Advanced Professional Software Engineering 2024-25

Lecture 3 – Software Architecture

S Waqar Nabi

(with thanks to Marco Tulio Valente, Universidade Federal de Minas Gerais)

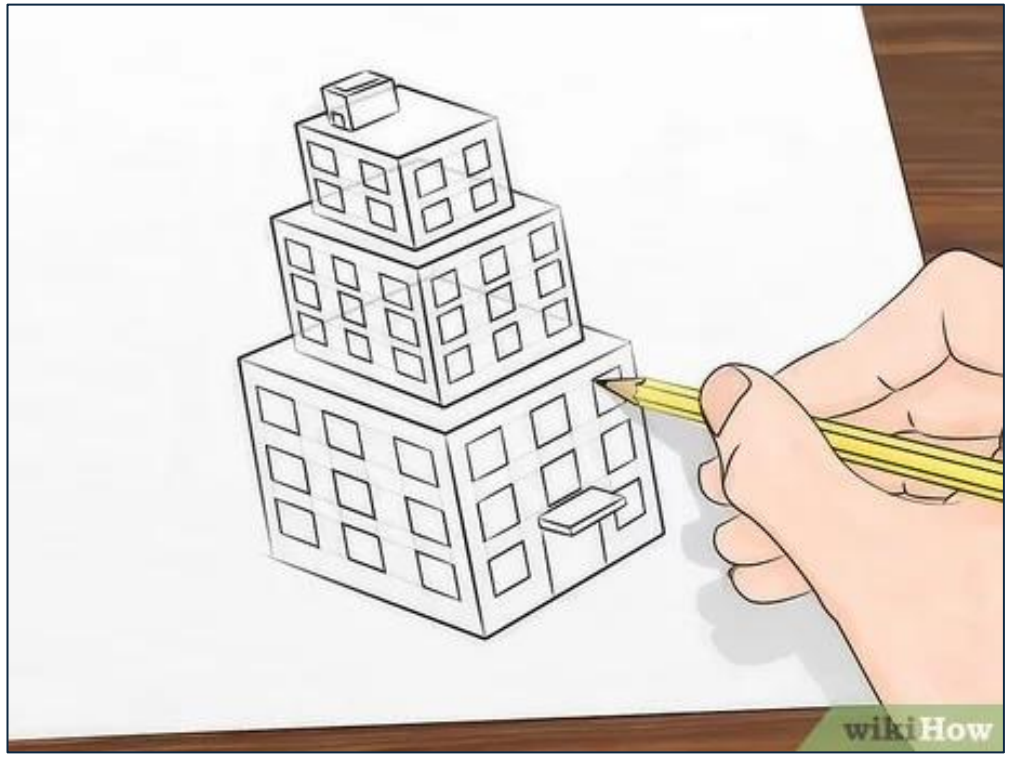
Mar 2025

Architecture is about the important stuff.

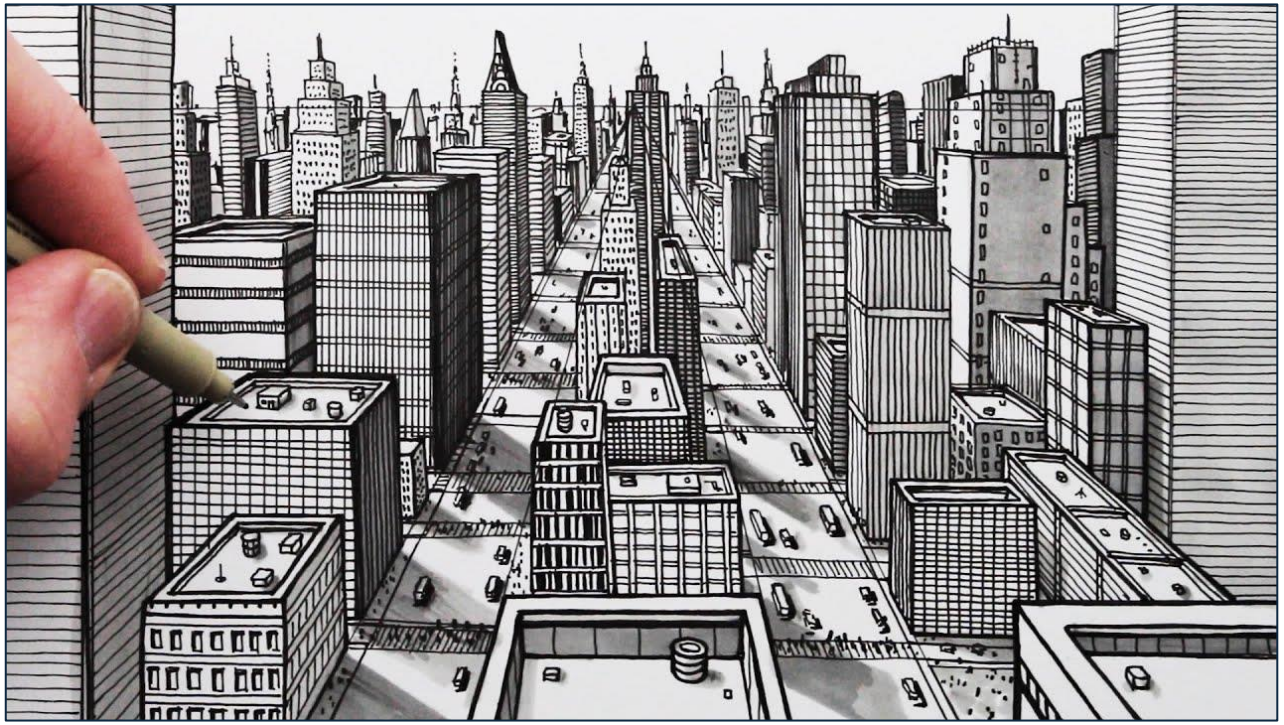
Whatever that is. – Ralph Johnson

Architecture = high-level design

- The focus shifts from small units (e.g., classes)
- Focusing instead on larger and more relevant units
- Such as packages, modules, subsystems, layers, services, etc



Software Design



Software Architecture

Architectural Patterns = predefined architectures

- Layered
- Model-View-Controller (MVC)
- Microservices
- Message-Oriented
- Publish/Subscribe
- Cloud Native and Serverless

Linus-Tanenbaum Debate (1992)

Linux

Favoured monolithic architecture

Minix

Critizied Linux's monolithic architecture, advocated for microservices architecture

Fast forward 17 years later (2009) to see Torvalds' statement at a Linux conference

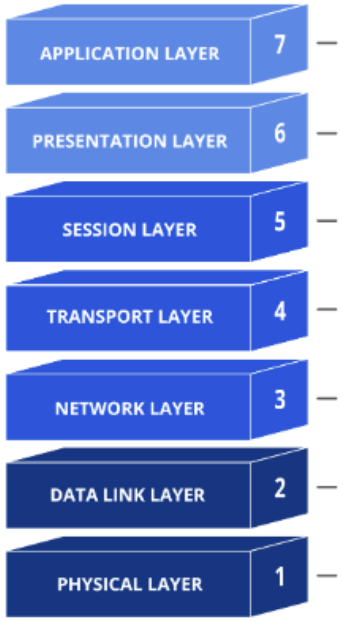
"We are definitely not the streamlined, small, hyper-efficient kernel that I envisioned 15 years ago. The kernel is huge and bloated... And whenever we add a new feature, it only gets worse."

Key takeaway: the costs of architectural decisions can take years to become apparent...

Layered Architecture

Layered Architecture

- A system is organized in a hierarchical way
- Layer *n* can only use services from layer *n-1*
- Widely used in networks and distributed systems



Advantages: divide and conquer

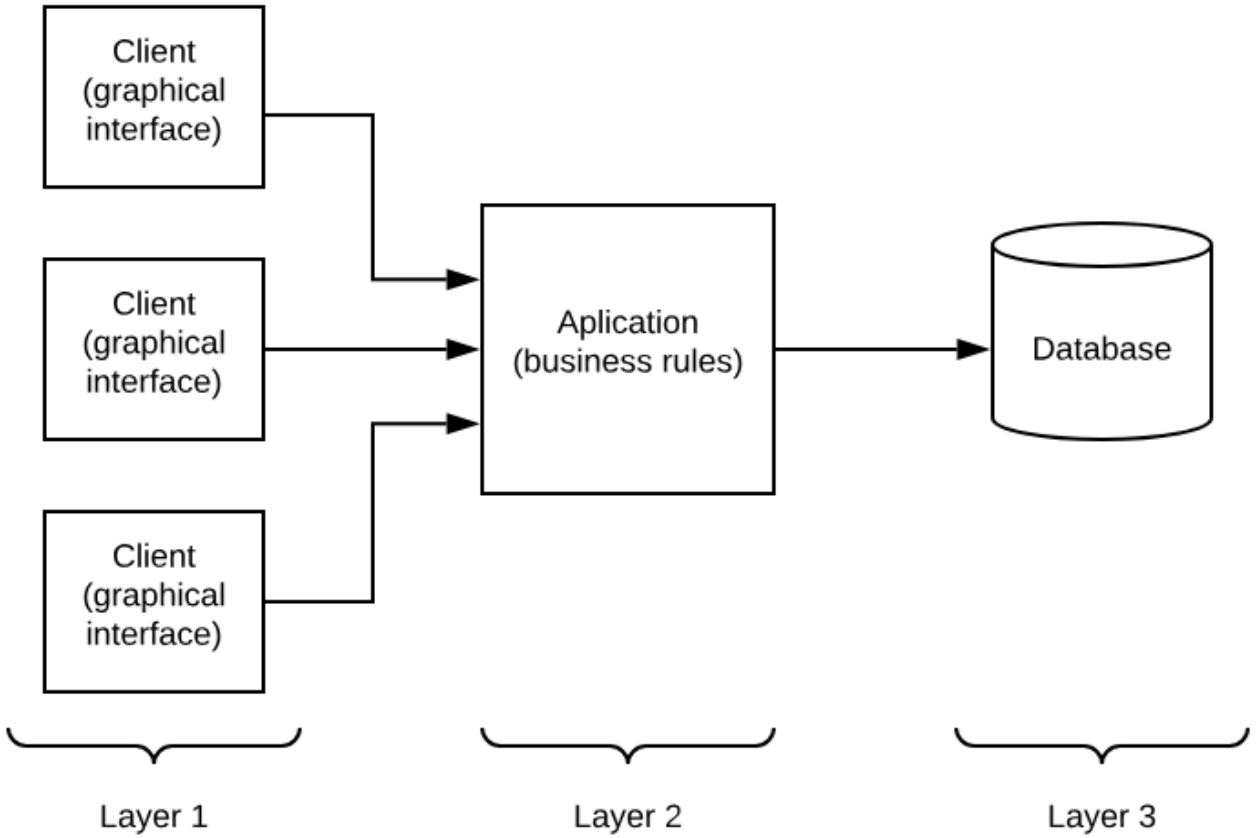
- Breaks down system complexity and facilitates:
  - Understanding of the system
  - Layer replacement (e.g., TCP to UDP)
  - Layer reuse (e.g., multiple apps use TCP)

