istic Informatique Électronique

Cloud native applications

understanding the notion of micro-services

Brice Ekane - (brice.ekane@univ-rennes.fr)

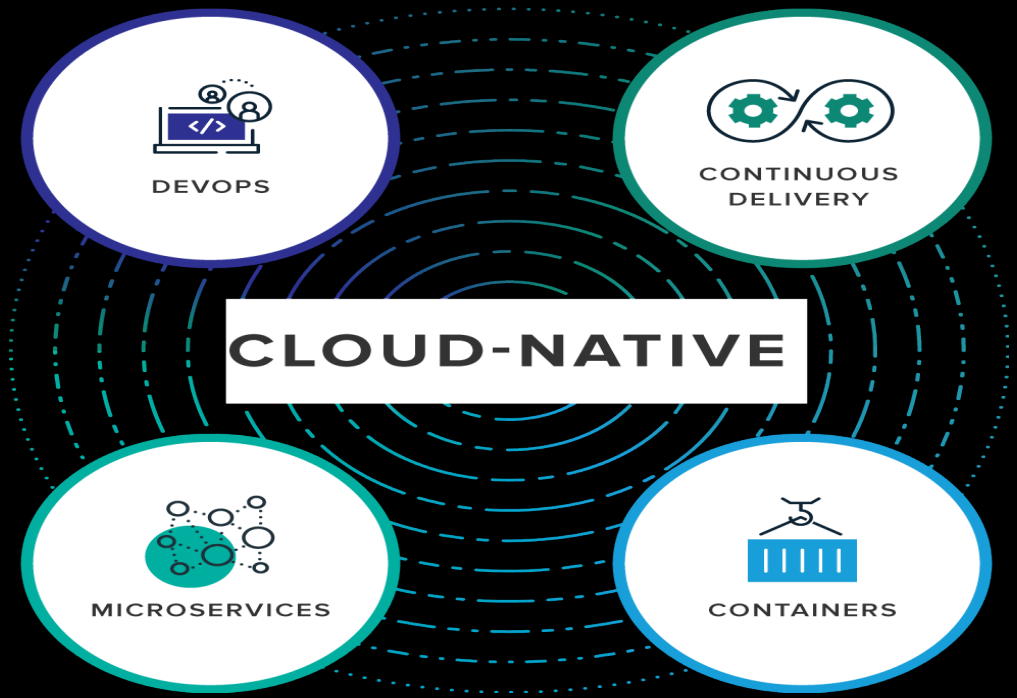
ISTIC Rennes - France

2024-2025

git clone https://github.com/bekane/tlc.git

Cours repris de Pr. Olivier Barais

Cloud native application



Cloud Native

Definition

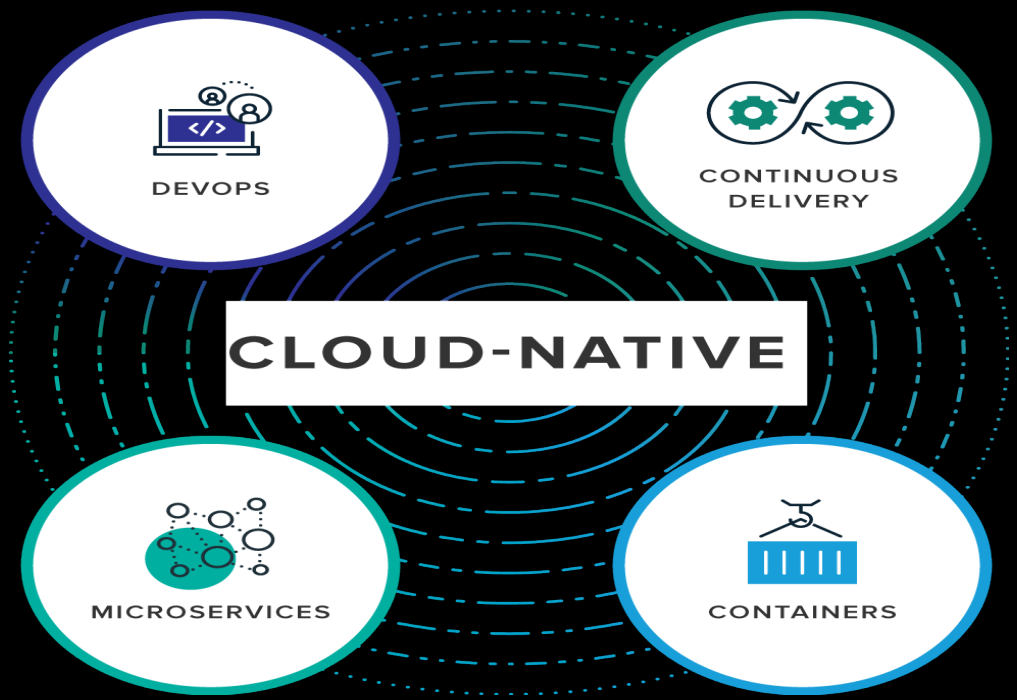
- Cloud-native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model.
- Cloud-native is about how applications are created and deployed, not where.
- Not related to public or private cloud.

Benefits of Cloud-Native

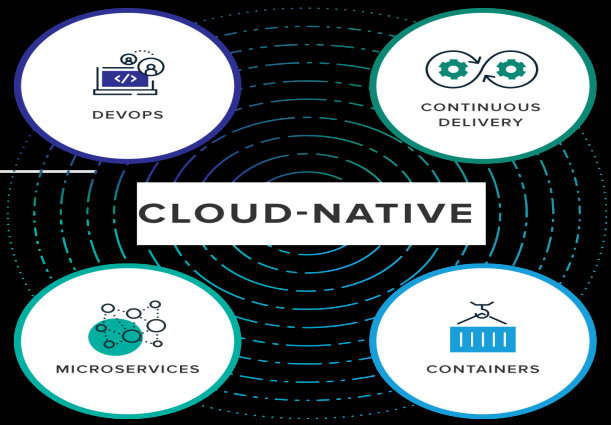
Key Benefit

When companies build and operate applications in a cloud-native fashion, they bring new ideas to market faster and respond sooner to customer demands.

