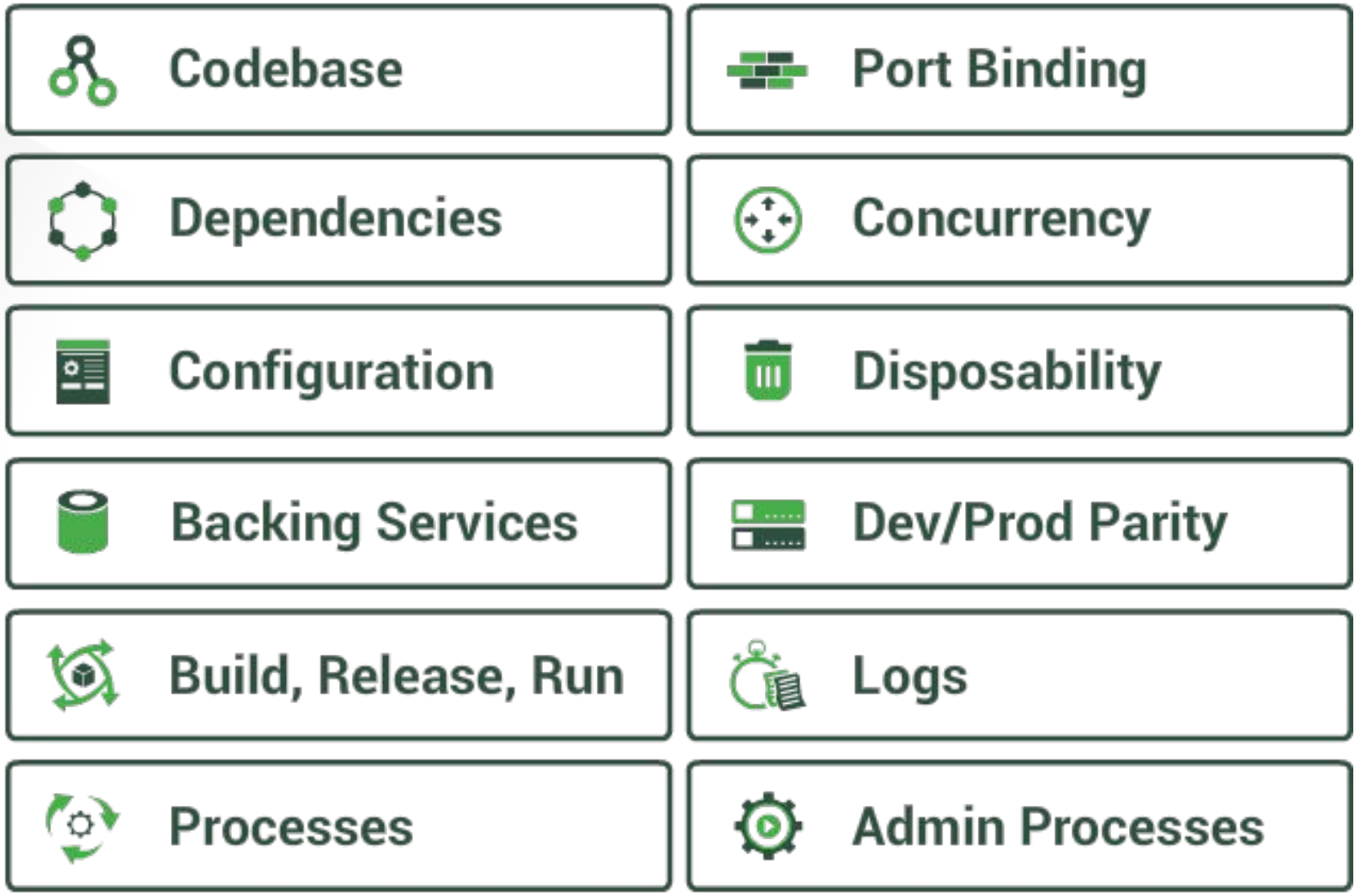
- **How faster can one fail?**
- **How on a push of a button, the team can get the work done?**
- **What process can be used so that code can be made available to production?**

12-FACTOR APPLICATION

Challenges

- **How to automate the above used process with continuous integration (CI), continuous deployment (CD), and even sometimes continuous delivery (CD)?**
- **How to convince the team to change methodology? What are the processes which needs to be defined for the work to be done?**

12-FACTOR APPLICATION BUILDING BLOCKS



12-FACTOR APPLICATION BUILDING BLOCKS

I. Codebase

One codebase needs to be defined on top of which one nee sot built, track, revise controls and various deployments. Automation should be implemented in deployment so that everything can run in different environments without work.

II. Dependencies

Isolate dependencies, the second factor is all about clearly declaring and isolating the dependencies as an app is a standalone structure, which needs to install dependencies. For this, whatever is required is declared in the code itself.

III. Config

In this one store configuration files in the environment. This factor focuses and how to store “The Data”. The URI for a database will be different in Development phase, QA phase and Production.

IV. Backing Services

All the backing services should be treated as attached recourses as the one may want diverse databases depending on which team one might be in. With this methodology, each developer can have their own config file as occasionally a dev will want a lot of log files, which a QA will not.

12-FACTOR APPLICATION BUILDING BLOCKS

V. Build, release, run

Separate build, release and run stages making sure all the process has the right libraries.

VI. Processes

All process should be stateless, ensure all the stuff is stored in a backing store and there is nothing is pass along when the application scales up and out. Run the app in one or more stateless processes.

VII. Port binding

This means execute all the port services via port binding, it allows the internal customers to access the endpoints without traversing security. This also helps in checking the authenticity of the working stack from different ports.

VIII. Concurrency

Break the app into smaller chunks allowing to scale out as and when needed to handle fluctuation in load.

12-FACTOR APPLICATION BUILDING BLOCKS

IX. Disposability

Highest robustness with fast startup, elegant shutdown and gracefully handling a crash.

X. DEV/PROD parity

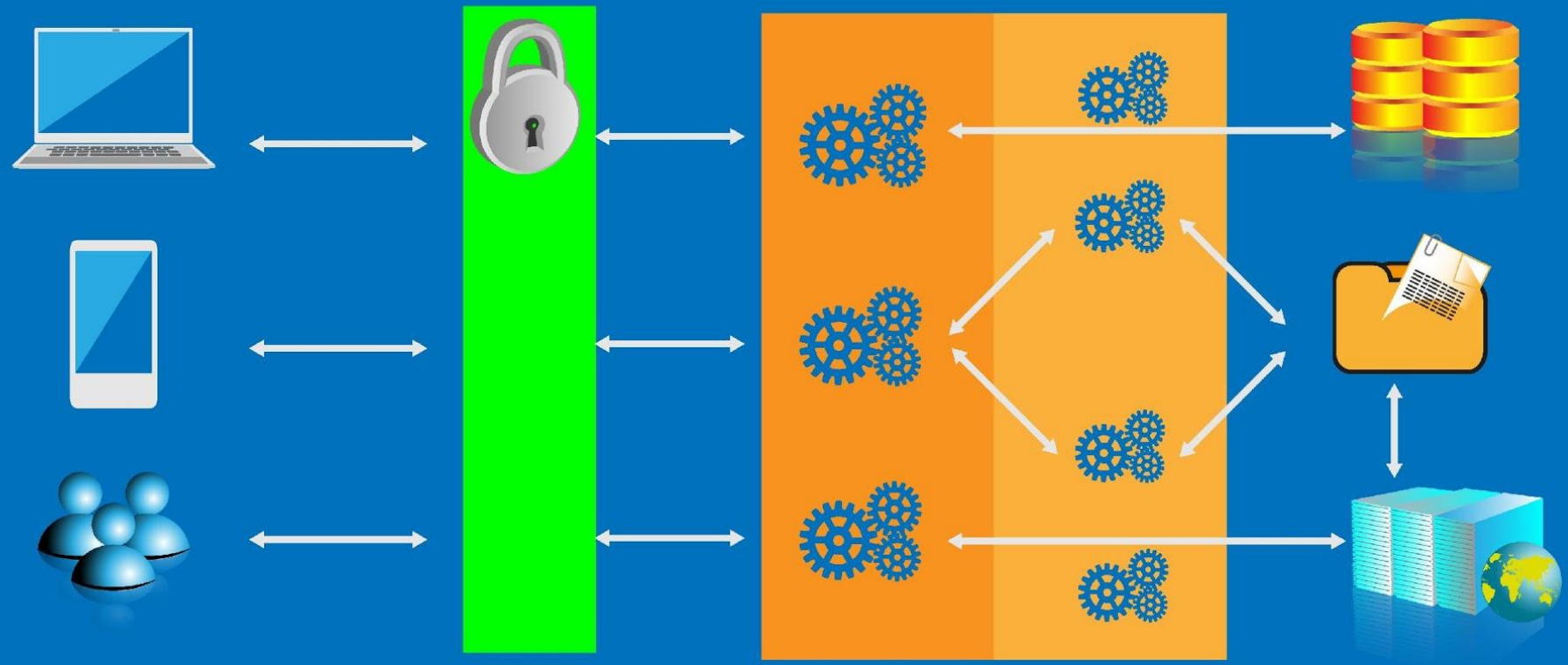
Continuous deployment requires continuous integration built on identical environments limiting divergence and blunders. keeping dev, staging and production as similar as possible helps anyone who comes in to understand the app and release it. This permits a lot more scalability and is also good development.

XI. Logs

Logs can be treated as event streams as the app only concerns itself about creating a sort of even stream. Which log needs to be published is then decided depending upon the configuration.

XII. Admin Processes

One of the off-processes can be running and managing admin tasks.



CHARACTERISTICS OF MICROSERVICES

CHARACTERISTICS OF MICROSERVICES

- Decentralized governance and data management.
- Automated and designed for failure.
- Infrastructure can be built, deployed and verified with CI/CD automation.

BUSINESS MOTIVATIONS

Business Agility

- Speed of change is faster with a more modular architecture.
- React quicker to feature and enhancement requirements.
- Easy integration with new technology, processes (Mobile, Cloud, Continuous DevOps)

BUSINESS MOTIVATIONS

Accelerate Digital Transformation

- Provide more flexibility through capability reuse.
- Helps reduce technical debt.

BUSINESS MOTIVATIONS

Facilitate onboarding of new personnel

- Reduce learning curve by providing a more understandable architecture.
- Improves developer velocity.

BUSINESS MOTIVATIONS

Enable Polyglot Rewrite

(e.g. Java, Node, .NET, etc.)

- Utilize the best technology choice for each service offering.
- Incremental approaches centered on smaller components radically reduces risk.

MICROSERVICE ARCHITECTURES

MICROSERVICE ARCHITECTURES

Pro's

- Easy to develop
- Easy to understand
- Easy to deploy
- Easy to monitor each service
- Flexible Release Schedule
- Use standard APIs (JSON, XML)

Con's

- Requires more deployments
- Requires translation (JSON, XML)
- Requires more monitoring
- Operations configuration more complex
- System can be very complex

MICROSERVICE BENEFITS (DESIGN)

Decoupled

- Services tend to be loosely coupled, making it easy to isolate services.
- Removes business and data logic from applications.
- Easy to integrate 3rd party service.
- Services are re-composable and re-configurable to serve multiple purposes
    - (e.g. Web clients and public API)

MICROSERVICE BENEFITS (DESIGN)

Decoupled

- Services tend to be loosely coupled, making it easy to isolate services.
- Removes business and data logic from applications.
- Easy to integrate 3rd party service.
- Services are recompose able and reconfigurable to serve multiple purposes (e.g. Web clients and public API)

Easier To Understand

- Services are small and easy to understand.
- Service architectures are typically better organized, since each service has a very specific job.

Increased Agility

- Easy to integration with new technology, processes (Mobile, Cloud, Continuous DevOps)
- Services should be designed assuming frequent deployments.
- Services can be designed in parallel by establishing well defined boundaries between services.

MICROSERVICE BENEFITS (DEVELOPMENT)

Reduced Risk While Refactoring

- Small services have small codebases and are easy to refactor.
- Services are easy to completely rewrite.

Polyglot Technology

- Technology options for each service can be dictated by each service independently.
- Utilize the best technology choice for each service offering.
- Utilize the latest technologies (framework, language , practice, etc.).

Increased Agility

- Code is easy to understand, modify and maintain.
- Small code changes require simpler packaging and testing.
- Services can be developed in parallel based on well defined API boundaries.