Variations

- Three-Tier Architecture
- Two-Tier Architecture

Three-Tier Architecture

Commonly used for Enterprise Applications (payroll, inventory, accounting, etc)



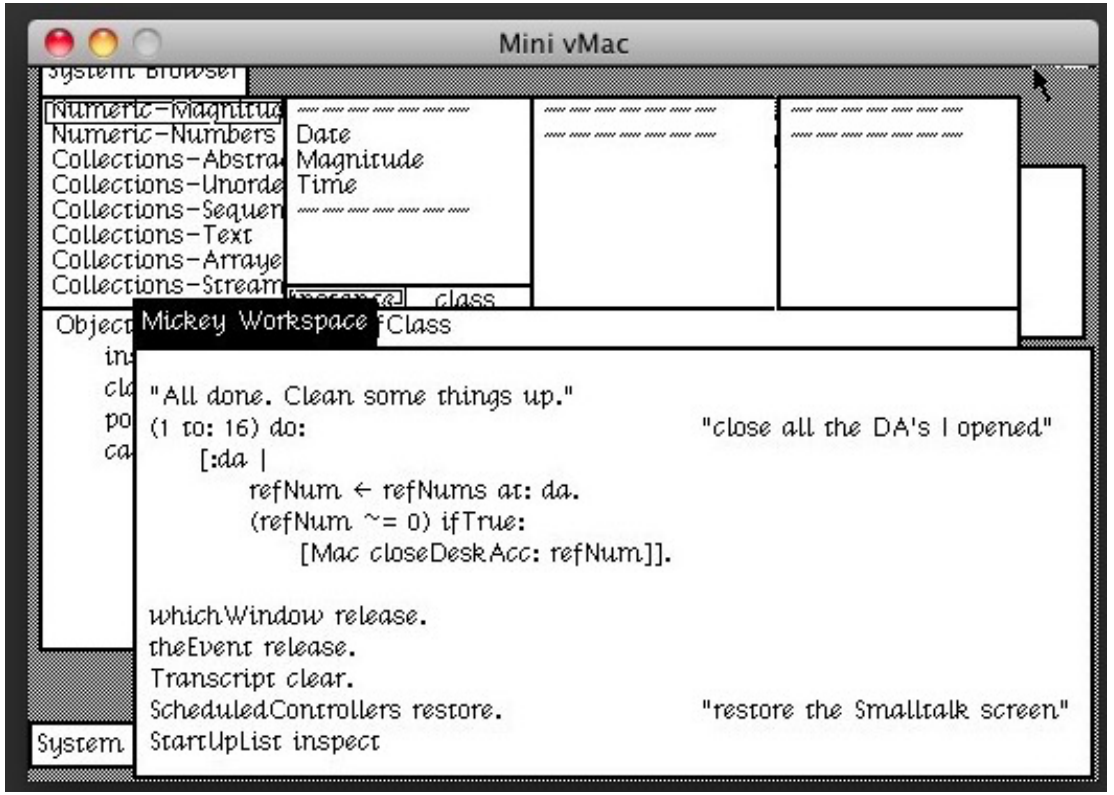
Two-Tier Architecture

- Advantages of being simpler:
  - Tier 1: client (user interface + business logic)
  - Tier 2: database server
- Disadvantage: processing primarily occurs on the client

Model-View-Controller (MVC)

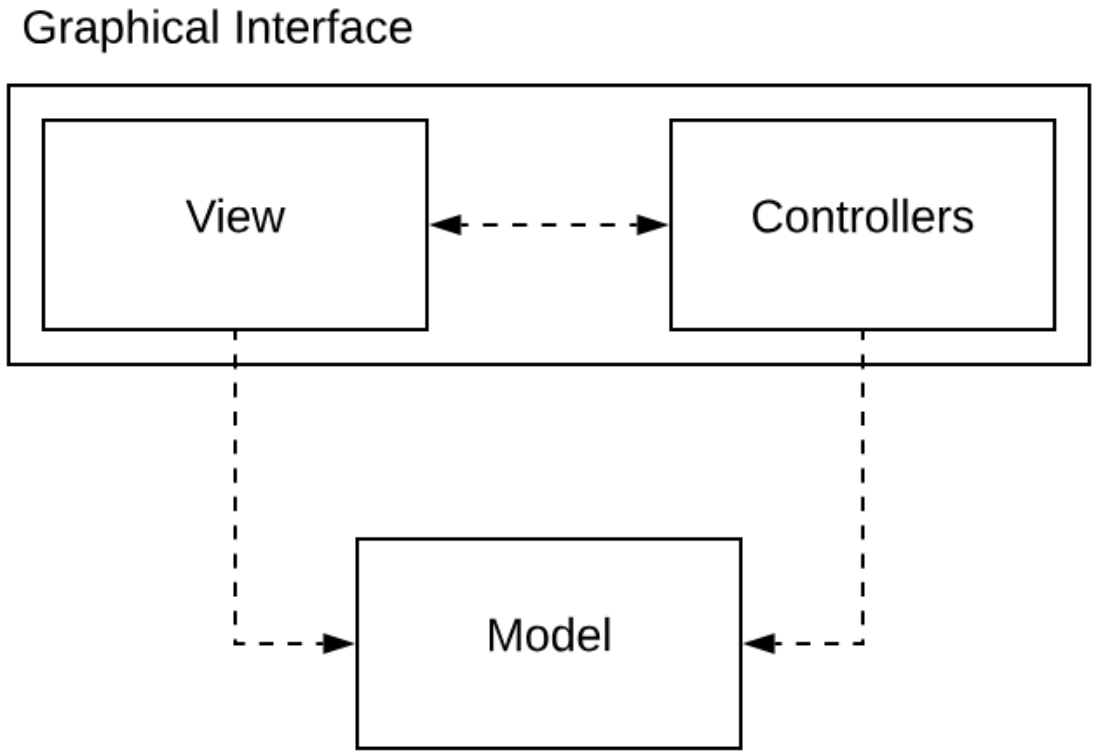
MVC Architecture

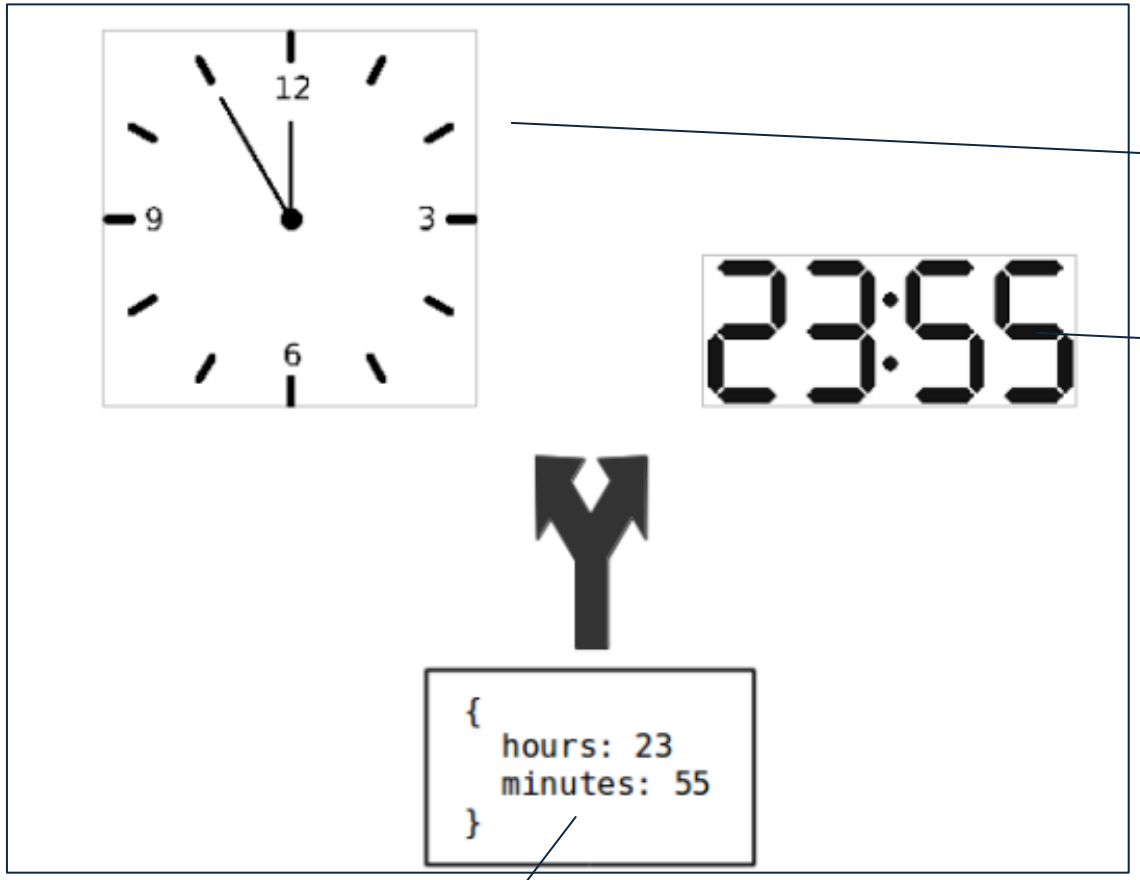
- First introduced in the 1980s through Smalltalk
- Designed to implement Graphical User Interfaces (GUIs)
- Strongly connected to Object Oriented Design