Cloud native application



Cloud-Native Applications



Key Components

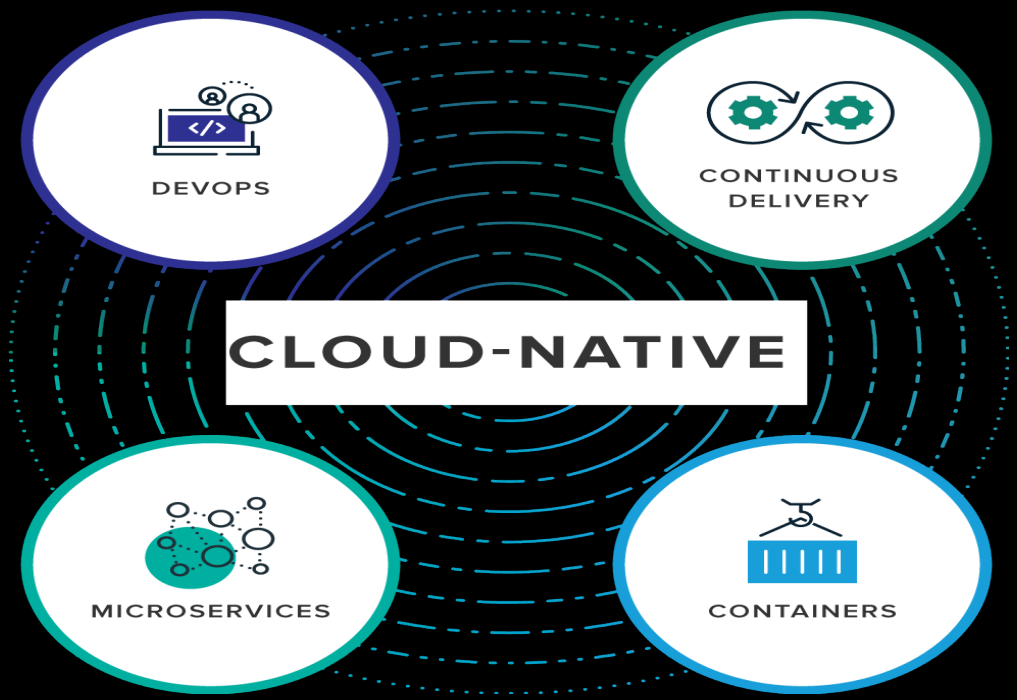
- **Microservices**: Architectural principle of distributed apps.
- **Containers**: Shipping and OS mechanism.
- **DevOps**: Objectives, team mindset, continuous feedback loop.
- **Continuous Delivery**: Mechanics.

Talk Outline

Overview

- **Micro-Services**
  - API management
- **Continuous Delivery**
  - Continuous integration
  - Continuous deployment
  - Build tool

Cloud native application



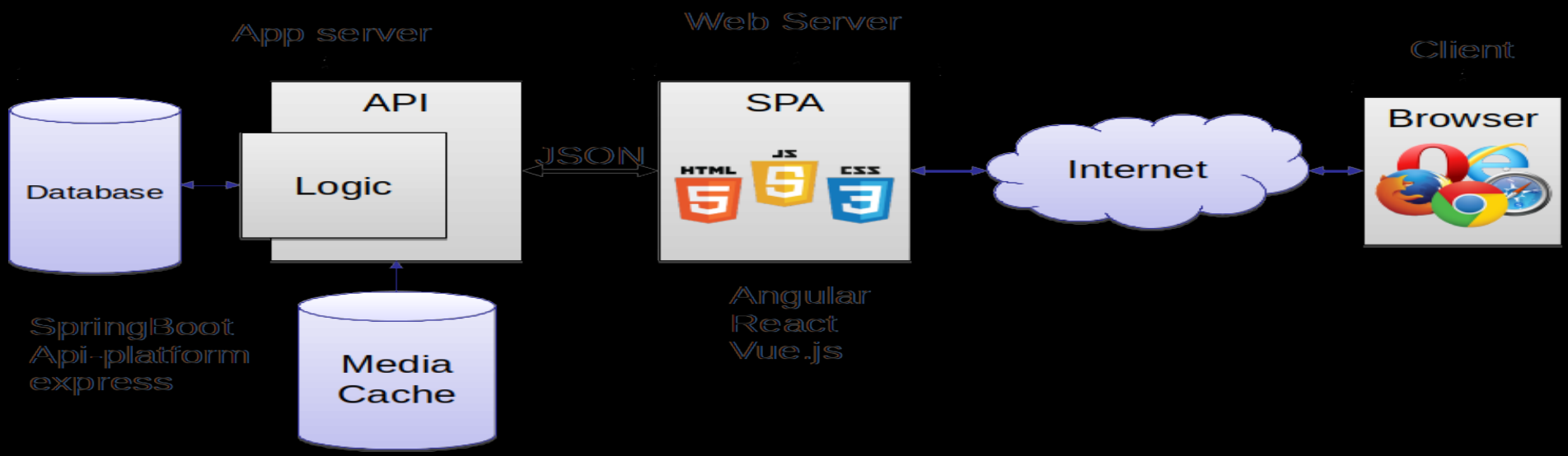
Figure

MICROSERVICES

Modern Software Architecture

Server Side

- Server Side: Spring Boot, Express, API Platform
- Client Side: Angular, React, Vue



Definition of Microservices Architecture

Key Characteristics

- A microservices architecture consists of a collection of small, autonomous services.
- Each service is self-contained and should implement a single business capability.