MICROSERVICE CHALLENGES

MICROSERVICE CHALLENGES

Cross-cutting Concerns Across Each Service

- Unanticipated Cross-cutting concerns might be missed at design time.
- Can lead to additional overhead for service redesign.

System Harder To Understand

- Microservice architectures are also much harder to understand as a whole.
- Issues are difficult to trace and address in a highly distributed deployment.

MICROSERVICE CHALLENGES (CONT'D)

More Operational Overhead

- Exponentially more deployments, and system tends to be more complex.
- Potential duplication of effort.
- System as a whole is more difficult to configure, test, and monitor.

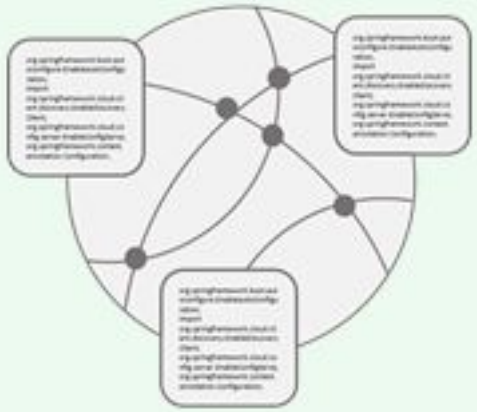
Performance

- Remote calls are more expensive than in-process calls.
- Far more difficult to manage transactional integrity.

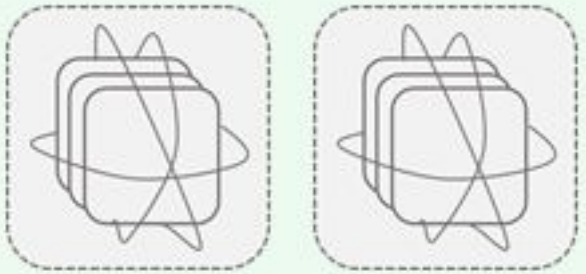
THE BUILDING BLOCKS OF MICROSERVICES

DevOps

Microservices & API:s

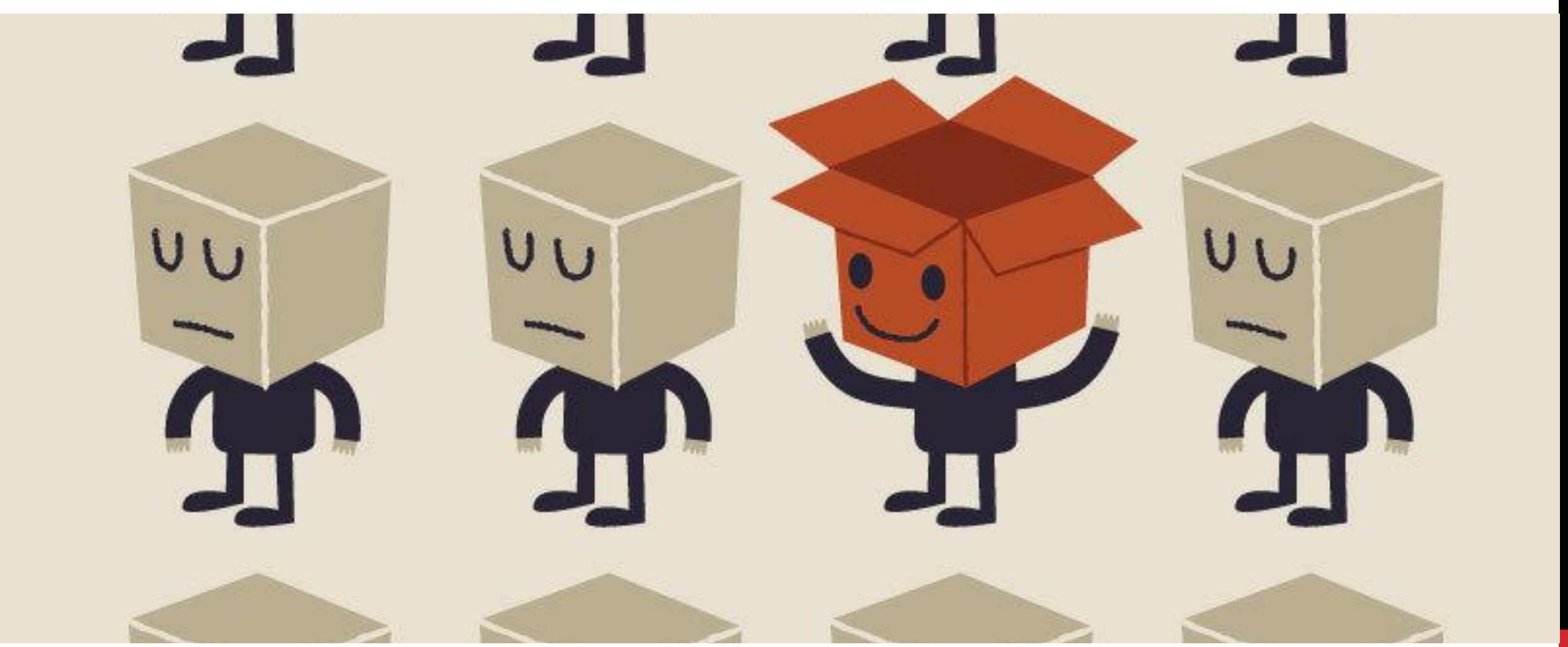


Containers



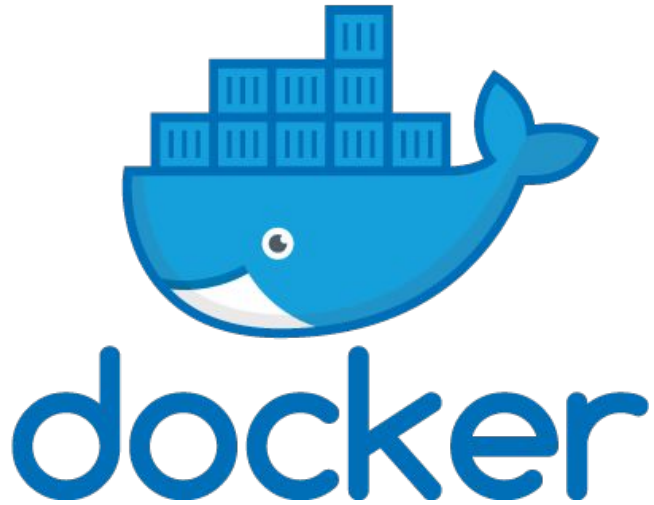
Scalable Cloud Infrastructure

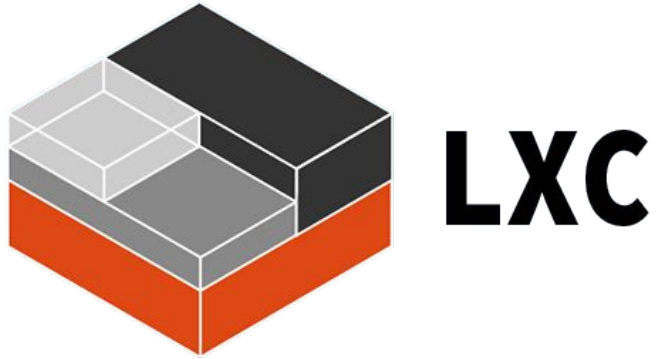
BUILDING BLOCKS



MICROSERVICES AND CONTAINERIZATION

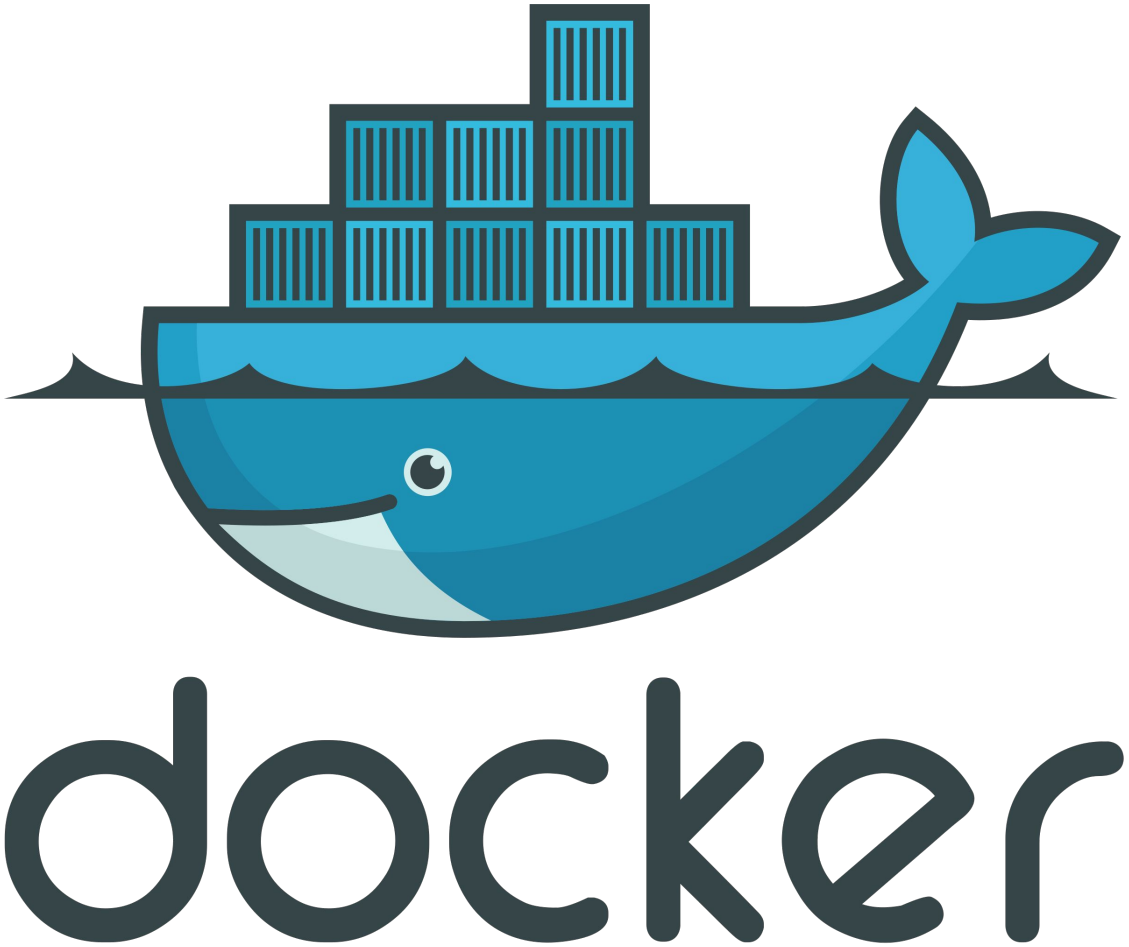
DOCKER, ROCKET, OCI AND OTHER RUNTIMES



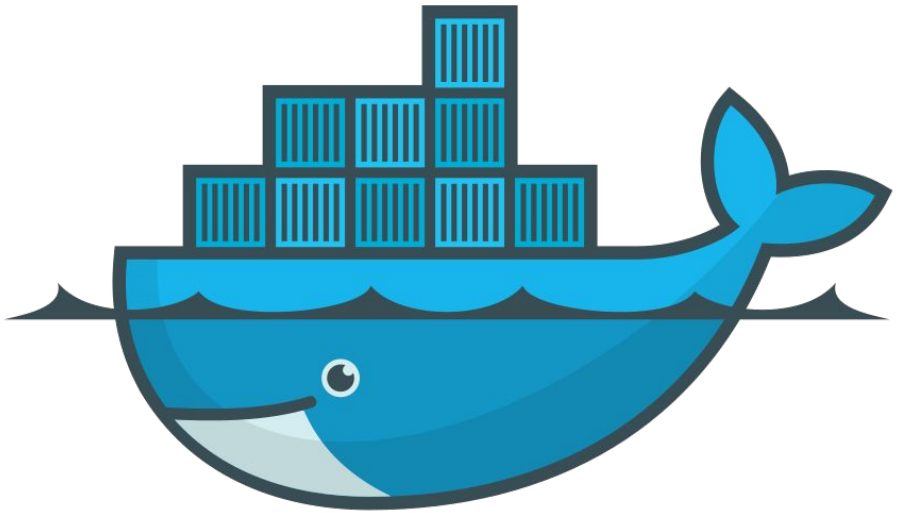


DOCKER

Build, Ship, and Run Any App, Anywhere.