MVC divides classes into 3 groups

- View: classes for implementing GUIs, including windows, buttons, menus, scroll bars, etc.
- Controller: classes that handle events produced by input devices such as mouse and keyboard
- Model: classes containing application logic and data





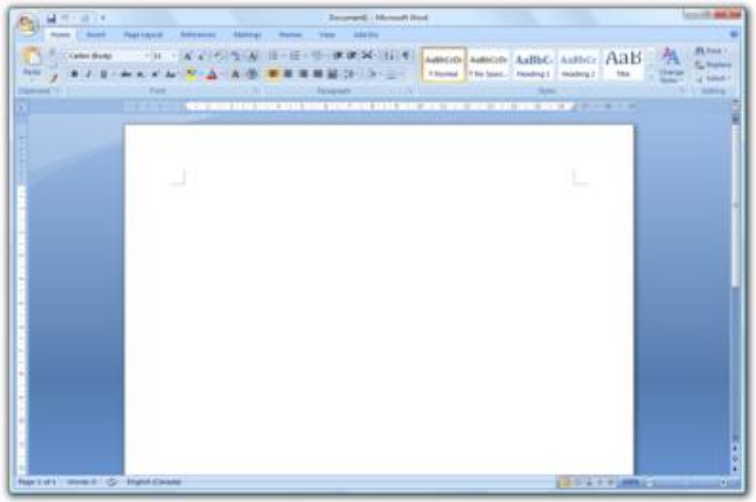
GUI #1

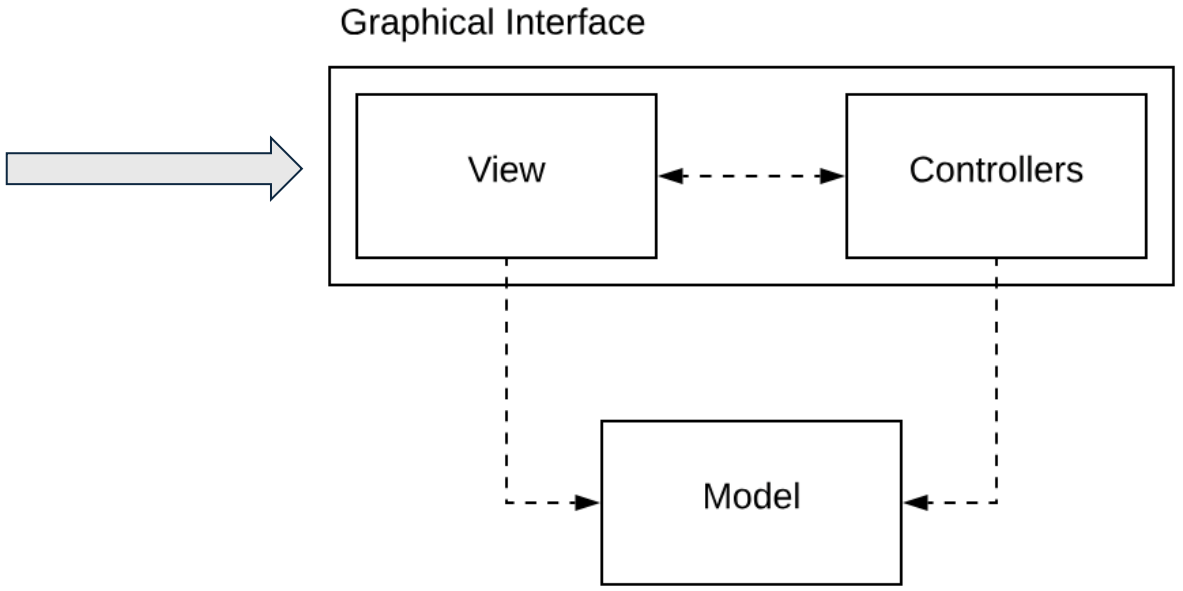
GUI #2

Model

Traditional MVC apps

- MVC was originally designed for desktop applications
- Examples: Microsoft Word, Google Chrome, etc.





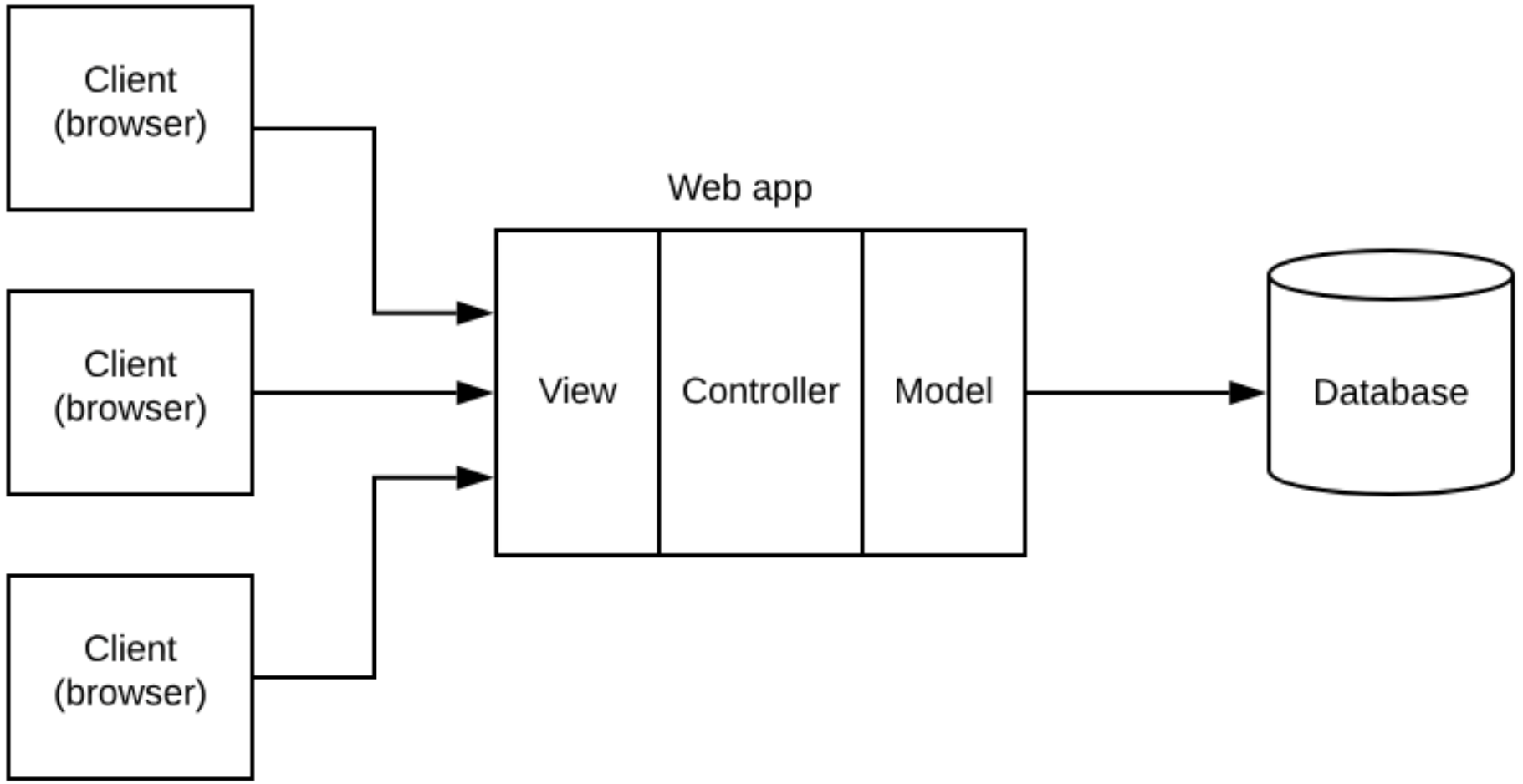
MVC Today

- MVC Web
- Single Page Applications

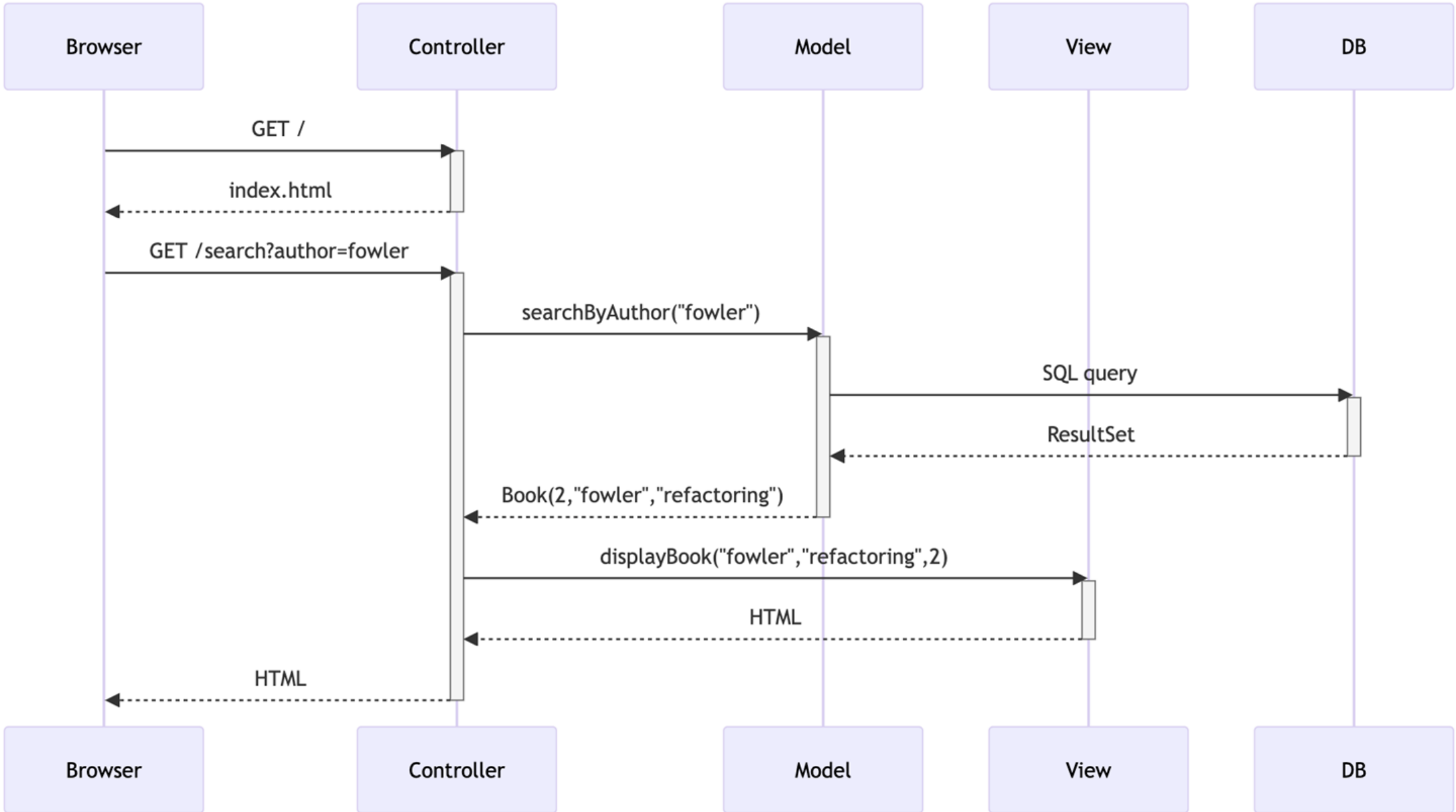
MVC Web

MVC Web

- MVC was adapted for the Web
- More closely resemebles the 3-tier architecture
- Popular frameworks include Ruby on Rails, Django, Spring, PHP Laravel, etc.