Definition 2: Microservices Architecture

What are Microservices?

Microservices is an architectural approach to developing an application as a collection of small services. Each service:

- Implements business capabilities.
- Runs in its own process.
- Communicates via **HTTP APIs** or **messaging.**

Characteristics of a Microservice

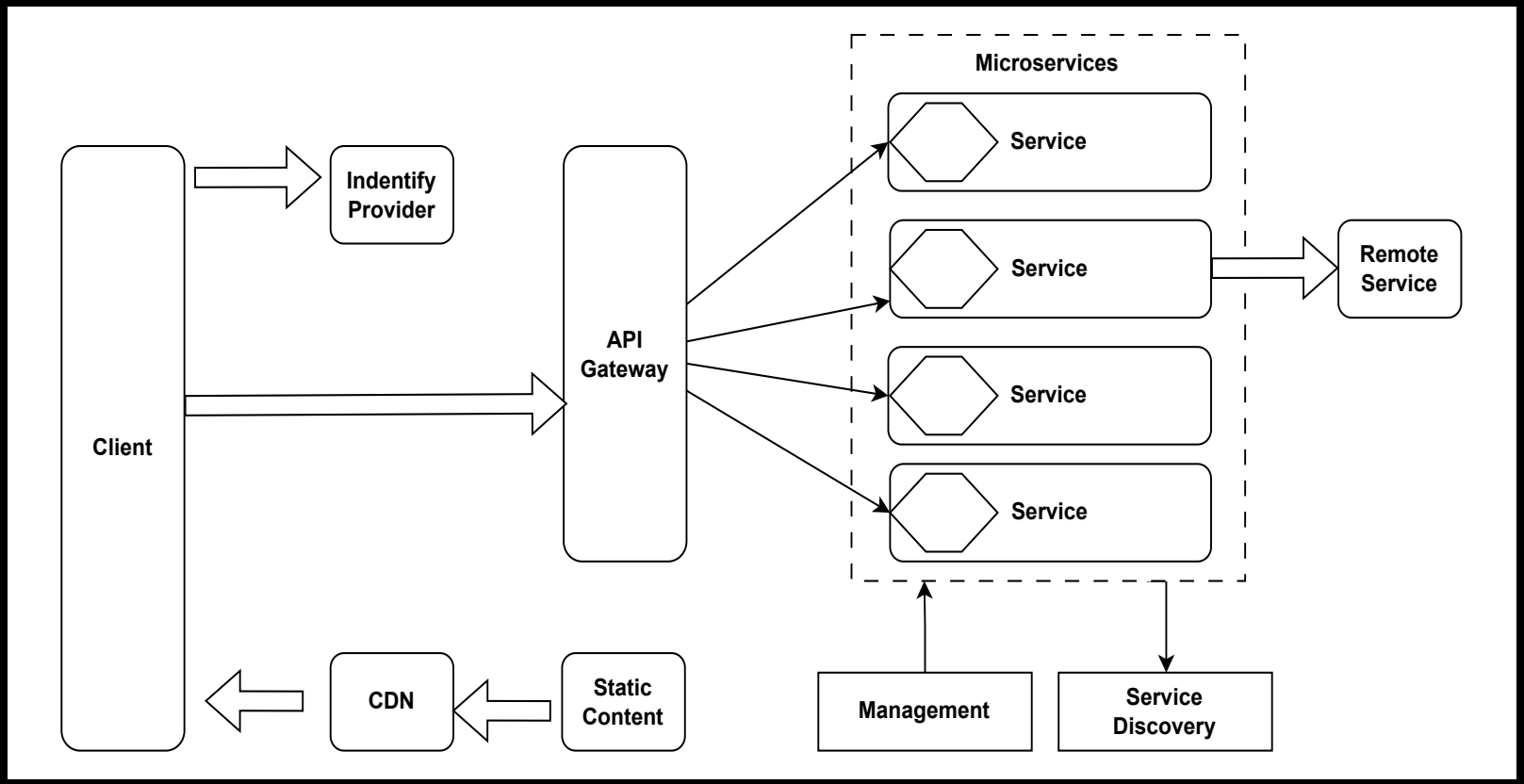
Key Features

- Services are small, independent, and loosely coupled.
- Each service is a separate codebase.
- Services can be deployed independently.
- Services are responsible for persisting their own data or external state.
- Services communicate with each other by using well-defined APIs.
- Services don't need to share the same technology stack, libraries, or frameworks.

FROM A MONOLITHIC APPLICATION

TO MODULAR SERVICE

Microservices Overview



Other Components in a Typical Microservices Architecture

Key Components

- Service Discovery
- API Gateway
- Management
- Identity Provider

Advantages of Using an API Gateway

Key Benefits

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as:
  - Authentication
  - Logging
  - SSL termination
  - Load balancing

When to Use This Architecture

Key Scenarios

- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.
- An organization that consists of small development teams.

Benefits of Microservices Architecture

Key Advantages

- Independent deployments
- Independent development
- Small, focused teams
- Fault isolation
- Mixed technology stacks
- Granular scaling

Challenges of Microservices Architecture

Key Challenges

- Complexity
- Development and testing
- Lack of governance
- Network congestion and latency
- Data integrity
- Management
- Versioning
- Skillset

Best Practices for Microservices Architecture

Key Recommendations

- Model services around the business domain.
- Decentralize everything.
- Data storage should be private to the service that owns the data.
- Services communicate through well-designed APIs.
- Avoid coupling between services.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway or IdP.
- Keep domain knowledge out of the gateway.
- Services should have loose coupling and high functional cohesion.
- Isolate failures.