DOCKER



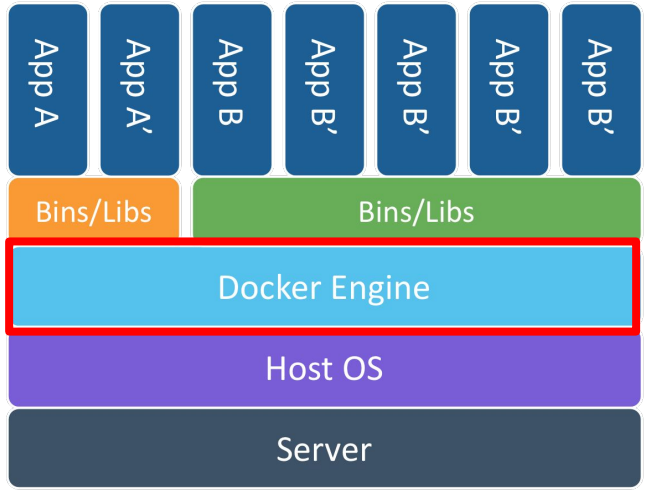
- Get more applications running on the same hardware than other technologies.
- Easy for developers to quickly create ready-to-run containered applications.
- Easier to manage and deploy applications.

DOCKER ENGINE

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

*Lightweight runtime program to **build, ship, and run Docker containers***

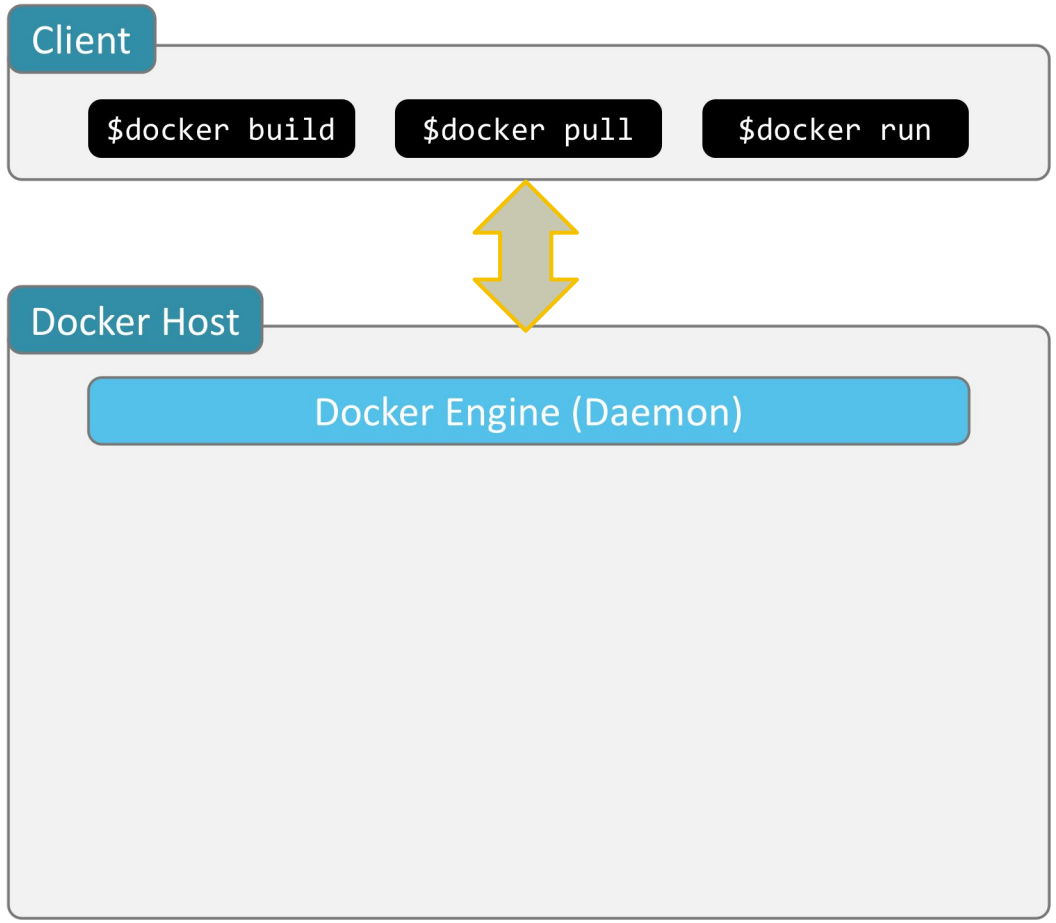
- **Also known as Docker Daemon**
- **Uses Linux Kernel namespaces and control groups**
  - **Linux Kernel (>= 3.10)**
- **Namespaces provide an isolated workspace**



DOCKER ENGINE

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

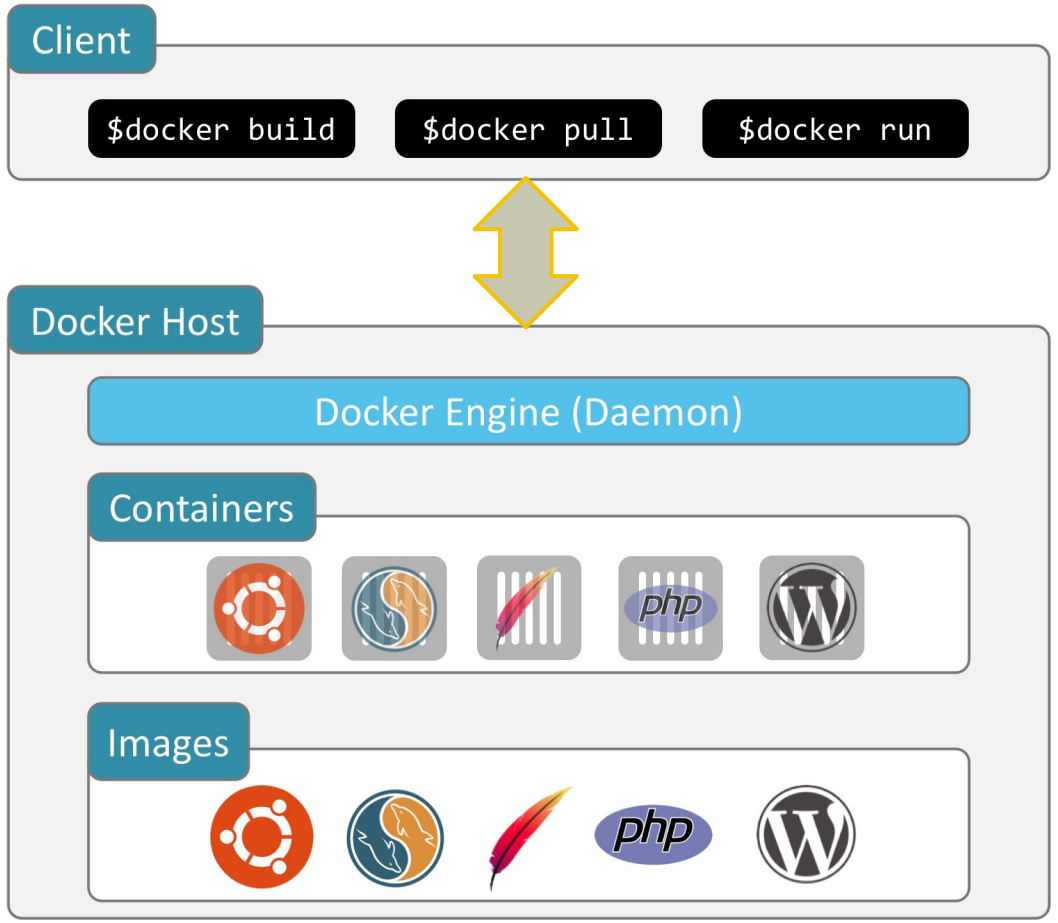
- **The Docker Client is the `docker` binary**
  - Primary interface to the Docker Host
- **Accepts commands and communicates with the Docker Engine (Daemon)**



DOCKER ENGINE

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

- **Lives on a Docker host**
- **Creates and manages containers on the host**



DOCKER REGISTRY

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Image Storage & Retrieval System

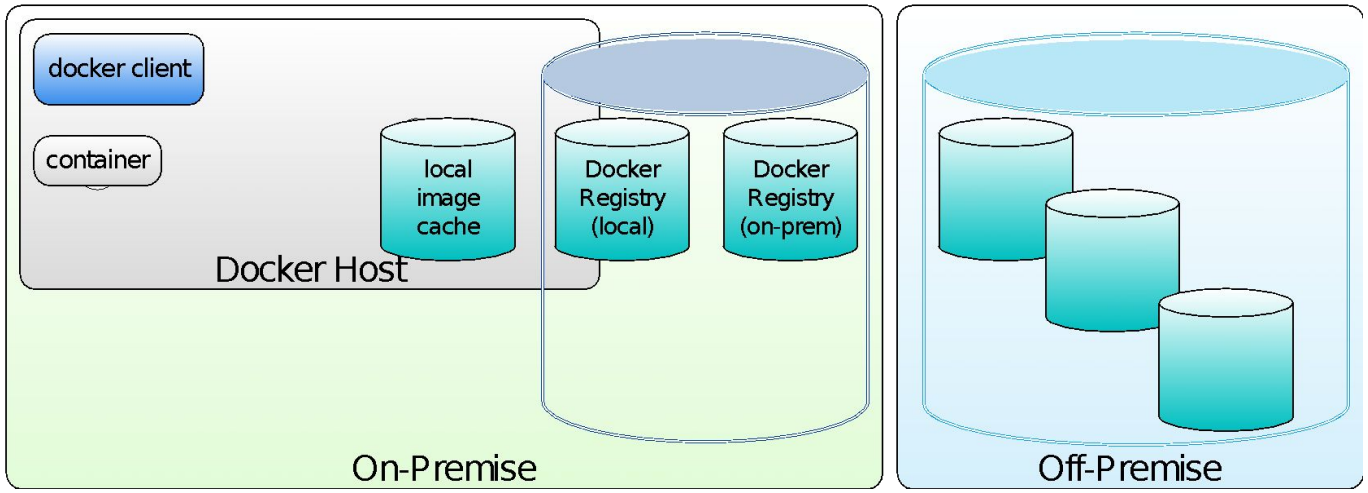
- **Docker Engine Pushes Images to a Registry**
- **Version Control**
- **Docker Engine Pulls Images to Run**

DOCKER REGISTRY

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Types of Docker Registries

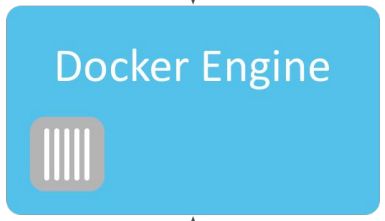
- Local Docker Registry (On Docker Host)
- Remote Docker Registry (On-Premise/Off-Premise)
- Docker Hub (Off-Premise)



DOCKER ENGINE AND REGISTRY

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Manage & Monitor



Push/Pull

Docker Engine:

- **Pushes** Images to a Registry
- **Pulls** Images to Run

DOCKER REGISTRY

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

The registry and engine both present APIs

- All of Docker's functionality will utilize these APIs

- RESTFUL API
- Commands presented with Docker's CLI tools can also be used with curl and other tools

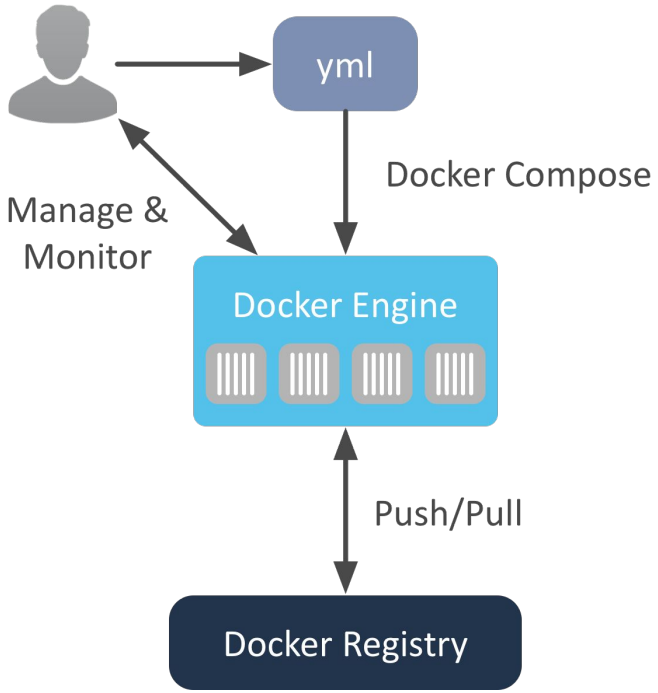
DOCKER COMPOSE

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Tool to create and manage multi-container applications