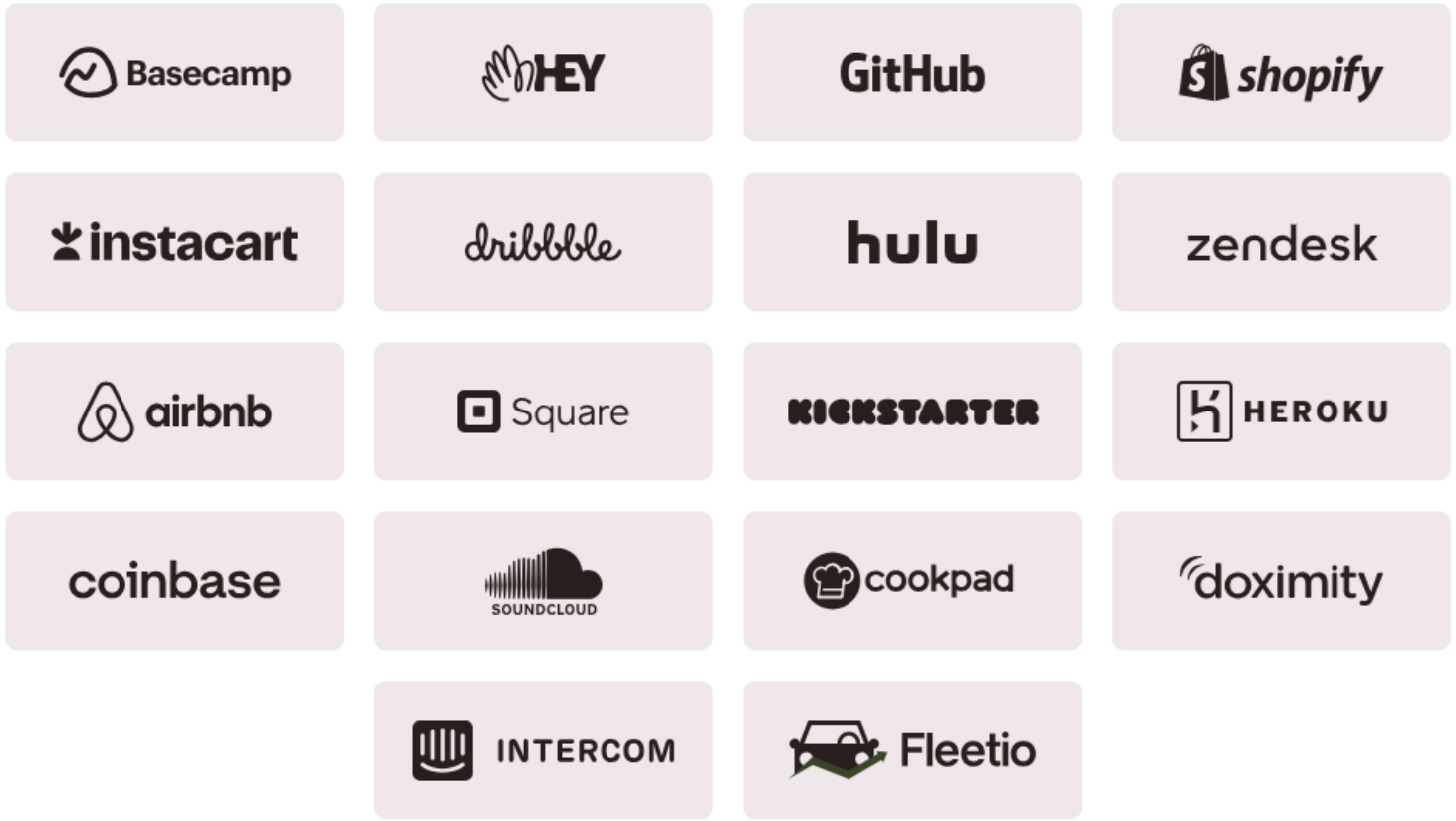


MVC Web: Has elements of both MVC and 3-tier architecture


Browser
Controller
Model
View
DB
GET /
index.html
GET /search?author=fowler
searchByAuthor("fowler")
SQL query
ResultSet
Book(2,"fowler","refactoring")
displayBook("fowler","refactoring",2)
HTML
HTML
Browser
Controller
Model
View
DB

MVC Frameworks remain relevant

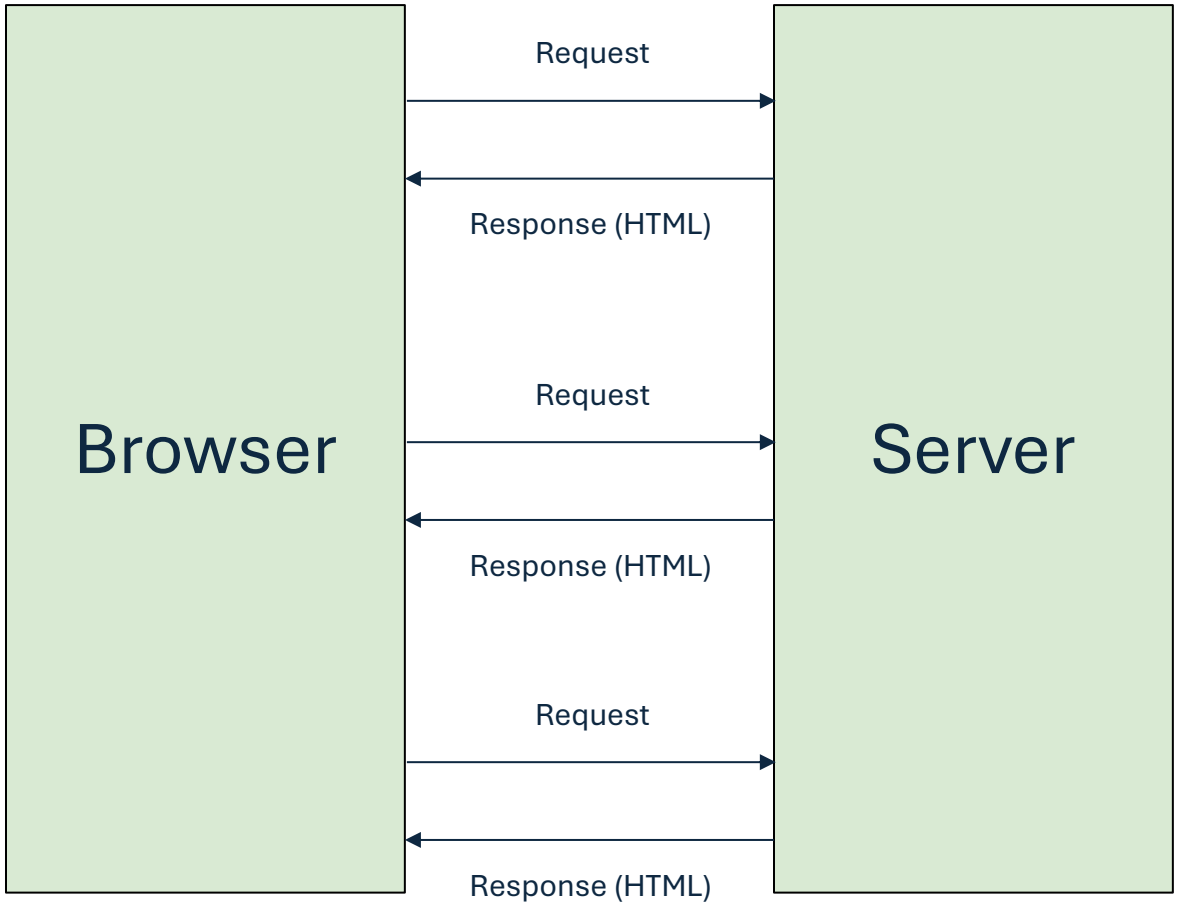
Over the past two decades, Rails has taken countless companies to millions of users and billions in market valuations.



https://rubyonrails.org (April 2024)

Single Page Applications (SPAs)

Traditional Web Apps



Problem: less responsive interfaces

Multiple Page Applications

Single Page Applications

- Run in the browser, but are more independent of the server
  - Manipulate its own interface
  - Store and manage local data
  - Access the server only to fetch more data
- Example: GMail, Google Docs, Facebook, Figma, etc
- Implemented using JavaScript frameworks (React, Vue, Svelte, etc)

MVC: Summary

Traditional MVC (Smalltalk): desktop apps, pre-Web

MVC Web: MVC adaptation for the Web (fullstack)