WELL-DESIGNED APIS

What is an API?

Definition

- An API is an interface or communication protocol between a client and a server, intended to simplify the building of client-side software.
- It is a "**contract**" between the client and the server, such that if the client makes a request in a specific format:
  - It will always get a response in a specific format, or
  - It will initiate a defined action.

What is a Web API?

Definition

A server-side web API is a programmatic interface consisting of:

- One or more publicly exposed endpoints to a defined request–response message system.
- Typically expressed in JSON or XML.
- Exposed via the web—most commonly through an HTTP-based web server.

What is OpenAPI?

Definition

The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for Web APIs. It enables:

- Humans to discover and understand the capabilities of a service.
- Computers to interact with the service in a predictable and standardized way.

Why OpenAPI?

Use Cases

Use cases for machine-readable API definition documents include, but are not limited to:

- Interactive documentation.
- Code generation for:
  - Documentation.
  - Clients.
  - Servers.
- Automation of test cases.

Application Programming Interface description language

```
{
  "swagger": "2.0",
  "info": {
    "description": "This is a sample server Petstore server.",
    "version": "1.0.0",
    "title": "Swagger Petstore",
    "termsOfService": "http://swagger.io/terms/",
    "contact": {
      "email": "apiteam@swagger.io"
    },
    "license": {
    "name": "Apache 2.0",
    "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
  }
},
```

Interactive documentation

https://petstore.swagger.io/

Open Source Linting Engine

Key Resources

- Repository: https://github.com/stoplightio/spectral
- ...

Mocker

Prism 3

- Open Source Mock Server.
- Complete rewrite in TypeScript.

https://github.com/stoplightio/prism

OpenAPI and API Gateway

API Gateway Definition

An API Gateway is a server that acts as an API front-end. It:

- Receives API requests.
- Enforces throttling and security policies.
- Passes requests to the back-end service.
- Passes the response back to the requester.

OpenAPI and API Gateway

Additional Features Often Supported by API Gateways

- Analytics
- Caching
- Authentication
- Authorization
- Security
- Audit
- Regulatory compliance
- Monetization: Functionality to support charging for access to commercial APIs.

OpenAPI Compatibility

Compatible with the Main Players

OpenAPI is compatible with major API management platforms, including:

- AWS API Gateway
- Apigee Edge
- Microsoft API Management
- `tyk.io` (open source)

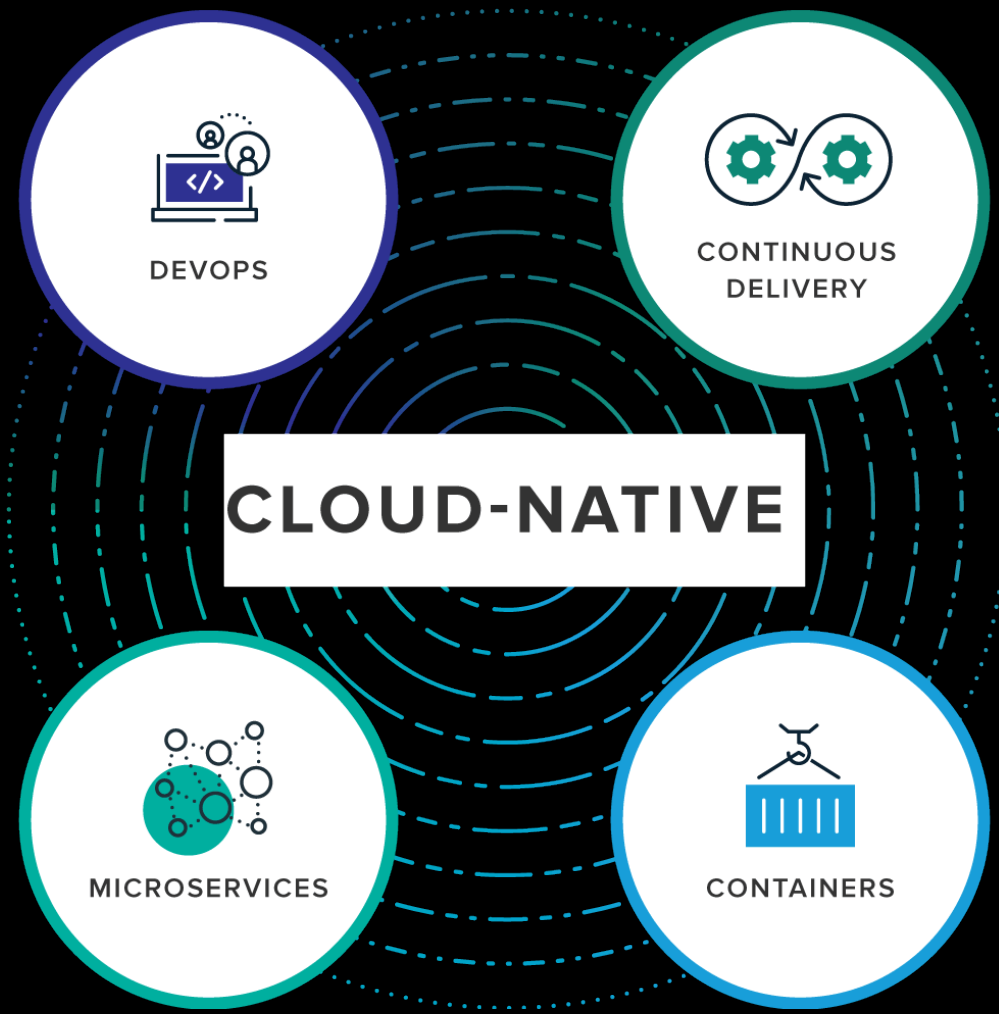
The Rules (to Make It Happen)

Use an API Gateway

- Put as much as you can on it (security, rate limiting, payload validation).
- You do not need microservices to use an API Gateway.
- Ensure that what is exposed matches what is declared in the OpenAPI specification file.
- Prefer a solution that is declarative.
- It will be easier to write automation scripts.
- Refrain from using custom extensions:
  - It will complicate things when these automations become the standard for all solutions.