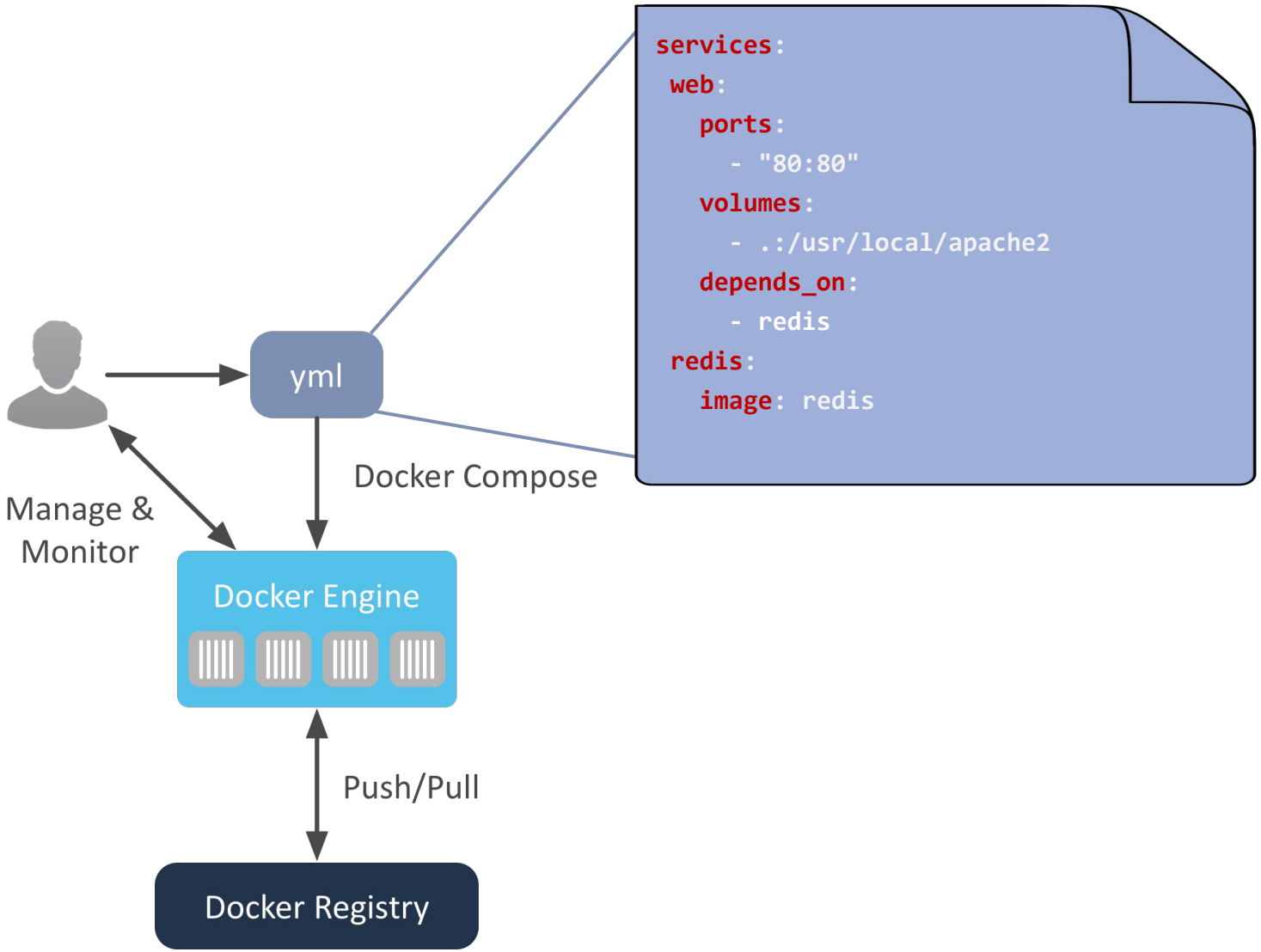
- **Applications defined in a single file:**

  docker-compose.yml
- **Transforms applications into individual containers that are linked together**
- **Compose will start all containers in a single command**



DOCKER COMPOSE

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

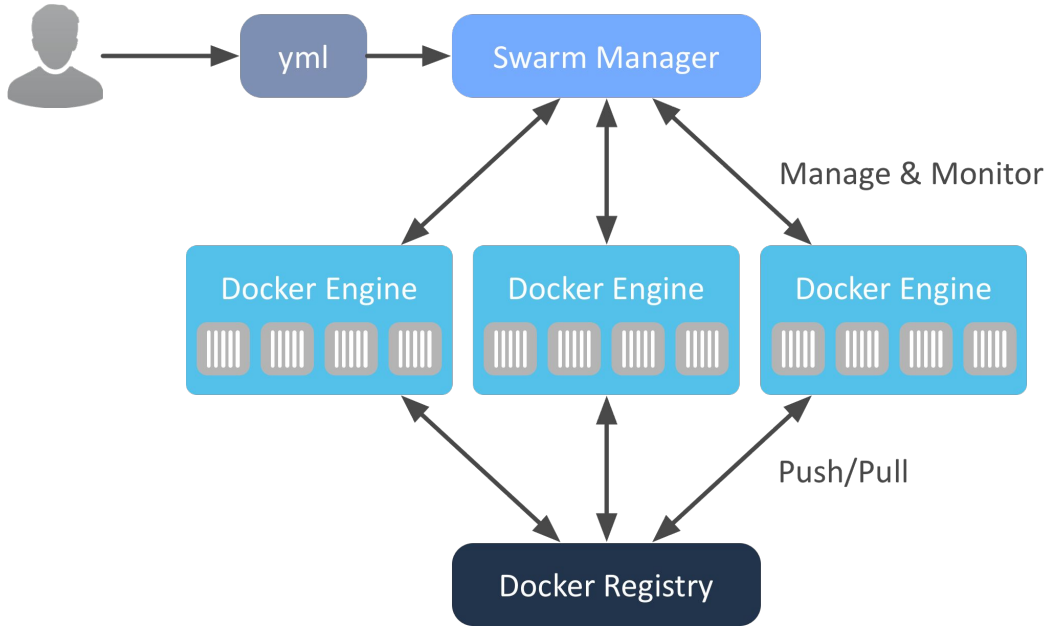


DOCKER SWARM

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

Clusters Docker hosts and schedules containers

- **Native Clustering for Docker**
  - Turn a pool of Docker hosts into a single, virtual host
  - Serves the standard Docker API

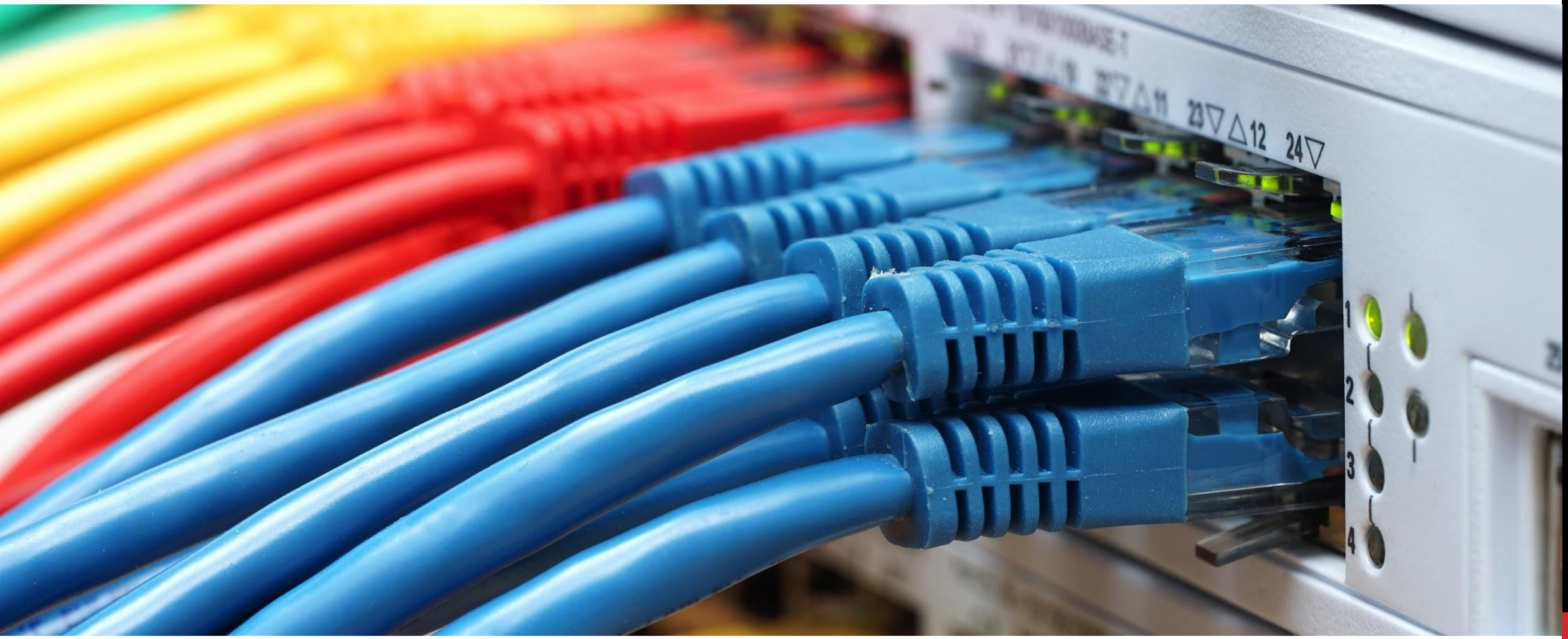


COREOS RKT (ROCKET)

A security-minded, standards-based container engine.

OPEN CONTAINER INITIATIVE (OCI)

The Open Container Initiative (OCI) is a lightweight, open governance structure (project), to create an open industry standards around container formats and runtime.



INTERFACES

INTERFACES

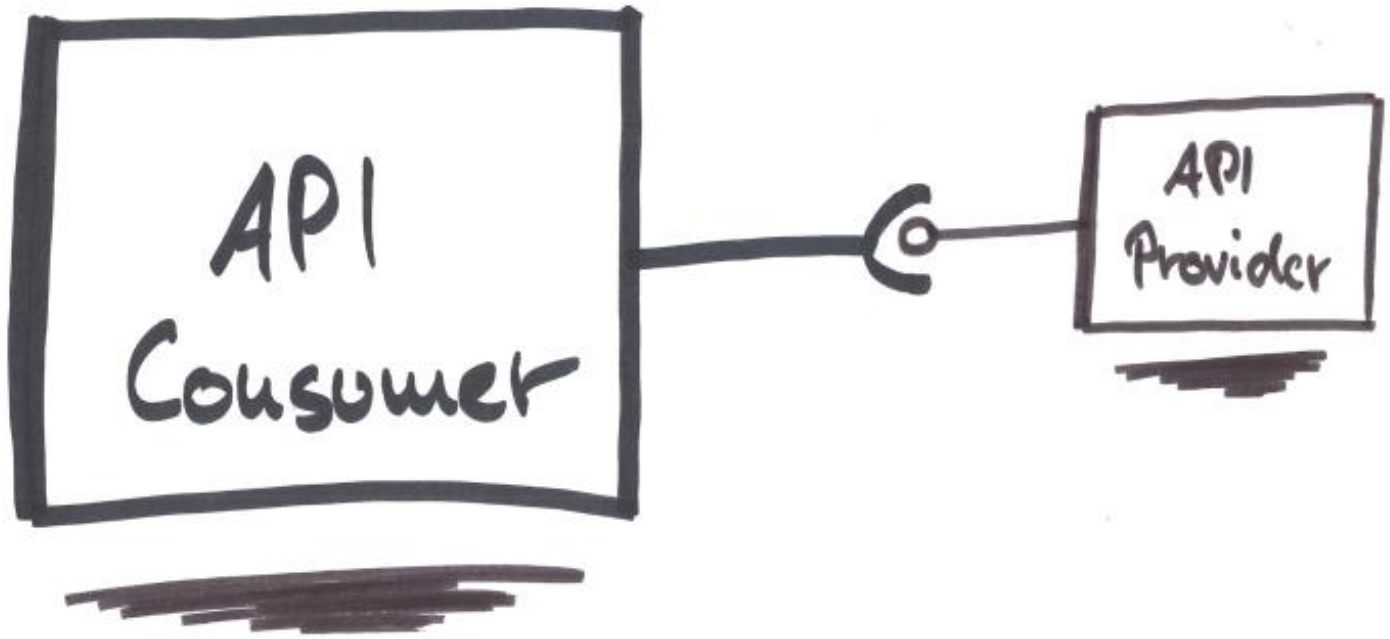
Designing the endpoint interface

- Data formats are generally either JSON or XML
- Asynchronous services may use JMS or other messaging protocol (including HTTP)
- Synchronous services preferably run on HTTP
- Alternative protocols limits service interoperability

CONTRACTS

CONTRACTS

Consumer driven contracts (CDC) are like TDD applied to the API.