RAILS

django

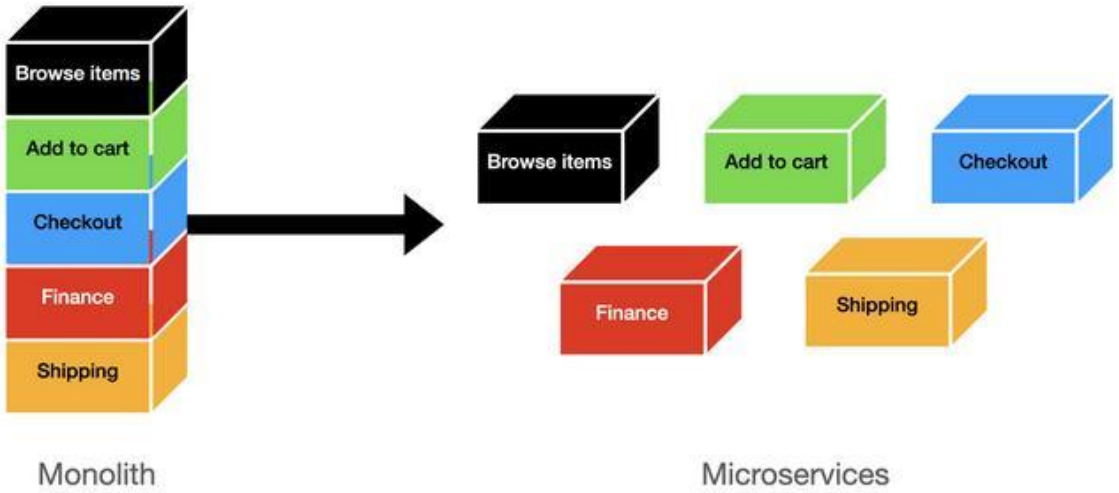
Laravel

SPA: MVC adaptation for responsive apps (frontend)

React

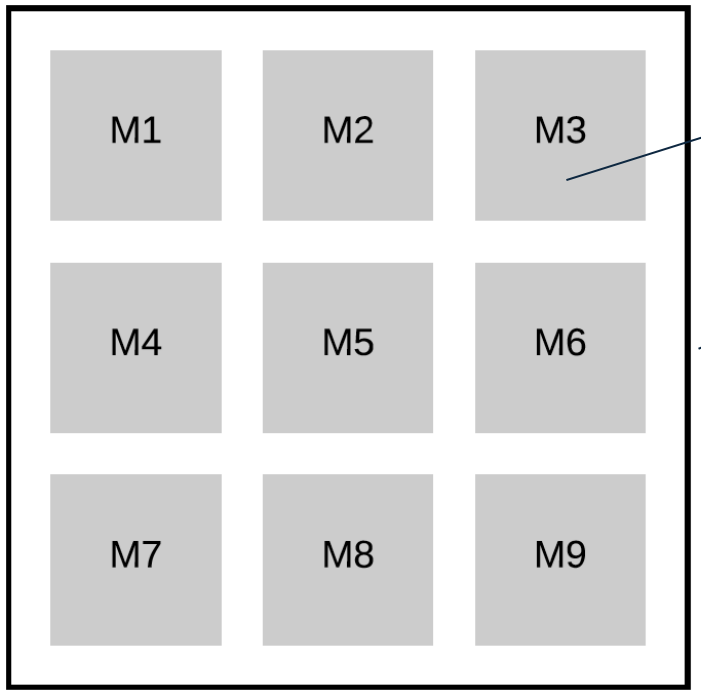
Vue.js

Microservices



Monoliths

- Monoliths: system exists as a single process at run-time
- Process: operating system process

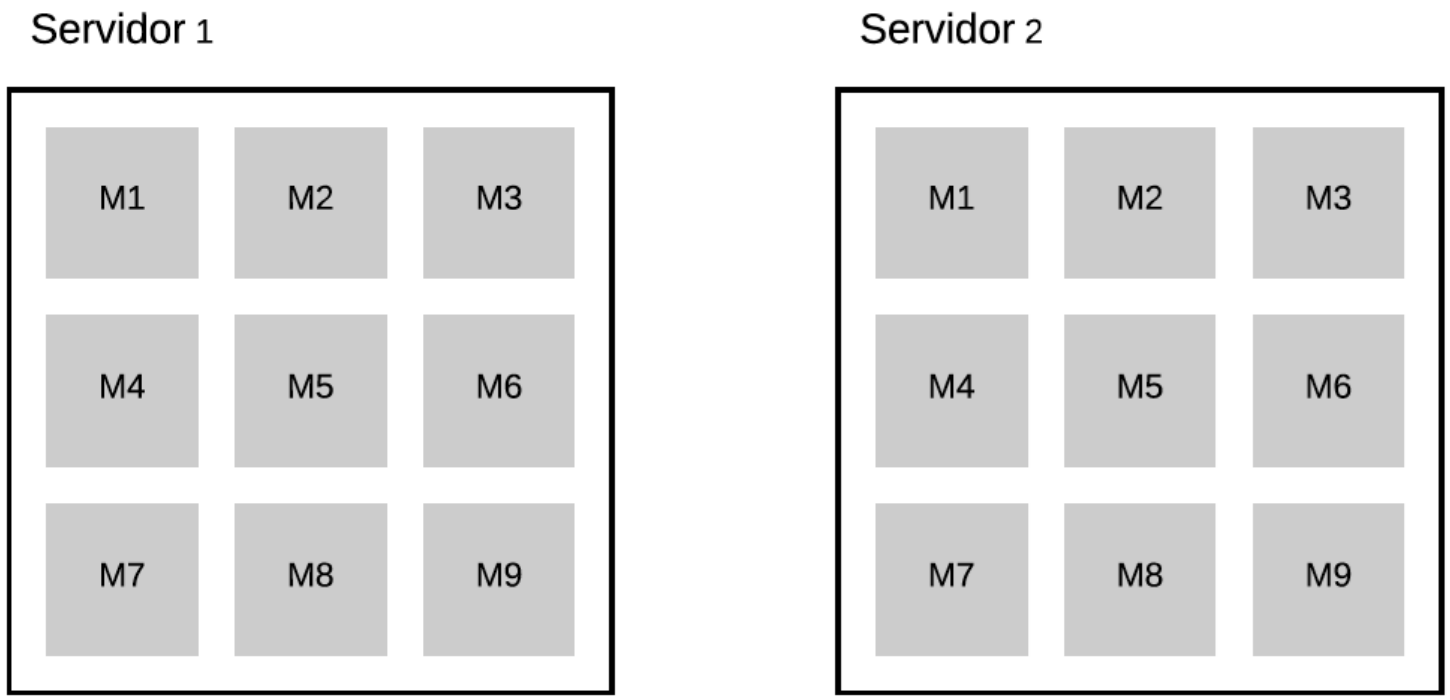


Modules: compilation units

At runtime, a monolithic application executes in a single process

Problem #1 with Monoliths: Scalability

- Scalability requires scaling the entire monolith
- This is inefficient when the bottleneck is in a single module



Problem #2 with Monoliths: Releases are slower

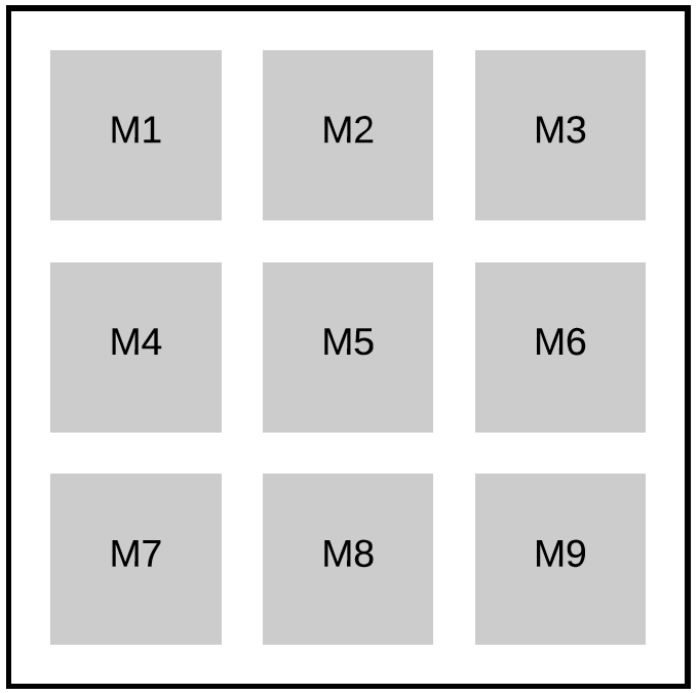
- The release process is slow, centralized, and bureaucratic
- Teams don't have autonomy to put modules into production
- Reason: changes can impact other teams' modules
- As a result:
  - Releases must follow predefined dates
  - Releases require several tests, sometimes manual, to ensure correctness

Especially true in a monolithic codebase

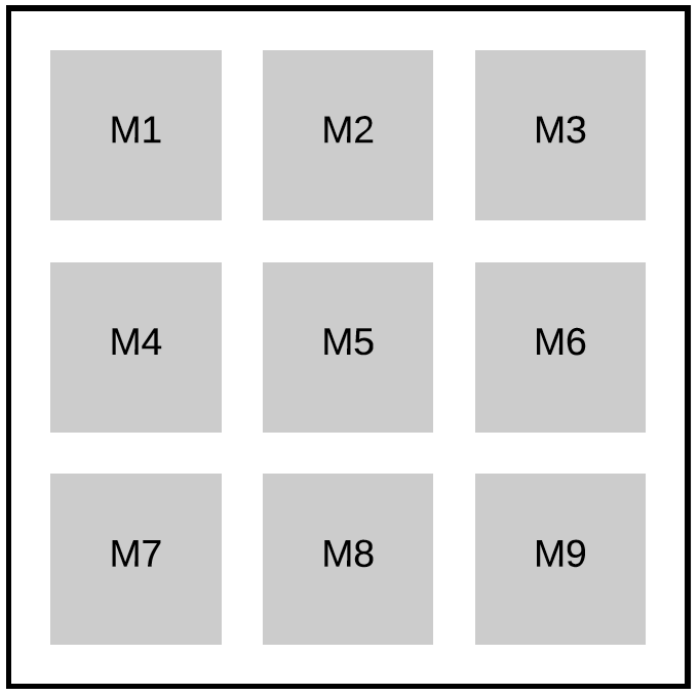
Microservices

- Services ⇒ Each module runs as an independent *process*
    - *Recall: Each independent process is managed separately by the OS, it has its own address space.*
- Micro ⇒ small modules