CONTINUOUS DELIVERY

Continuous delivery



Outline

_

- What?
- Why?
- How?

What's Continuous Delivery?

Continuous Integration (CI)

The process of constantly merging development with a master branch for testing (e.g., on a daily basis).

Continuous Delivery (CD)

The practice of automatically deploying code to internal systems for further testing as soon as committed changes have passed automated tests.

Other Definition of Continuous Integration

Definition

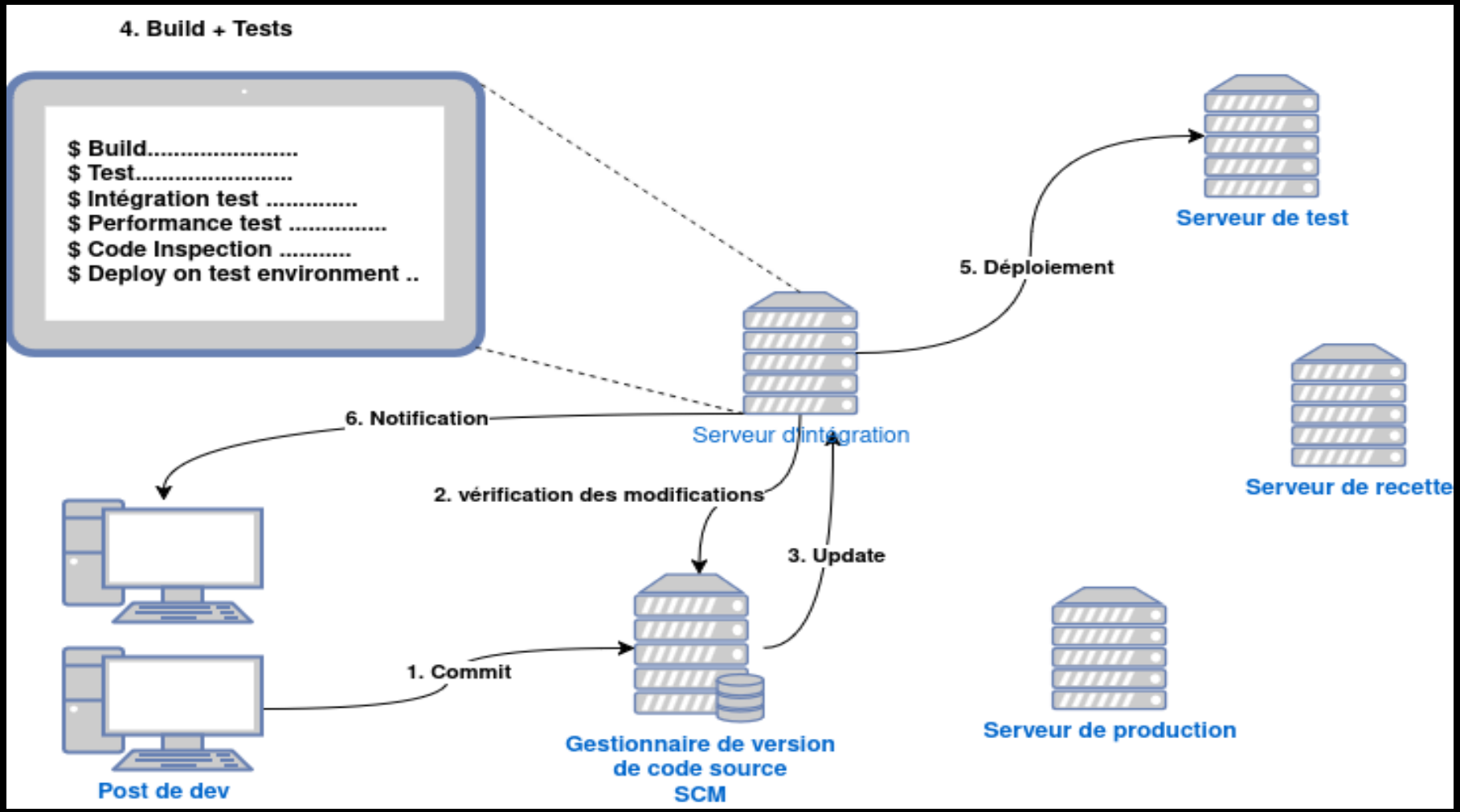
Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including tests) to detect integration errors as quickly as possible.

Reference

http://martinfowler.com/articles/continuousIntegration.html

What is Continuous Integration?

The practice of integrating source code continuously


4. Build + Tests
$ Build.....
$ Test.....
$ Intégration test .....
$ Performance test .....
$ Code Inspection .....
$ Deploy on test environment ..
Serveur de test
5. Déploiement
Serveur d'intégration
Serveur de recette
6. Notification
2. vérification des modifications
3. Update
1. Commit
Post de dev
Gestionnaire de version
de code source
SCM
Serveur de production

What is “Integration”?