COHESION

Definition

The degree to which the elements within a component belong together.

Think single responsibility principle and single reason to change.

COUPLING

Definition

The degree to which components have knowledge of other components.

Think well-defined interfaces, inversion of control etc.

OBSERVABILITY

Definition

Observability is the activities that involve measuring, collecting, and analyzing various diagnostics signals from a system.

These signals may include metrics, traces, logs, events, profiles and more.

COUPLING, COHESION AND CONTINUOUS DELIVERY

Design

- Cohesion makes reasoning easy
- Loose coupling reduces impact

Build, unit and integration

- Cohesion limits dependencies
- Loose coupling allows simulation

End-to-end testing

- Cohesion/coupling orchestration

Deployment

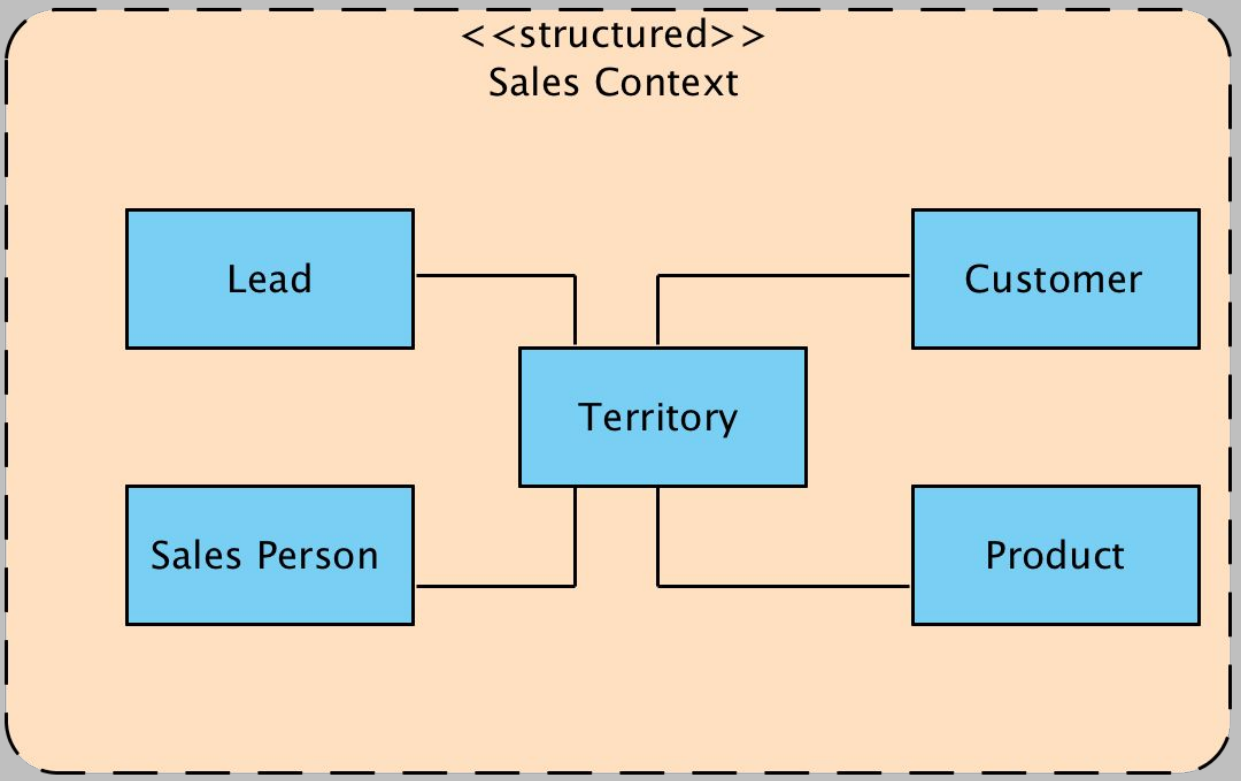
- Cohesion minimizes number of components in play
- Coupling leads to “monoliths”

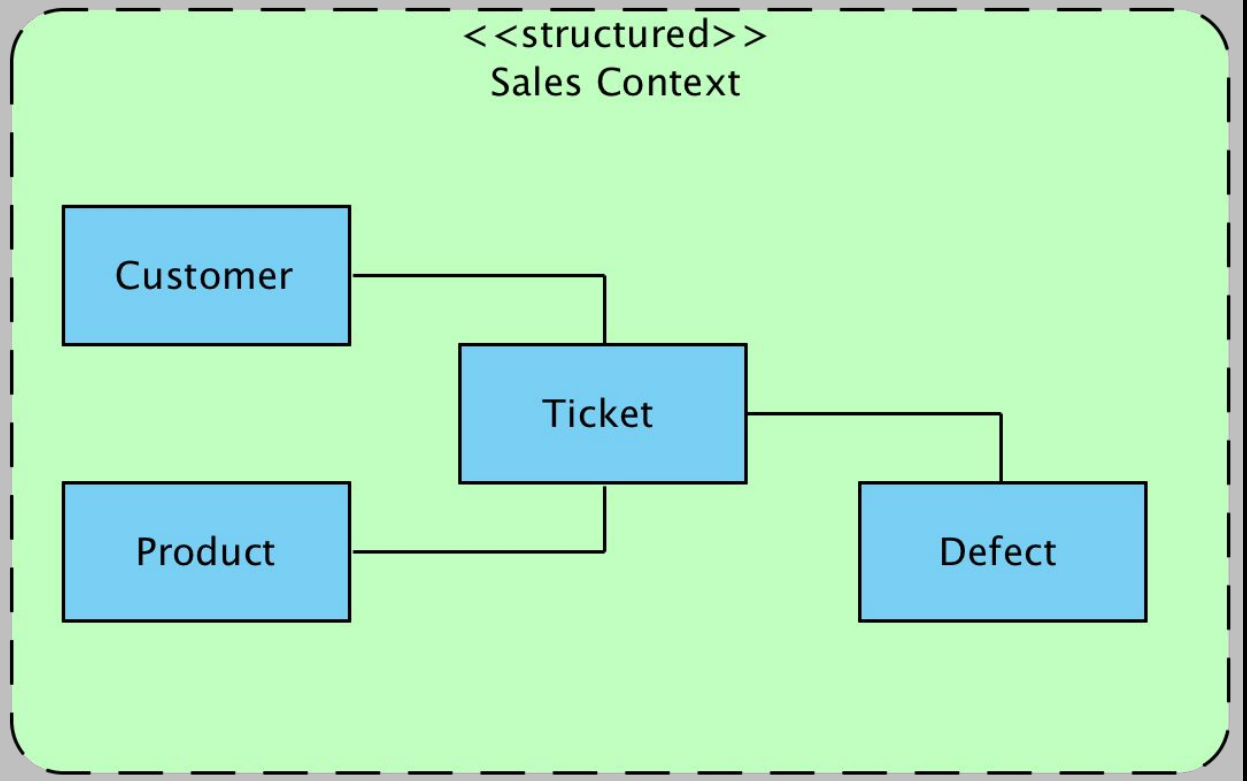
Observability

- Cohesion is easier to understand
- High coupling typically leads to large blast radius and “murder mystery” style debugging.

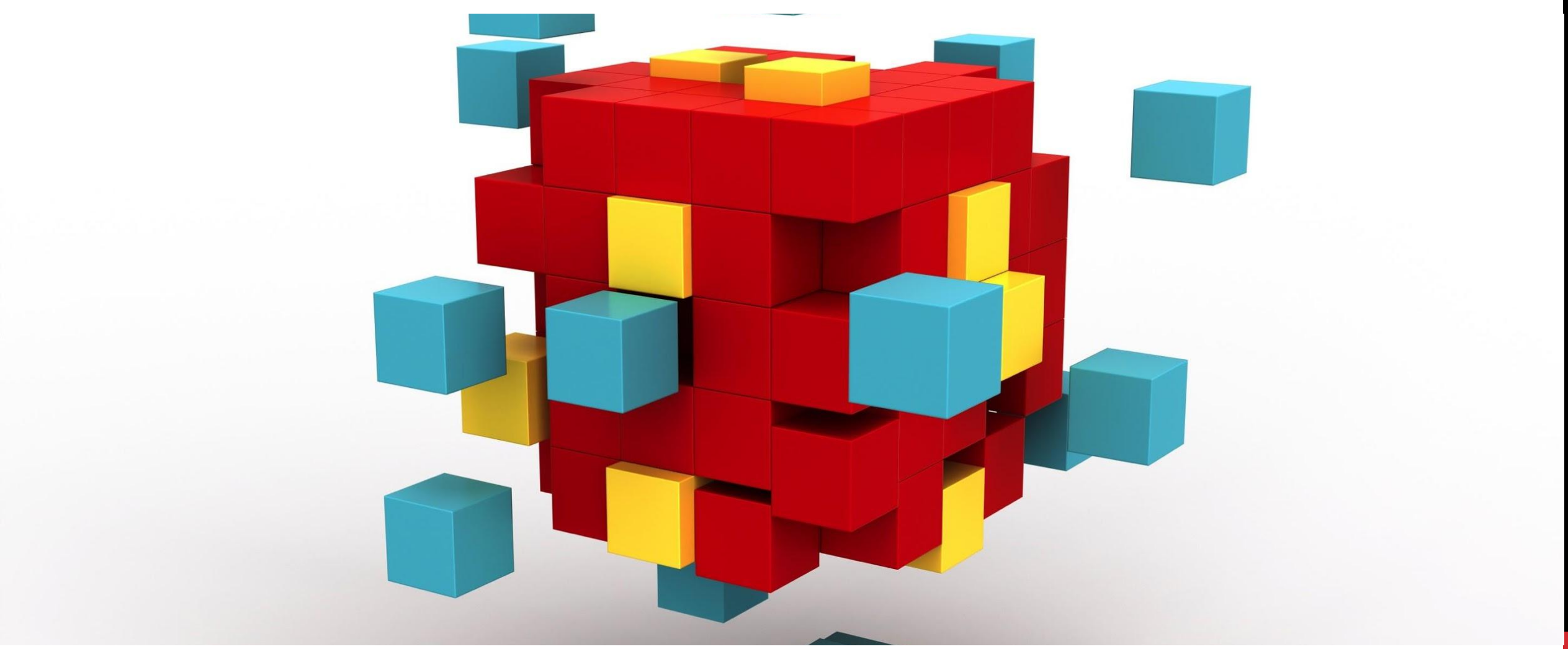
BOUNDED CONTEXT

- Bounded context is a logical boundary.
- A bounded context clarifies, encapsulates, and defines the specific responsibility to the model.
- It ensures the domain will not be distracted from the outside.
- Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.
- This kind of boundary is based on concrete technical implementation and it must not have linguistic ambiguity because, first and foremost, it's a linguistic boundary.





BOUNDED CONTEXT



SERVICE DECOMPOSITION AND RE-COMPOSITION

Before Decomposition

SERVICE DECOMPOSITION

Definition

Method of analysis that dissects a complex process to show its individual elements.

SERVICE RE-COMPOSITION

Definition

Method of analysis that reconnect complex processes into a broader contextual service.

BREAKING DOWN MONOLITHS

"The main characteristics of a microservices-based applications are defined in Microservices, Monoliths, and NoOps.”

“They are functional decomposition or domain-driven design, well-defined interfaces, explicitly published interface, single responsibility principle, and potentially polyglot.

