UMD DATA605 - Big Data Systems

7.4: Big Data Architectures

Instructor: Dr. GP Saggese - gsaggese@umd.edu

Software testing

- Evaluate functionality, reliability, performance, and security of a product to ensure it meets requirements
  - Software testing is critical in development
- Adage:
  - "If it's not tested, it doesn't work"
  - "Debugging is 2x harder than writing code"
    - Corollary: If you do your best to write code, how can you debug it?
- **Different types of testing**
  - *Unit testing*: Test individual components to ensure each part functions correctly in isolation
  - *Integration testing*: Ensure components work together as expected (e.g., detect interface defects)
  - *System testing*: Evaluate a fully integrated system's compliance with specified requirements

Software testing

- **Smoke/sanity testing**: Quick check of functionalities to ensure main functions work
  - E.g., decide if a new build is stable
  - E.g., application doesn't crash on launch
- **Regression testing**: Ensure new changes don't affect existing functionality
- **Acceptance testing**: Final testing phase before release
  - More common in waterfall than Agile
- **Performance testing**: Load, stress, and spike testing
- **Security testing**: Identify vulnerabilities, threats, and risks
- **Usability testing**: Assess ease of use for end-users
  - E.g., ui/UX
- **Compatibility testing**: Check compatibility with browsers, database versions, OS, mobile devices

CI / CD

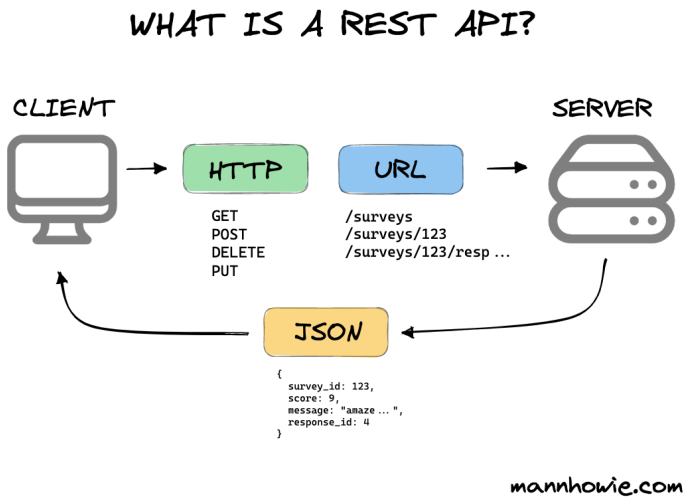
- **Continuous integration (CI)**
  - Merge code changes into a central repository multiple times a day
  - Automate build and test after each change
  - **Goal**: Detect and fix integration errors quickly
  - Add code with unit tests
- **Continuous deployment (CD)**
  - Automatically deploy code changes to production
    - Without human intervention
    - After build and test phases pass
  - **Goal**: Deliver features, bug fixes, and updates continuously
  - E.g., GitHub actions, GitLab workflows, AWS Code, Jenkins

RESTful API

- REST API
  - Web service API conforming to REST style
  - REST = REpresentational State Transfer
  - Style for distributed systems
- **Uniform interface**
  - Refer to resources (e.g., document, services, URI, persons)
  - Use HTTP methods (`GET`, `POST`, `PUT`, `DELETE`)
  - Naming convention, link format
  - Response (XML or JSON)
- **Stateless**
  - Each request contains all necessary information
  - No shared state
  - Inspired by HTTP (modulo cookies)

RESTful API

- **Cacheable**
  - Label response data as cacheable or non-cacheable
  - Reuse cacheable responses
  - Increase scalability and performance
- **Layered system**
  - Each layer interfaces only with the immediate layer
  - E.g., in a tier application