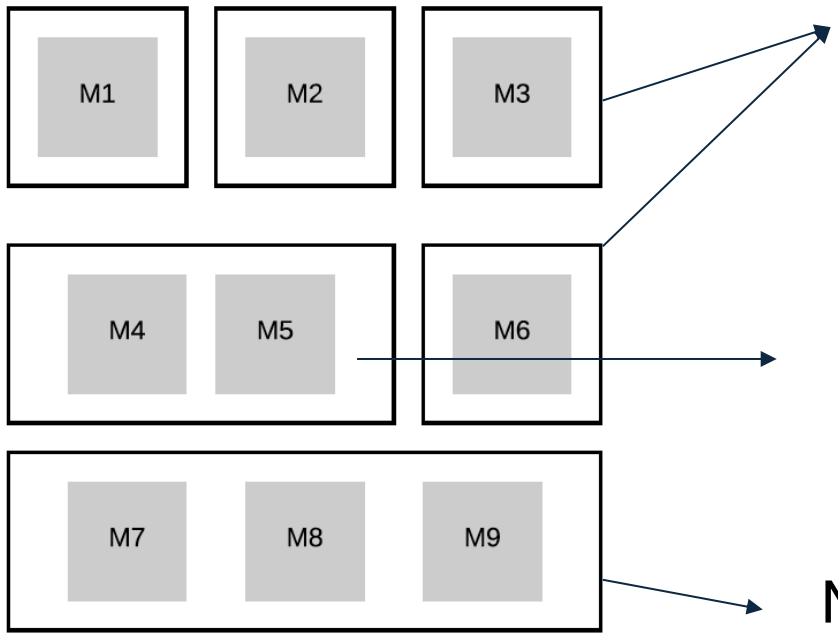
Monolithic Architecture



Monolithic Architecture



Microservices-based Architecture



Microservice with 1 module

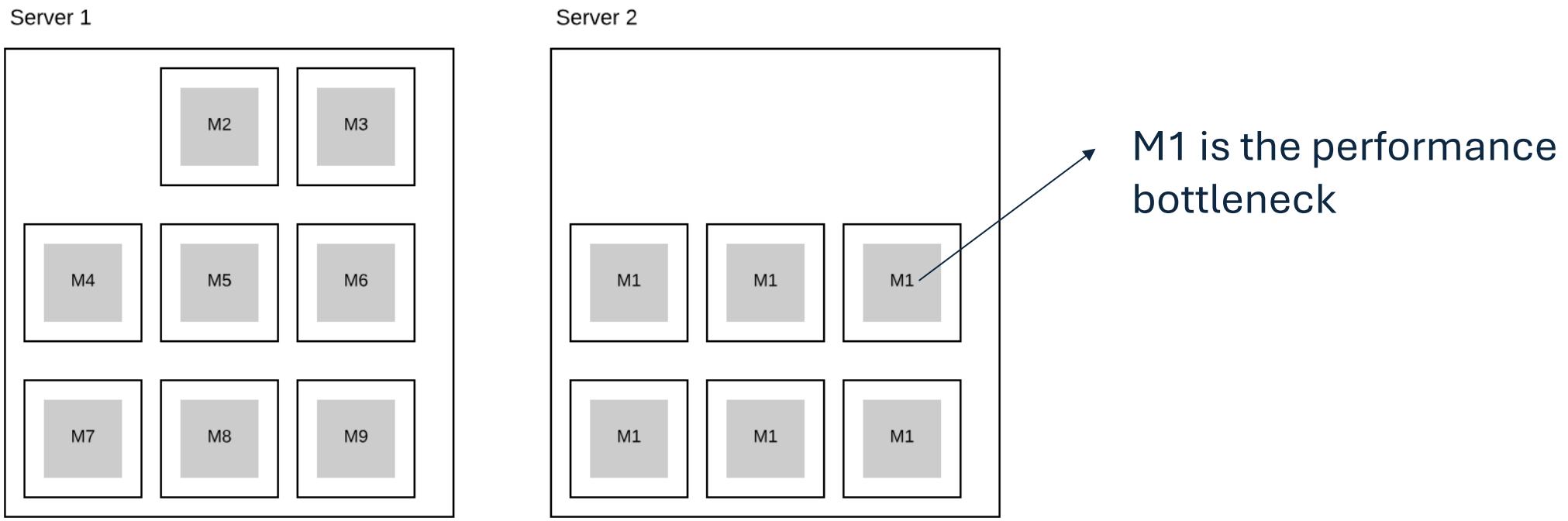
Microservice with 2 modules

Microservice with 3 modules

microservice = process (run-time, operating system)

Advantage #1: Scalability

- Each module can be scaled independently



Advantage #2: Flexibility for Releases

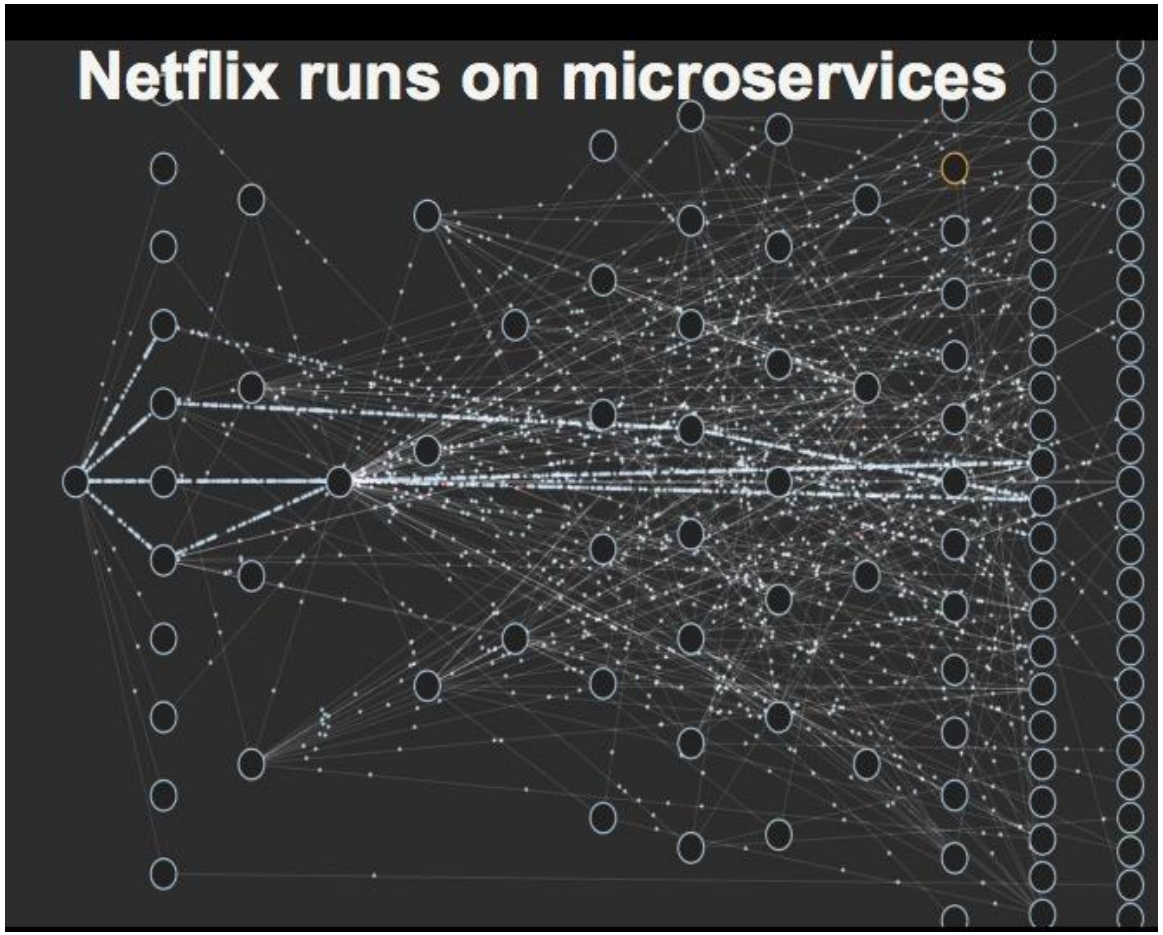
- The risk of interference between processes is smaller
- This is because each process has its own address space
- As a result, teams have autonomy to put microservices into production

Other Benefits of Microservices

- Microservices can use different technologies
- Partial failures (e.g., only one microservice may be offline)

Who uses microservices?