Each service is fully autonomous and full-stack. Thus changing a service implementation has no impact to other services as they communicate using well-defined interfaces.

There are several advantages of such an application, but it’s not a free lunch and requires a significant effort in NoOps."

PRINCIPLE: SINGLE RESPONSIBILITY

Single Responsibility Principle

- The Single Responsibility Principle states that a service should have only one reason to change
- The main responsibility of the service is considered to be one reason to change
- If there are two reasons for change, we need to split the functionality of the service into two services

Two services should not share one responsibility

- Replication of functionality becomes maintenance burden

One service should not have multiple responsibilities

- The responsibilities then become tightly coupled

PRINCIPLE: AUTONOMY

A microservice must be

- Self-contained
- Independently deployable
- Have full responsibility for a capability and executing that capability

Autonomy is a primary differentiator between microservices and other architectures

- Minimizes coupling and maximizes abstraction

POTENTIAL CANDIDATES FOR MICROSERVICES

Situations where microservices might not be best

- Organizations with strong policies and practices oriented around centrally managed, heavy or monolithic components
- Organizations where processes are based on waterfall, lengthy development cycles, and heavyweight lifecycle processes

MICROSERVICE BOUNDARIES

MICROSERVICE BOUNDARIES

As with most design decisions, many metrics are arbitrary

Single responsibility principle is the most important metric

A bounded context is a subdomain that is responsible for a particular function

- Strongly isolated, business function or capability

Several approaches can help with identifying bounded contexts

- Top-down domain decomposition
- Top-down use case decomposition
- Migration from existing, monolithic system

ESTABLISHING BOUNDARIES

Considerations in establishing boundaries

- Autonomous unit of functionality
- Size of deployable component
- High value unit when migrating from monolithic application
- Need for transactional integrity is a major consideration
- Scaling considerations
- Isolation considerations
- Volatility and evolutionary considerations

BOUNDARY CONSIDERATIONS

Autonomous unit of functionality

- Identify dependencies on external resources
- Work to minimize those dependencies
- If can eliminate dependencies, good boundary to consider

Size of deployable component

- Large deployable units present challenges for automated processes
- Constrict boundaries to support managing deployable units

BOUNDARY CONSIDERATIONS (CONT'D)

High value unit when migrating from monolithic application

- Look for functions with highest or most valuable usage

Need for transactional integrity is a major consideration

- Transaction boundaries should not cross service boundaries

BOUNDARY CONSIDERATIONS (CONT'D)

Scaling considerations

- A function may have different scaling needs than other, related functions
- Establishing a boundary based on scaling can make sense

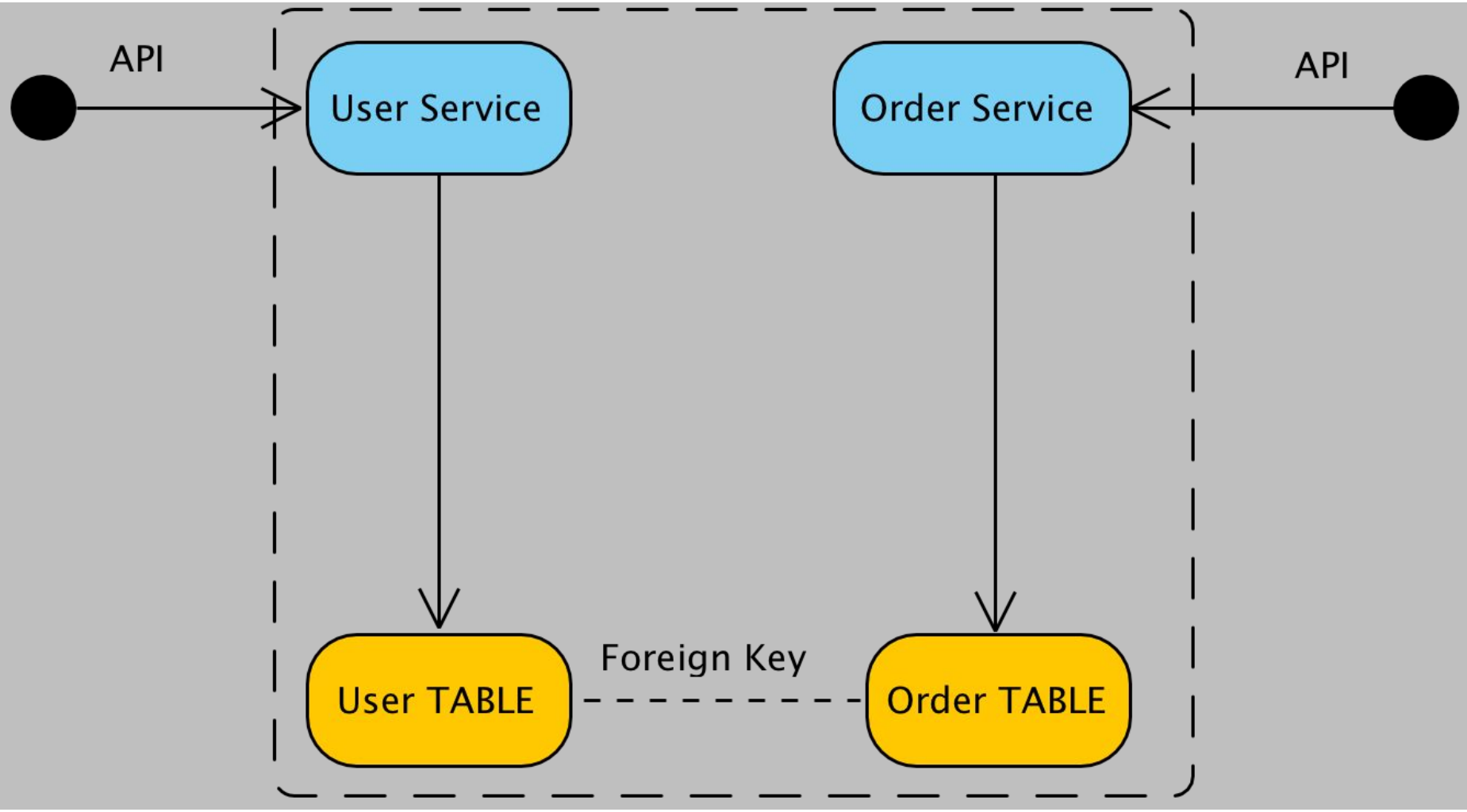
Isolation considerations

- Function should be easily detachable from larger system
- Strive for high cohesion within the service and loose coupling between services
- View a microservice as a product by itself
    - Targeted consumers, flexibility, marketability, etc.

BOUNDARY CONSIDERATIONS (CONT'D)

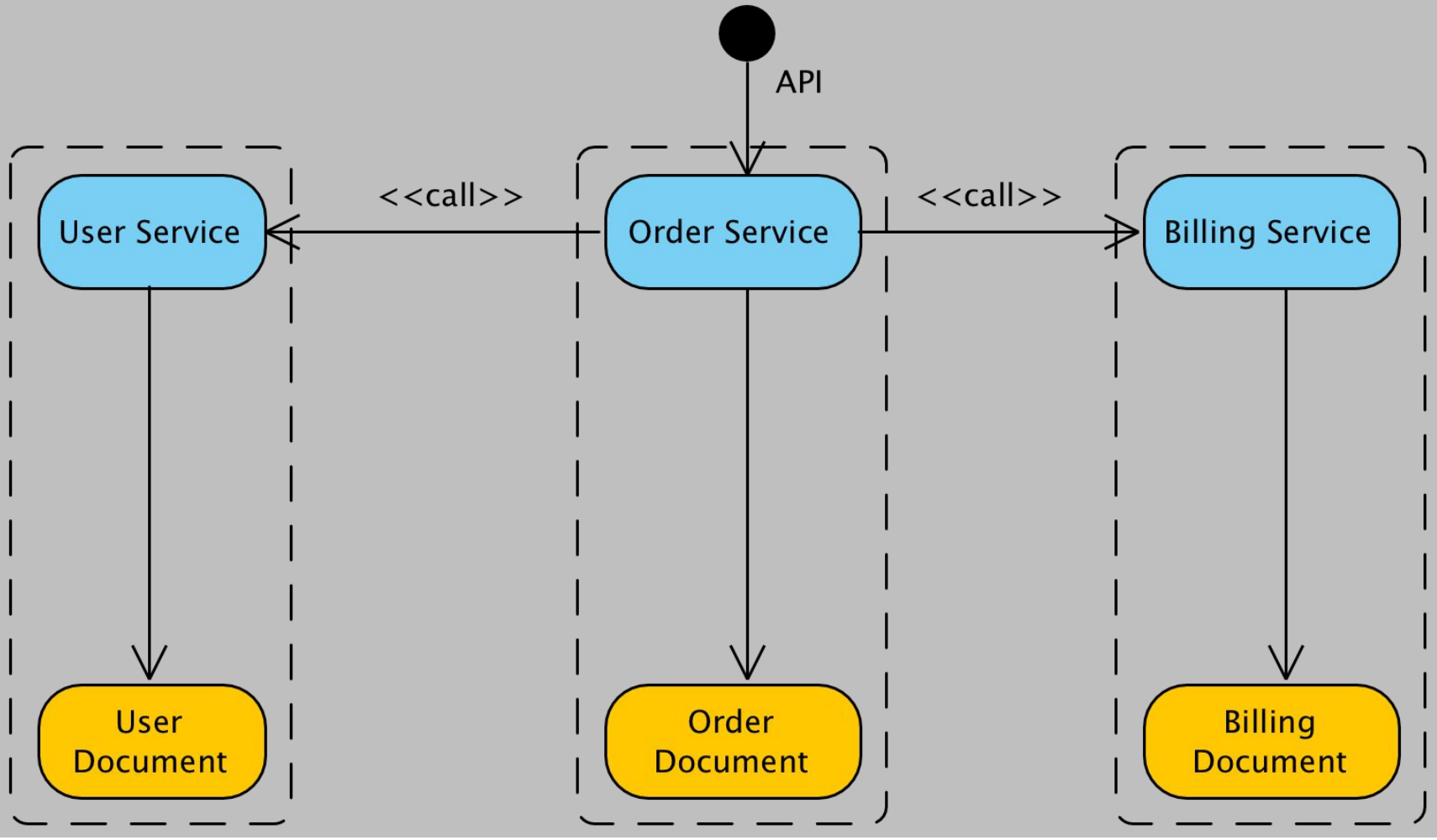
Volatility and evolutionary considerations

- New or experimental "products" may be a bounded context
- Develop Minimum Viable Products (MVPs)
- Allow system to evolve