At a Minimum

Integration involves the following steps:

- Gather the latest source together.
- Compile.
- Execute tests.
- Verify success.

What is “Integration”?

Additional Tasks

Integration can include other tasks such as:

- Rebuild the database.
- Build release distribution.
- Run code analysis and coverage tools.
- Generate documentation.

How Often is Continuously?

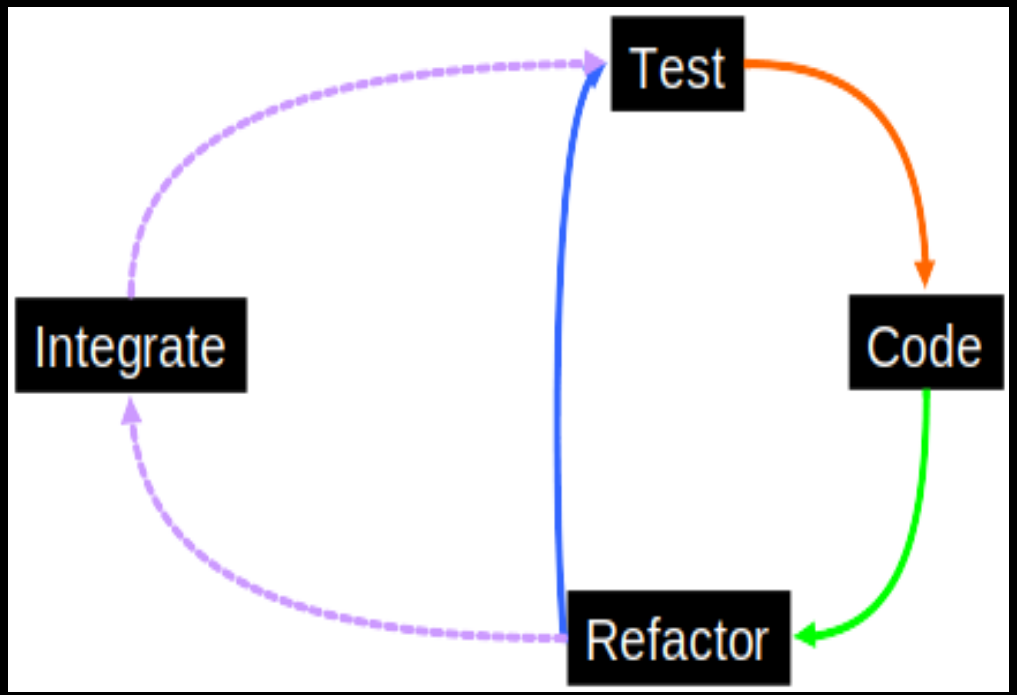
Guidelines

Continuously means:

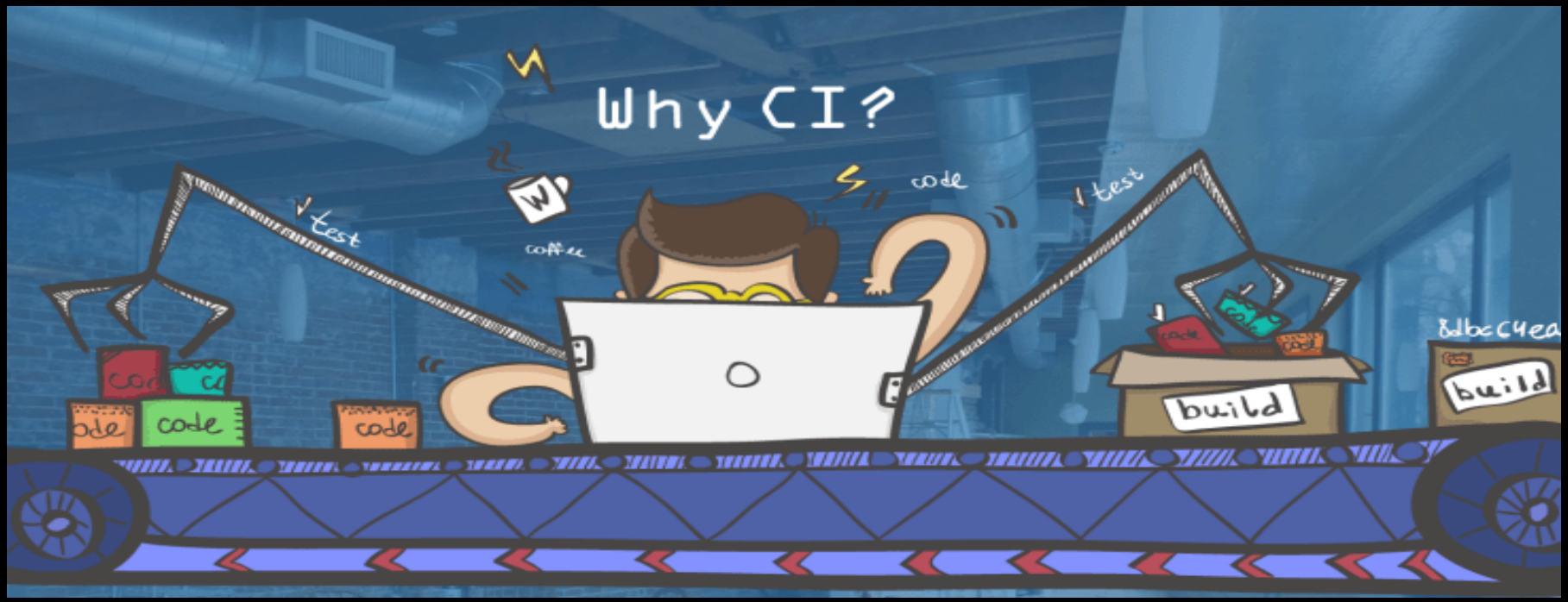
- As frequently as possible.
- More like once per hour than once per day.
- At a minimum, before leaving at the end of the day.