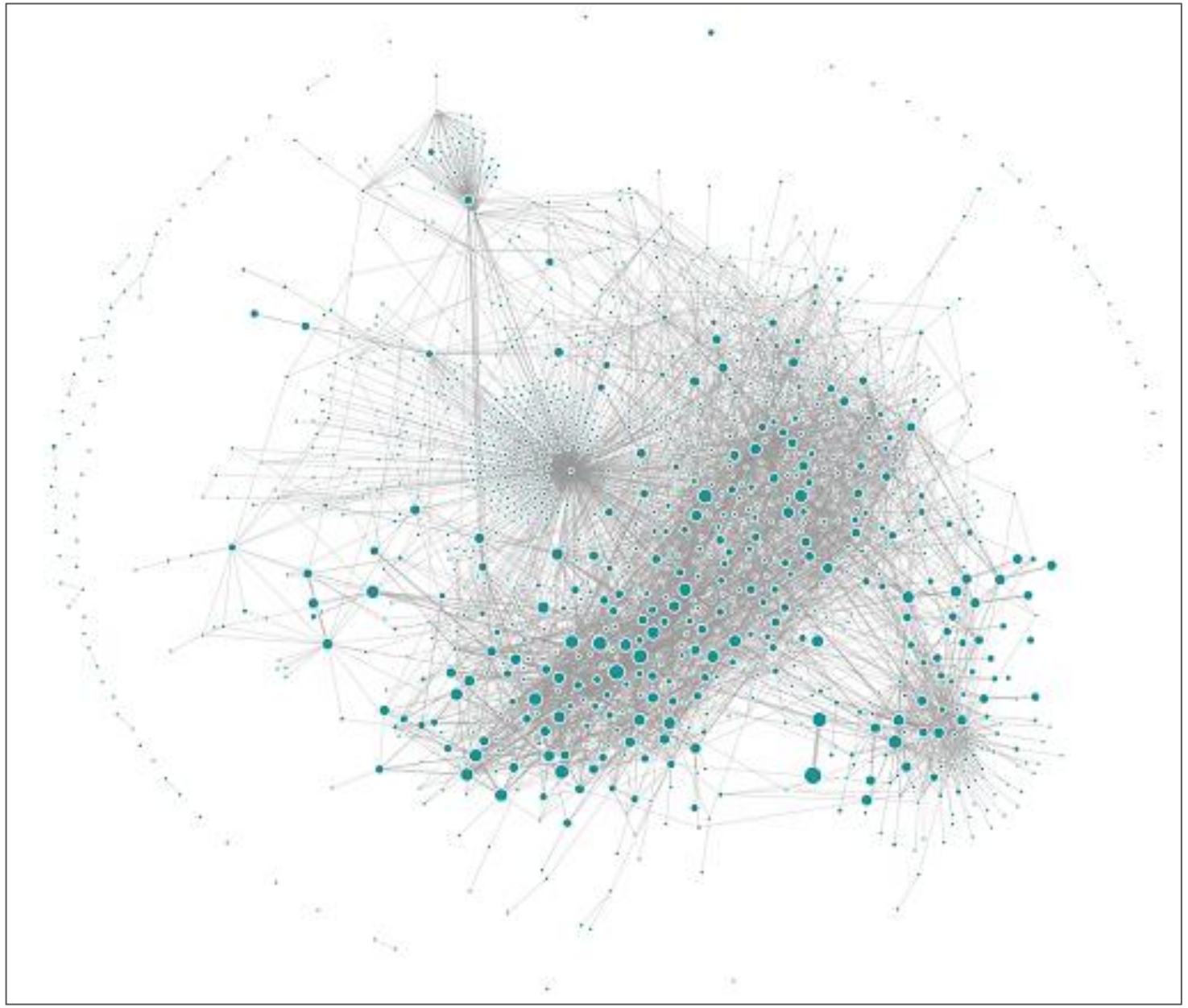
- Large companies including Netflix, Amazon, Google, etc



Each node is a microservice

Example:

Uber (~2018)



https://eng.uber.com/microservice-architecture/

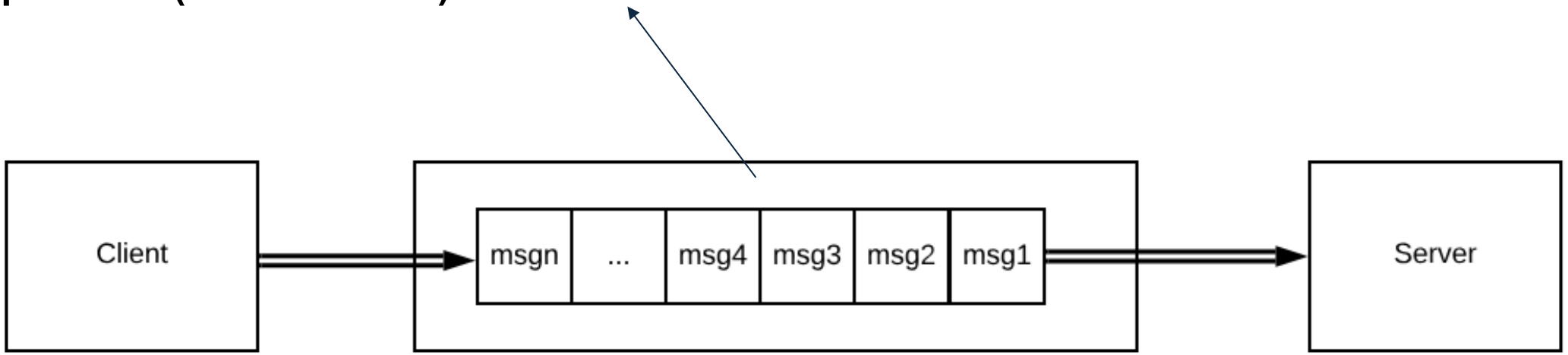
Microservices introduce significant complexity

- Managing hundreds of processes
- Increased network latency
- Complex data consistency (distributed transactions)

Message-Oriented Architecture

Understanding Message-Oriented Architecture

- Used in distributed applications
- Clients communicate with servers indirectly
- Communication occurs through an intermediary: a message queue (or broker)

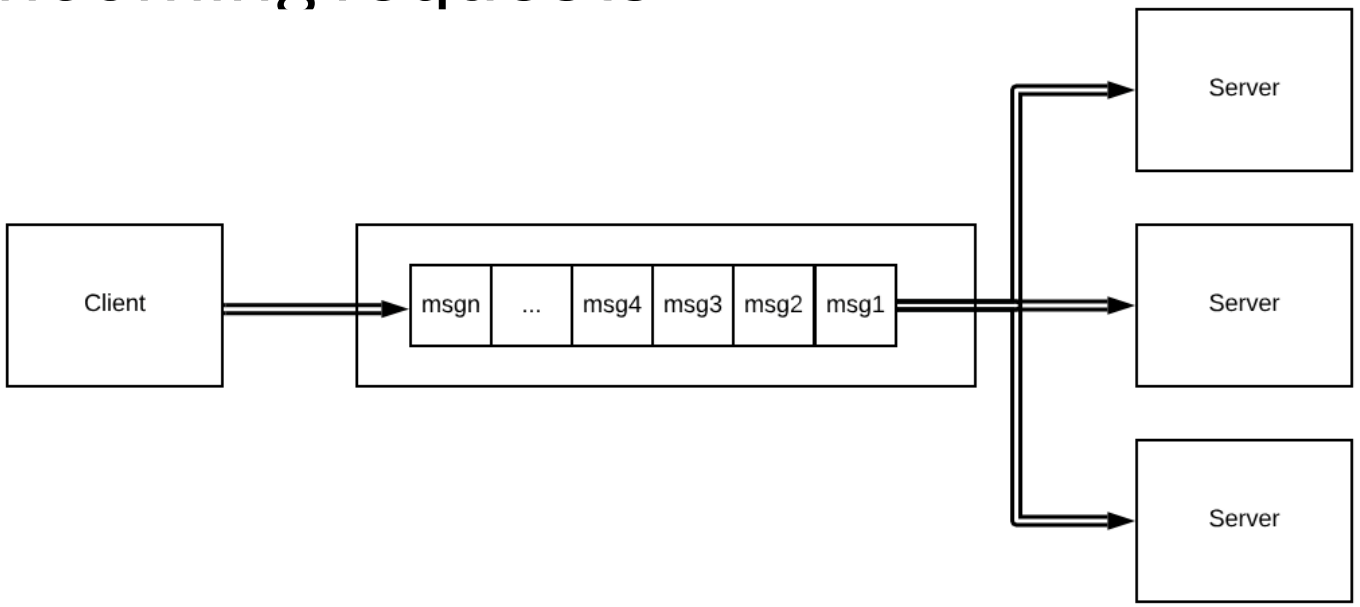


Advantage #1: Fault Tolerance

- Messages are preserved when the server is down
- Assuming the message queue runs on a reliable server

Advantage #2: Scalability

- Servers can be added dynamically to handle increased load
- Message queues also prevent server overload by buffering incoming requests



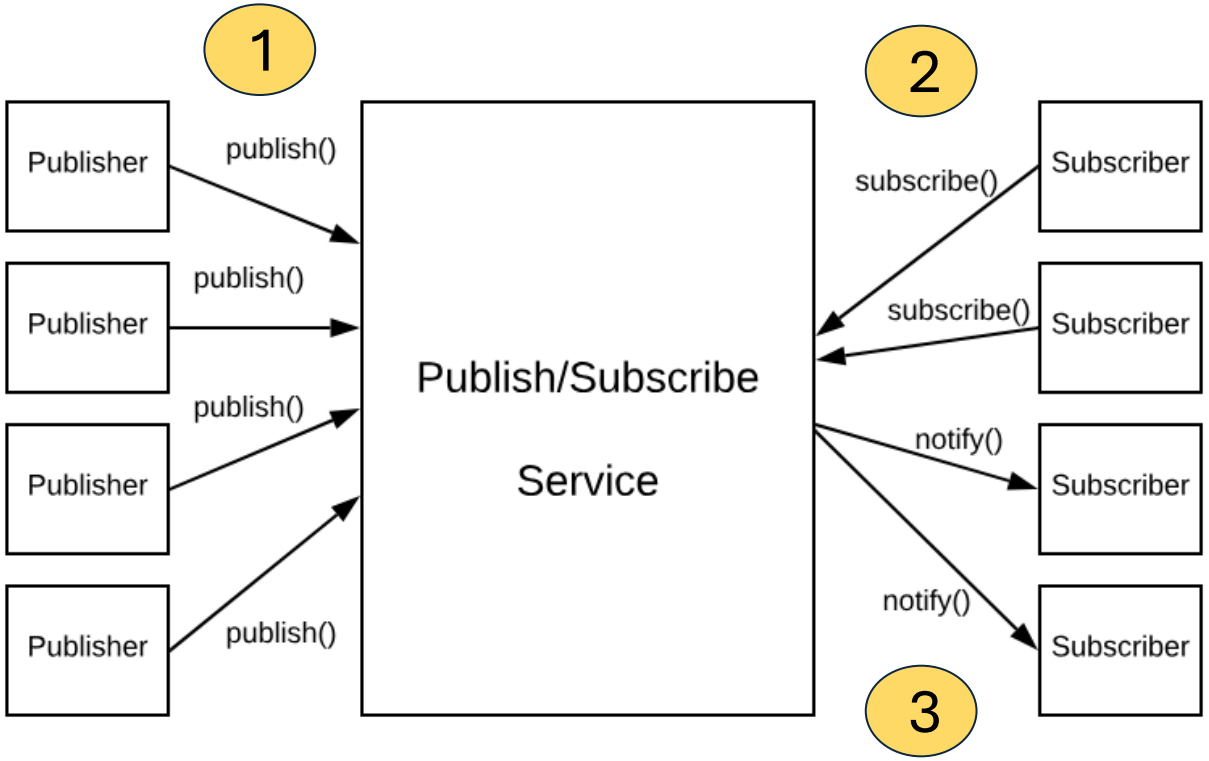
Publish/Subscribe Architecture

Publish/Subscribe

- Architectural pattern that extends message queue functionality
- Messages are called events