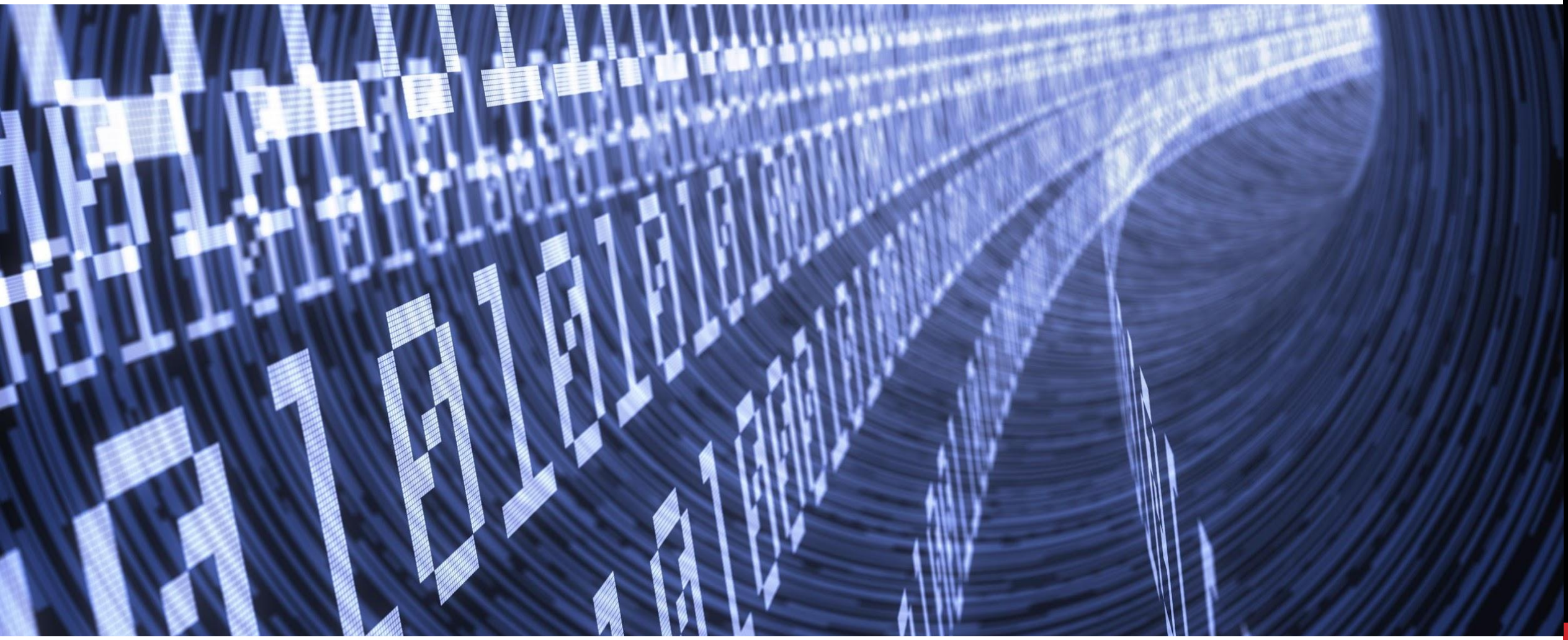
DECOMPOSITION

Before Decomposition

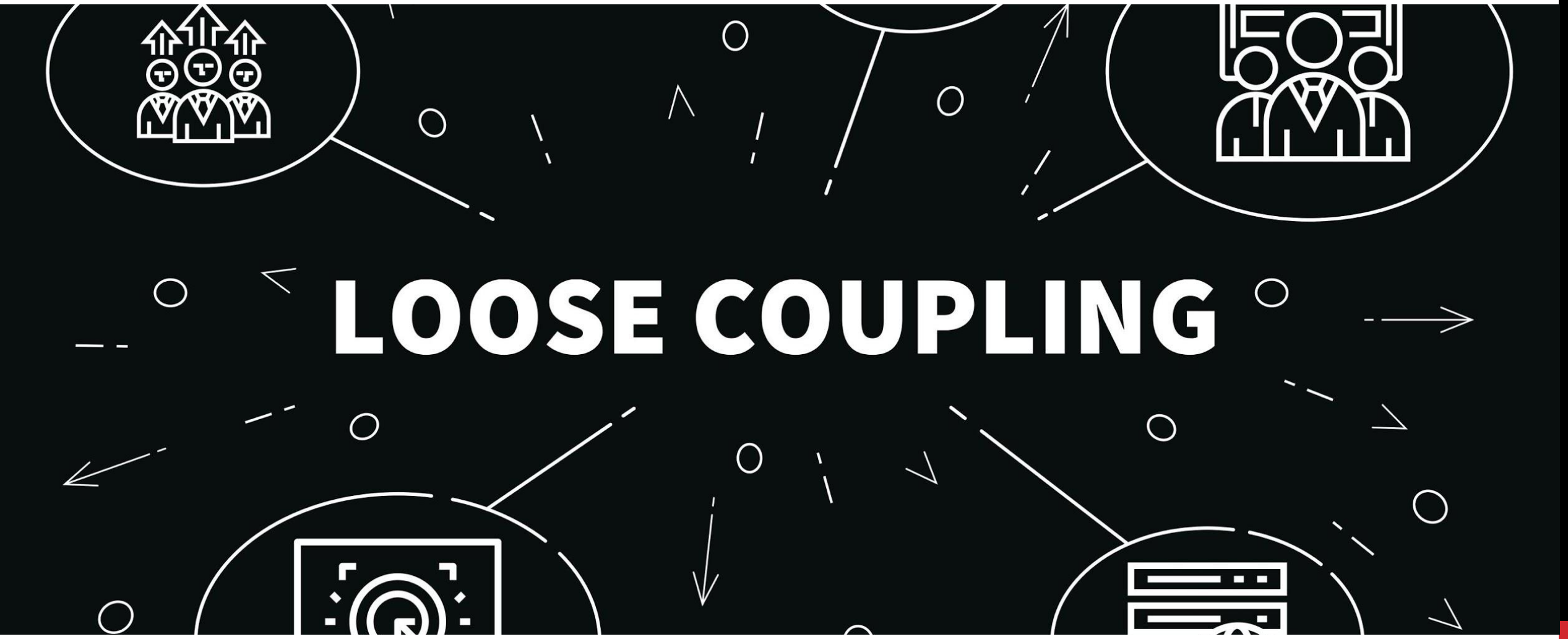


DECOMPOSITION

After Decomposition



MICROSERVICES COMMUNICATIONS



LOOSELY COUPLED SYSTEMS

LOOSELY COUPLED SYSTEMS

Characteristics

- **Loose coupling is an approach to interconnecting the components in a system or network so that those components, also called elements, depend on each other to the least extent practicable.**
- **Coupling refers to the degree of direct knowledge that one element has of another.**

COMMUNICATION STYLES

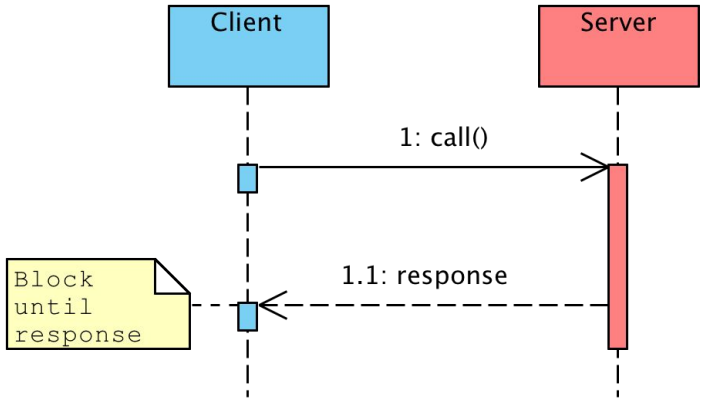
SYNCHRONOUS

- Client connected at same time
- Real-time Conversation
- Limited participants

ASYNCHRONOUS

- Client not connected at same time
- ‘Delayed’ conversation
- Unlimited participants

COMMUNICATION STYLES (SYNCHRONOUS)



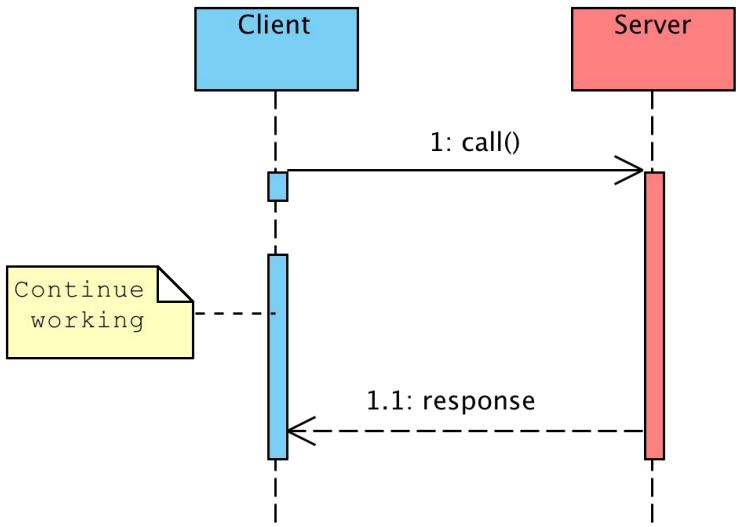
Advantages:

- Service is stateless (relative to caller)
- Allows many active instances
- Lower management overhead since no message server
- Immediate and direct feedback (success or failure)

Disadvantages:

- Caller has to wait, potentially limiting caller scalability
- Can add hard dependencies between services if chained

COMMUNICATION STYLES (ASYNCHRONOUS)



Advantages:

- Improves scalability since no coupling between caller and service instance
- Queueing allows for degree of inherent load balancing

Disadvantages:

- Dependent on external message server
- Additional tuning and monitoring is required

COMMUNICATION STYLES

Both styles have their places

- Asynchronous is typically more scalable, synchronous less resource dependent
- Caller may be waiting for a response no matter which style is used
- Often start with synchronous style and move to asynchronous when and if it makes sense

ASYNCHRONOUS OPERATIONS

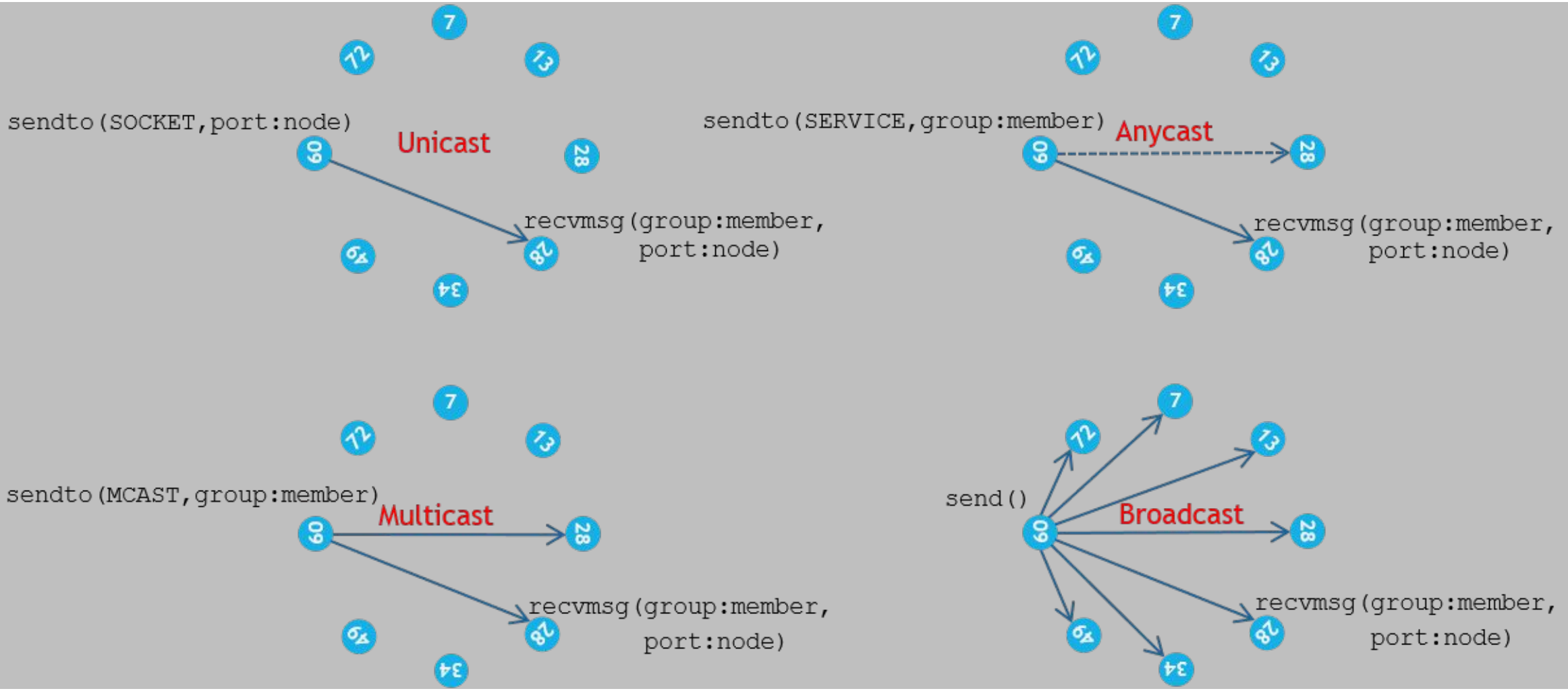
Asynchronous operations can be carried out in a number of ways

- Lightweight messaging system
  - RabbitMQ (AMQP / STOMP)
  - ActiveMQ (AMQP / STOMP)
- Non-blocking remote calls
  - AJAX
  - WebSockets
  - Reactive-Streams
  - Server Sent Events (SSE)