When to Integrate?

- Implement just enough, then integrate
- If using Test Driven Development, it forms a natural break in the cycle
- Taking small steps



WHY ?



WHY

Regular feedback

- For the integrator: “Did that work?”
- For the rest of the team: “Is the build OK?”
- Reduces Risk overall

Why?

Reduce integration pain

- No more ‘merge hell’
- XP Mantra: Do the ‘hard things’ often so they’re not hard any more

Why?

Increased automation

- Don't repeat yourself - automate to increase speed and to make less mistakes

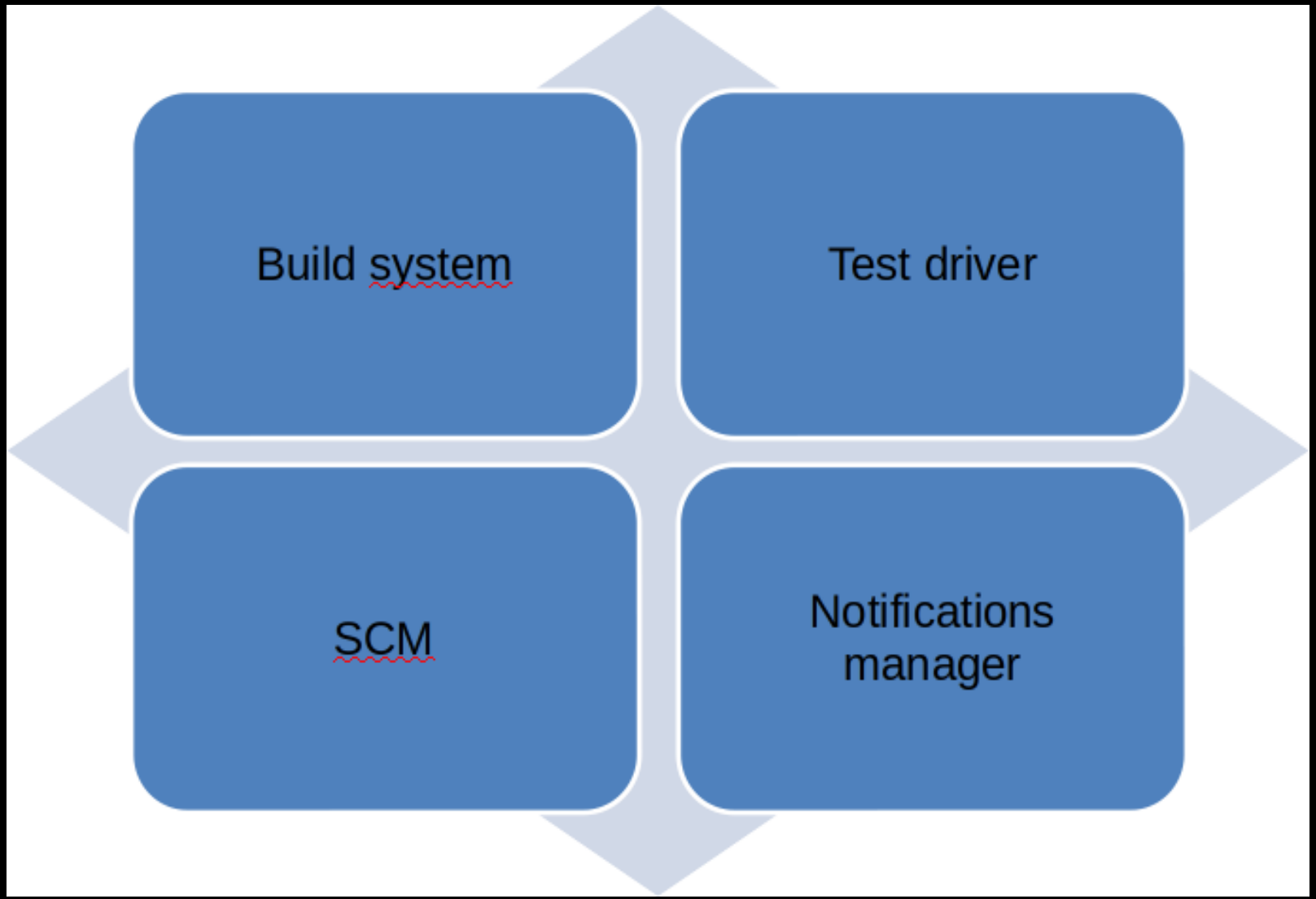
HOW ?

Technical pre-requisites

_

- Source Code checked into Source Control
- Automated (fast) build
  - Compile
  - Test
  - Command line without interaction
- Dedicated (communal) Integration Machine

Technical pre-requisites



Social Pre-requisites

- **Developer Discipline**
  - Continuous means continuous, not 'once per week'
- **Shared Ownership**

Automated CI

Automated Continuous Integration Server

- Detects changes in source control
- Launches integration build
- Publishes results

Why Use Automated CI?

Benefits

- Makes integration easy
- Guarantees integration happens
- Better feedback options
- Encourages test automation
  - Through metrics

Immediate Feedback is Key

Build Manager

A Build Tool + A Dependency Management Tool

- `maven`, `gradle`, `ivy`
- `npm`, `grunt`, `gulp`
- `composer`
- `...`

Objectives

Goals

- Make the development process visible or transparent
- Provide an easy way to see the health and status of a project
- Decrease training time for new developers
- Bring together the tools required in a uniform way
- Prevent inconsistent setups
- Provide a standard development infrastructure across projects
- Focus energy on writing applications

Objectives

Benefits

- Standardization
- Fast and easy to set up a powerful build process
- Dependency management (automatic downloads)
- Repository management
- Extensible architecture

SHOW ME!