Stages of deployment

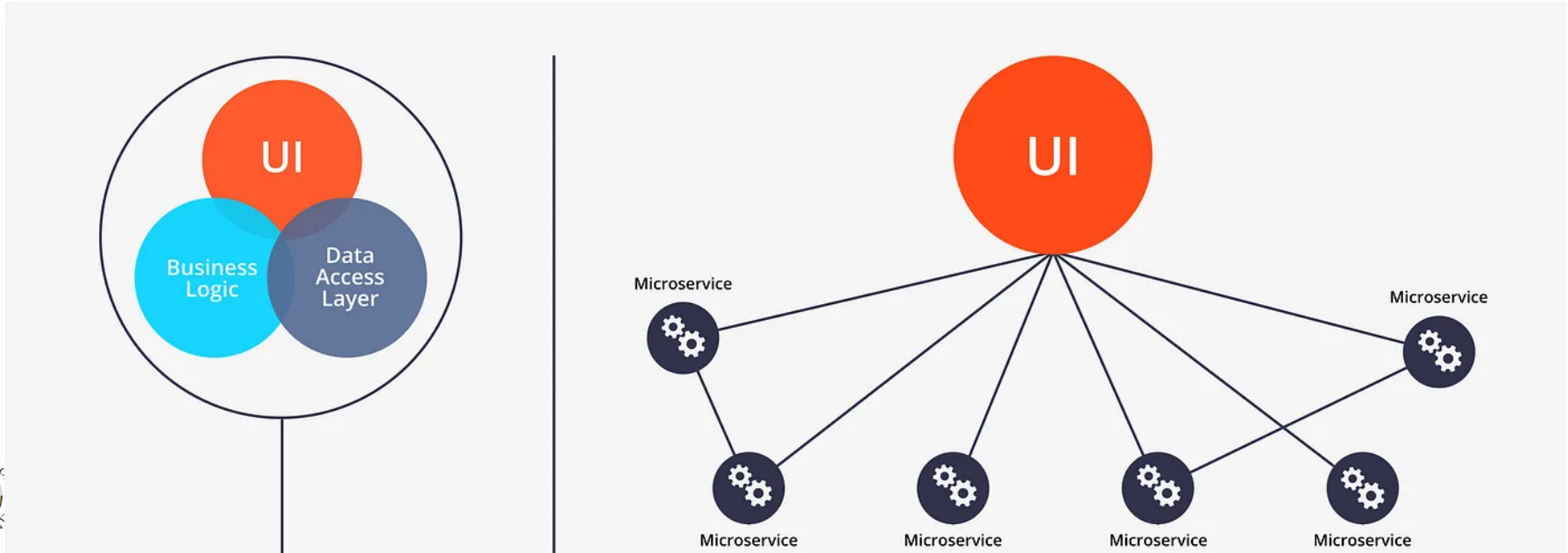
- The deployment of software progresses through several environments
  - Each environment tests, validates, and prepares software for release to end user
- **Development environment (Dev)**
  - Individual for each developer or feature team
  - Goal: Developers write and initially test code
- **Testing or Quality Assurance (QA) environment**
  - Mirrors production environment to perform under similar conditions
  - Goal: Systematic testing to uncover defects and ensure quality
- **Staging/Pre-Prod environment**
  - Final testing phase before deployment to production
  - Replica of production environment for final checks and stakeholder review
- **Production Environment (Prod)**
  - Live environment where software is available to end users
  - Optimized for security, performance, and scalability
  - Focus on uptime, user experience, and data integrity

Semantic versioning

- **Semantic versioning** is a versioning scheme for software that conveys meaning about underlying changes
  - Systematic approach
  - Understand potential impact of updating to a new version
- Major Version `X.y.z`
  - Increment for incompatible API changes or significant updates that may break backward compatibility
- Minor Version `x.Y.z`
  - Increment for backward-compatible enhancements and significant new features that don't break existing functionalities
- Patch Version `x.y.Z`
  - Increment for backward-compatible bug fixes that address incorrect behavior
- Pre-release Version:
  - Label to denote a pre-release version that might not be stable
    - E.g., `1.0.0-alpha`, `1.0.0-beta`
  - Releases for testing and feedback, not for production use
- Build Metadata
  - Optional metadata to denote build information or environment specifics
  - E.g., `1.0.0+20210313120000` or `1.0.0+f8a34b3228c`

Microservices vs Monolithic Architecture

::: columns :::: {.column width=40%} - Different styles of building complex systems - Find the right granularity :::: :::: {.column width=60%}



Microservice Architecture

- **Modularity**: small, independently deployable services, each with specific business functionality
- **Scalability**: scale services independently for efficient resource use based on demand
- **Technology diversity**: develop each service with the best technology stack for its functionality
- **Deployment flexibility**: supports continuous delivery and deployment for faster updates
- **Resilience**: isolate and address faults; one service failure doesn't affect the entire system
- **Cons**
  - Complex deployment
  - Requires tooling

Monolithic Architecture

- **Simplicity**: Simpler to develop, test, deploy, and scale as a single unit
- **Tightly coupled components**: Components run in the same process, leading to scalability and resilience issues
- **Technology stack uniformity**: Developed with a single technology stack, limiting flexibility
- **Deployment complexity**: Updates require redeploying the entire application
- **Single point of failure**: Issues in any module can affect the entire application