COMPARING MESSAGING PATTERNS

Comparing messaging patterns

- Pub/Sub
- Multicast
- Anycast
- Streaming

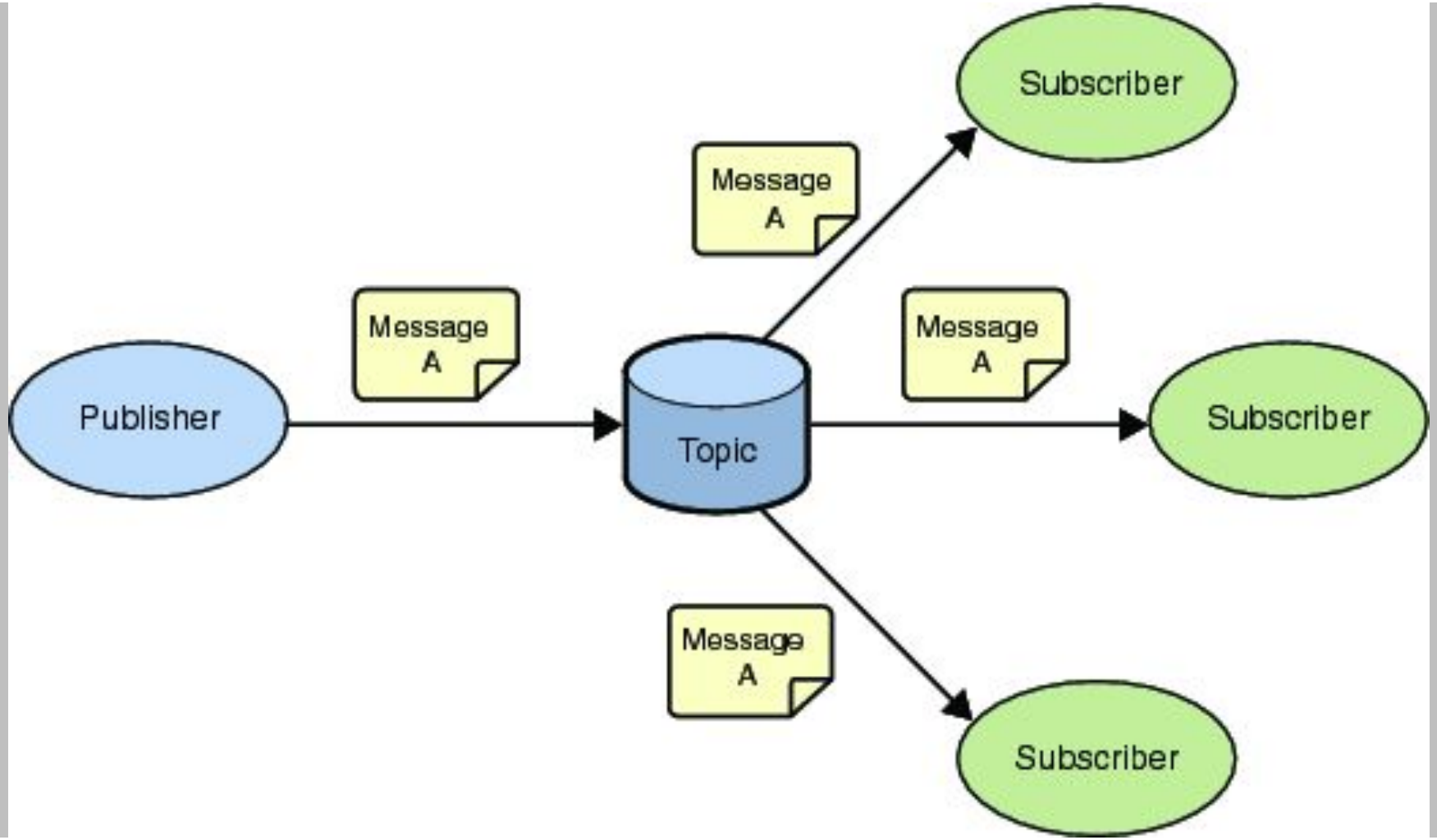


COMPARING MESSAGING PATTERNS

MESSAGING PATTERNS

Publish & Subscribe (Pub/Sub)

- **Messaging pattern where publishers push messages to subscribers.**
- **Messaging provides instant event notifications for distributed applications.**

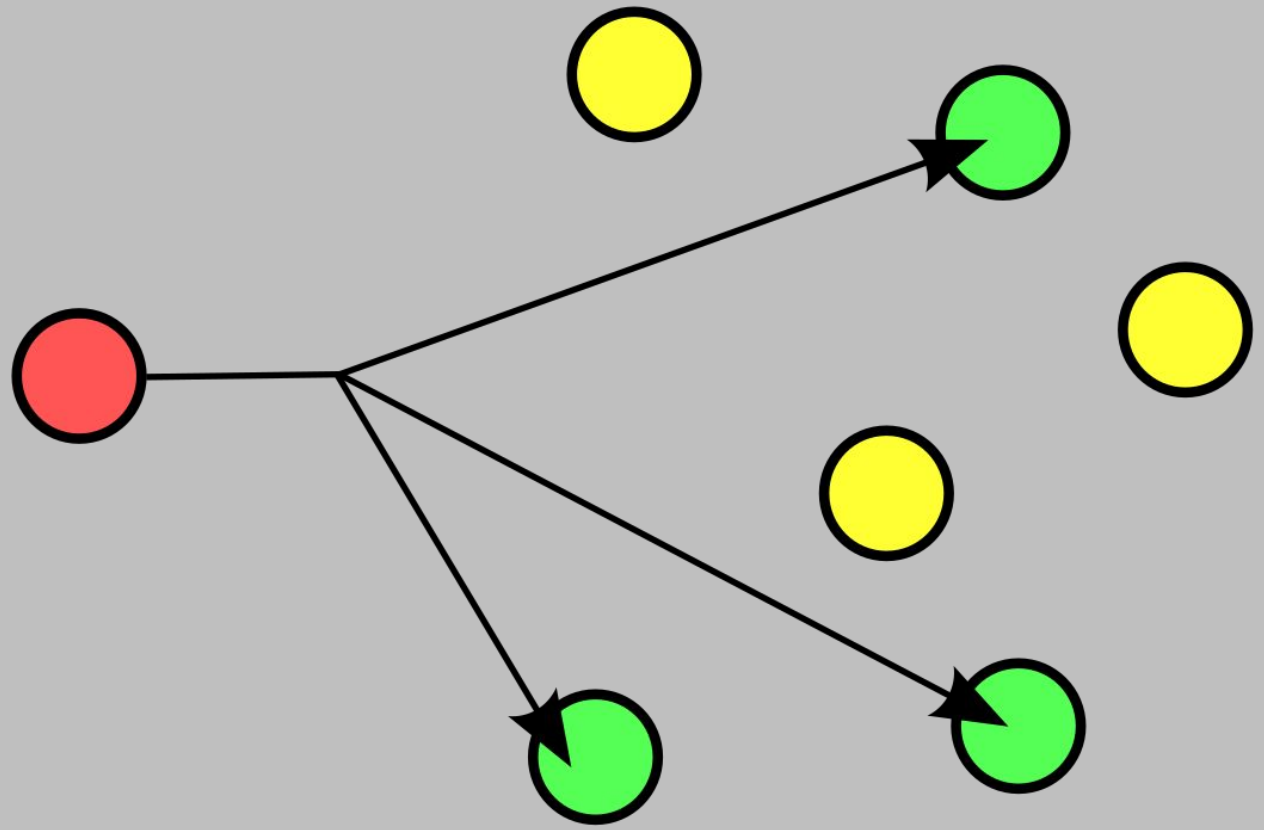


PUBLISH & SUBSCRIBE (PUB/SUB)

MESSAGING PATTERNS

Multicast

- **Method of routing data that allows a single sender, or a group of senders, to communicate efficiently with a group of receivers.**
- **Supports one-to-many routing, in which a single device sends data to a group of devices.**

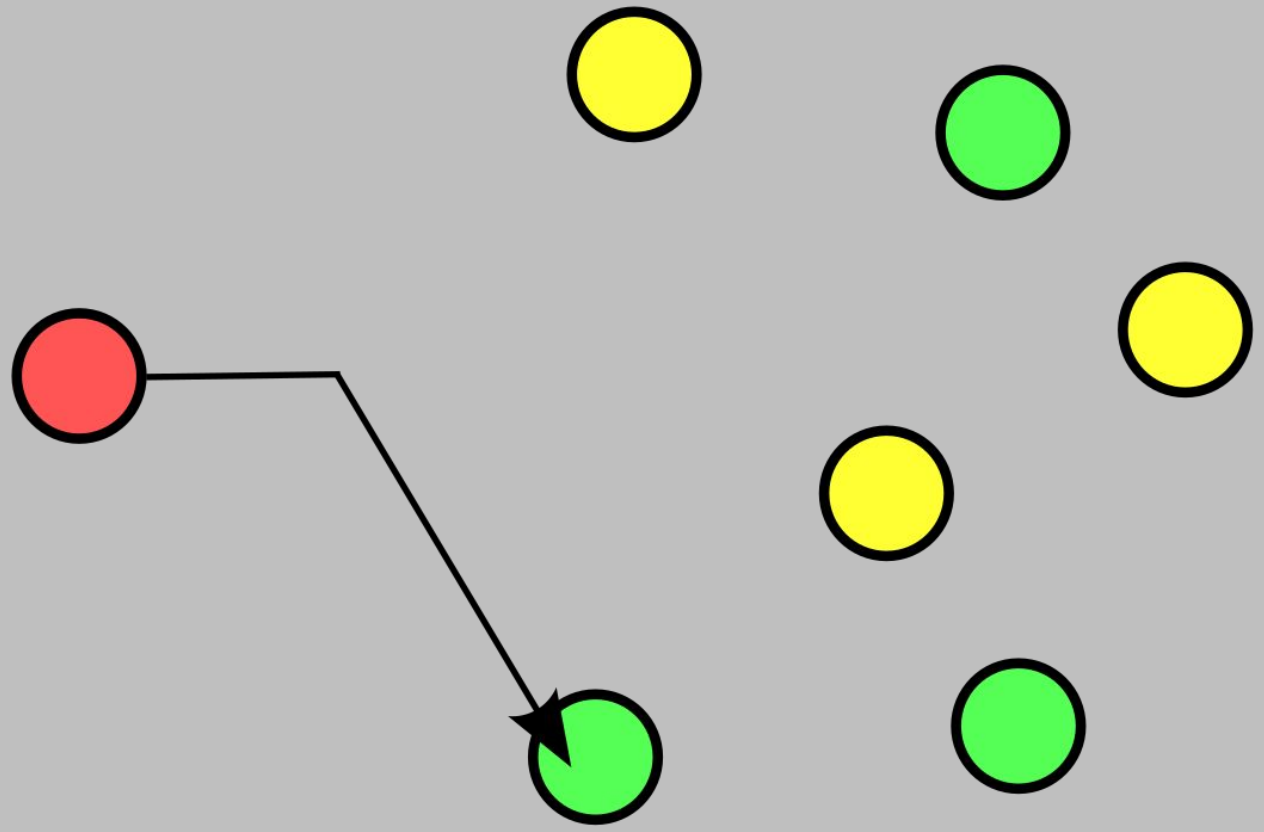


MULTICAST

MESSAGING PATTERNS

Anycast

- Method for routing Internet traffic.
- Specially-programmed router can dynamically determine the best destination for data packets.
- When data is requested, traffic is routed to different servers depending on a set of determining factors.
- Regardless of which server is chosen as the destination, the client receives the same data.



ANYCAST

MESSAGING PATTERNS

Streaming

- **Streaming transmits data as a continuous flow, which allows the clients to utilize the data almost immediately.**

MESSAGING COMPARISON

MESSAGING

- **Framed message based protocol**
- **Per event or operation**
  - e.g. Customer order fulfillment

STREAMING

- **Unframed data (bytes) stream based protocol**
- **Complex analytics:**
  - e.g. Microservice logging

EXAMINING DIFFERENT MESSAGE BROKERS

BROKERS

- RabbitMQ
- Kafka
- ZeroMQ
- ActiveMQ

PROTOCOLS

- AMQP
- STOMP
- MQTT

RABBITMQ

- Specifically designed for lightweight web messaging
- Many language supported
- Supports many messaging protocols:
  - SMTP, STOMP and HTTP
- Doesn’t limit clients to one time messages
- Consumers can retrieve different message at one time

ADVANCED MESSAGE QUEUING PROTOCOL (AMQP)

- **Created a functional method of accord between middle-ware and client**
- **Distribution protocol model**
- **Includes various messaging systems, such as:**
  - point-to-point, publish, fan-out, subscribe and request-response