Continuous deployment

The Basic Idea

- Commit frequently to reduce merge issues
- Automatically test each commit
- Automatically create a production candidate build for each successfully tested commit
- Deploy that build to necessary environments for manual testing
- Deploy builds that pass manual testing to production

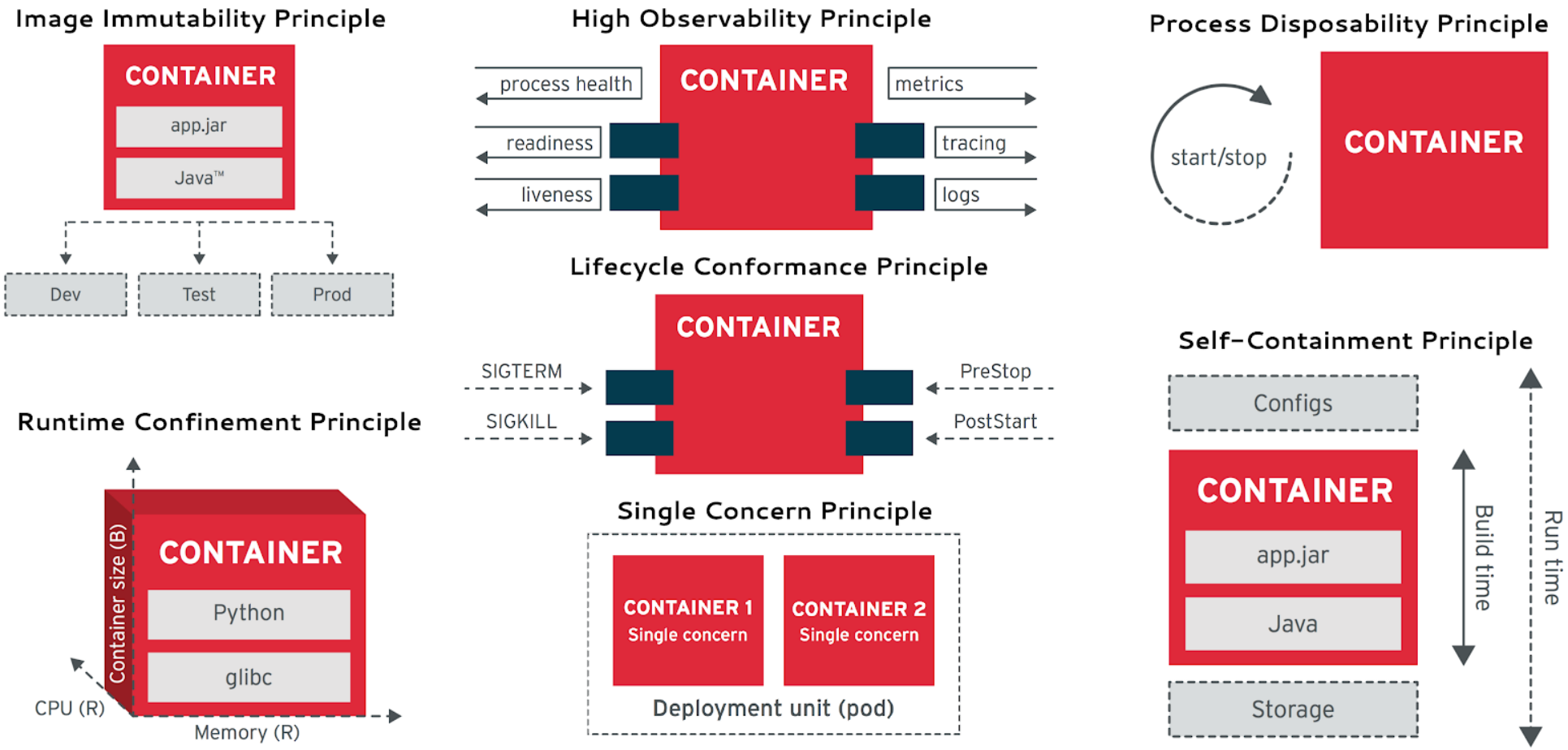
Continuous Deployment

The practice of automatically deploying code to production as soon as committed changes have passed automated tests

- Basically removing manual testing from the picture
- Automated tests have to be fantastic
- Not for every situation, especially as bigger organizations usually need approvals in order to deploy

CLOUD NATIVE

Cloud Native Container Design Principles



Build Time

Principles

- **Single Concern**: Each container addresses a single concern and does it well.
- **Self-Containment**: A container relies only on the presence of the Linux kernel. Additional libraries are added when the container is built.
- **Image Immutability**: Containerized applications are meant to be immutable, and once built are not expected to change between different environments.

Runtime

Principles

- **High Observability**: Every container must implement all necessary APIs to help the platform observe and manage the application in the best way possible
- **Lifecycle Conformance**: A container must have a way to read events coming from the platform and conform by reacting to those events
- **Process Disposability**: Containerized applications must be as ephemeral as possible and ready to be replaced by another container instance at any point in time
- **Runtime Confinement**: Every container must declare its resource requirements and restrict resource use to the requirements indicated

Monitoring tools

https://www.monperrus.net/martin/monitoring.pdf

Further Material

- Software Engineering Radio
- KTH DevOps Course
- Gene Kim, Patrick Debois, John Willis, and Jez Humble. 2016. The Devops Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press.
- Cloud Native Container Design Principles