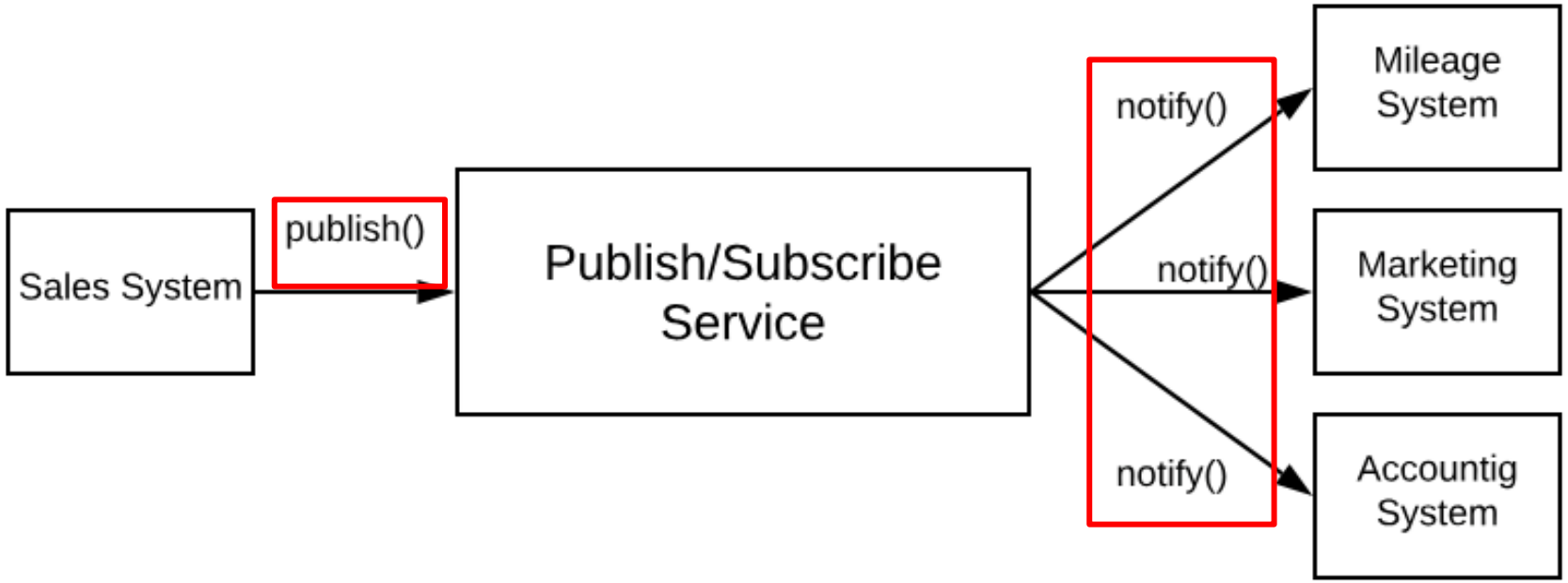
Publish/Subscribe

- Systems can (1) publish events; (2) subscribe to events; (3) receive notifications about events



Example: Airline System

- Event: ticket sale



Cloud Native and Serverless Architectures

Cloud Native

- Cloud Computing "… is the availability and process of delivering computing resources on-demand over the internet."[1]

- Cloud **Native:** "…is the software approach of building, deploying, and managing modern applications in cloud computing environments."[2]

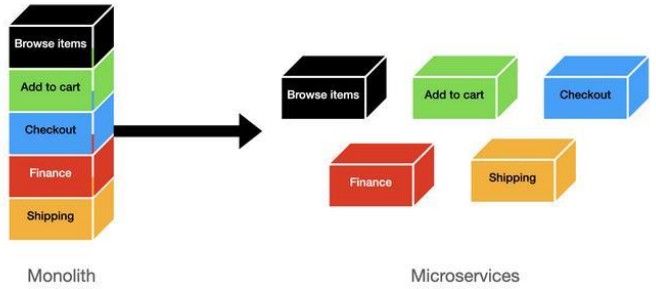
[1] CM, Safeer, *"Architecting cloud native serverless solutions: design, build, and operate serverless solutions on cloud and open source platforms"*
[2] AWS, "What is Cloud Native?"

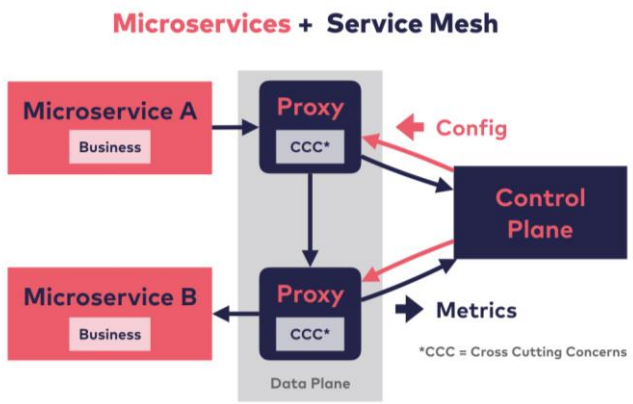
Cloud Native - Benefits

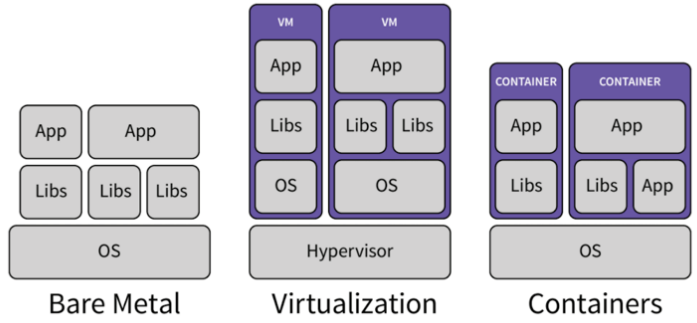
- Increased efficiency
  - Closely coupled with **agile practices** like DevOps, CD
- Reduced cost (potentially!)
  - No upfront investment in physical infrastructure
  - Not always cheaper over longer duration, depends on usage pattern and other factors
- Availability
  - Can lead to resilient and highly available application

Cloud Native – Architecture of Applications

- Based on **microservices**
  - small, interdependent services, loosely coupled
  - more scalable, better resilience
- Extensive use of **APIs**
- Use of ***Service mesh***
  - A software layer that manages interactions between multiple microservices
- Use of **Containers**
  - Software components that pack microservice code and other required dependenies in a cloud-native environment
  - Microservices can run independently of underlying “system” (OS, CPU architecture)







Cloud Native – Developing Applications

- Developing cloud native applications is a cultural shift; specific software practices are common. E.g.:
  - Continuous Integration
  - Continuous Deployment
  - DevOps
  - **Serverless**

Serverless Computing

- A cloud native model where cloud provider is responsible for managing the underlying server infrastructure
  - Does *not* mean no servers!
  - More about your experience as a developer – you perceive in some sense a “serverless” environment

- The CSP (Cloud Service Provider) manages:
  - provisioning the required infrastructure on the cloud
  - scaling the infrastructure up and down as needed
  - routine infrastructure management

- “Function as a Service” (FaaS) model is central to serverless
  - Though serverless is more than just FaaS
  - Serverless is the entire stack of services: includes e.g. **serverless databases,** and **serverless storage**

- Serverless architectures well suited to event-driven workloads

Serverless Computing

- Pros
  - Improved productivity
  - Pay for execution only (no idle time cost)
  - Develop in multiple languages
  - Streamlined with DevOps cycles
  - Can be cost-effective
  - Good visibility of usage
- Cons
  - Less control over hardware and execution environments
  - Vendor lock-in
  - Slow startup (“cold start”)
  - Testing and debugging is complex
  - Porting legacy applications for serverless implementation is complex
  - Higher cost for running long applications

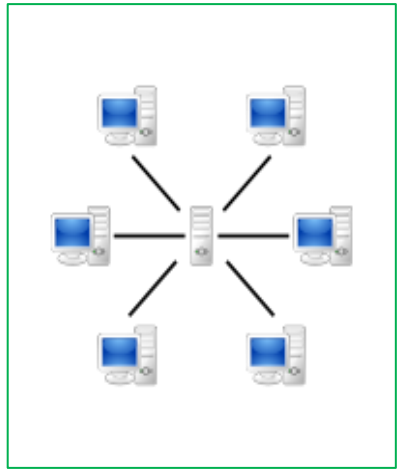
Other Architectural Patterns

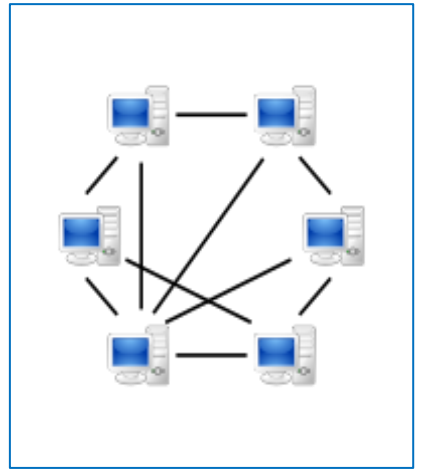
Other Architectural Patterns

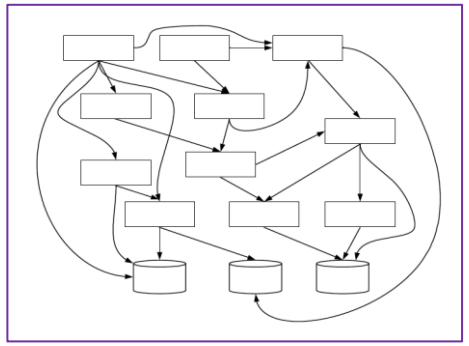
- Pipes and Filters
- Client/Serve
- Peer-to-peer
- “Anti-patterns”
  - E.g. “Big ball of mud”

Example: `ls | grep csv | sort`

filter pipe







Summary

- Different architectural styles are possible for building software systems
- Architectural choices have a long-term impact, and effects of right/wrong choices can also take a long time to appear
- Some styles we looked at:
- Some styles are more closely coupled with Agile methogologies

- Layered
- Model-View-Controller (MVC)
- Microservices
- Message-Oriented
- Publish/Subscribe
- Cloud Native and Serverless