Lecture 8: Architecture 2

CS 5150, Spring 2025

Administrative Reminders

- Project Report #2 is due on Feb 28 EOD.
- Assignment A2 due Feb 20
- Assignment A3 coming soon!

Previously on 5150…

Design steps

- Given requirements, must design a system to meet them
  - System architecture
  - User experience
  - Program design

- **Ideal**: requirements are independent of design (avoid implementation bias)
- **Reality**: working on design clarifies requirements
  - Methodology should allow feedback (strength of iterative & agile methods)

Design principles

- Design is an especially **creative** part of the software development process
  - More a "craft" than a science
  - Many tools are available; must select appropriate ones for a given project
- Strive for simplicity
  - Use modeling, abstraction to (hopefully) find simple ways to achieve complex requirements
  - Designs should be easy to implement, test, and maintain
- Easy to use correctly, hard to use incorrectly
- **Low coupling, high cohesion**

Levels of abstraction

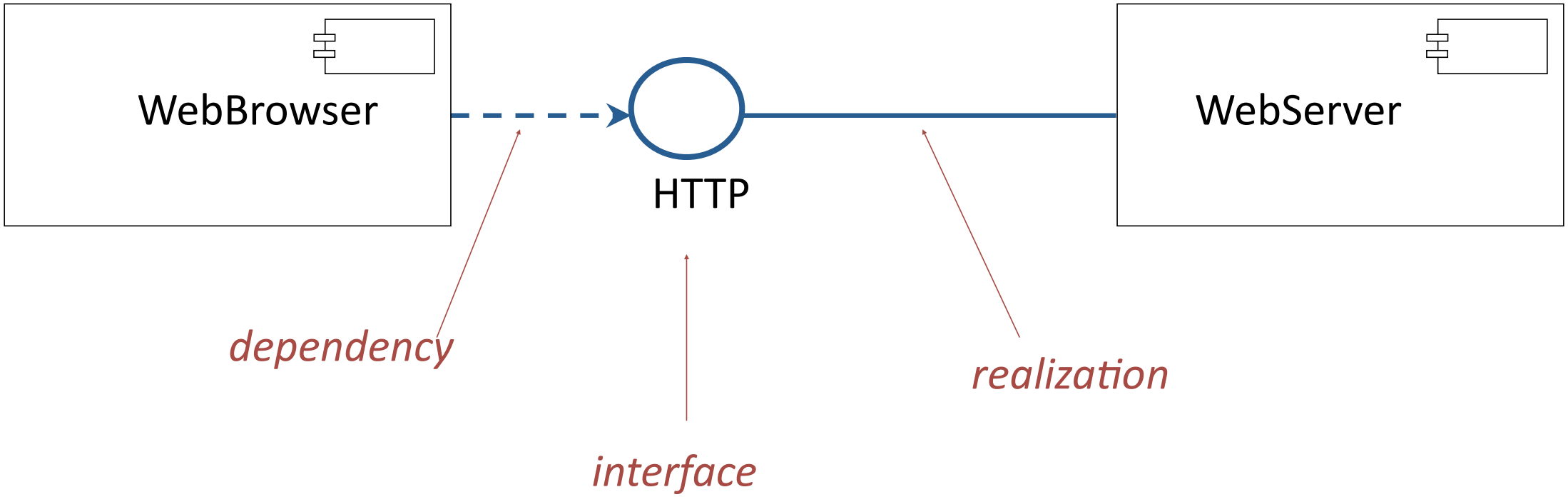
- Requirements
  - High-level "what" needs to be done
- Architecture
  - High-level "how"
  - Mid-level "what"
- Program design (Design patterns)
  - Mid-level "how"
  - Low-level "what"
- Code
  - Low-level "how"

- Documentation for each step should respect its level of abstraction
  - Avoid biasing later steps
  - Avoid redundancy

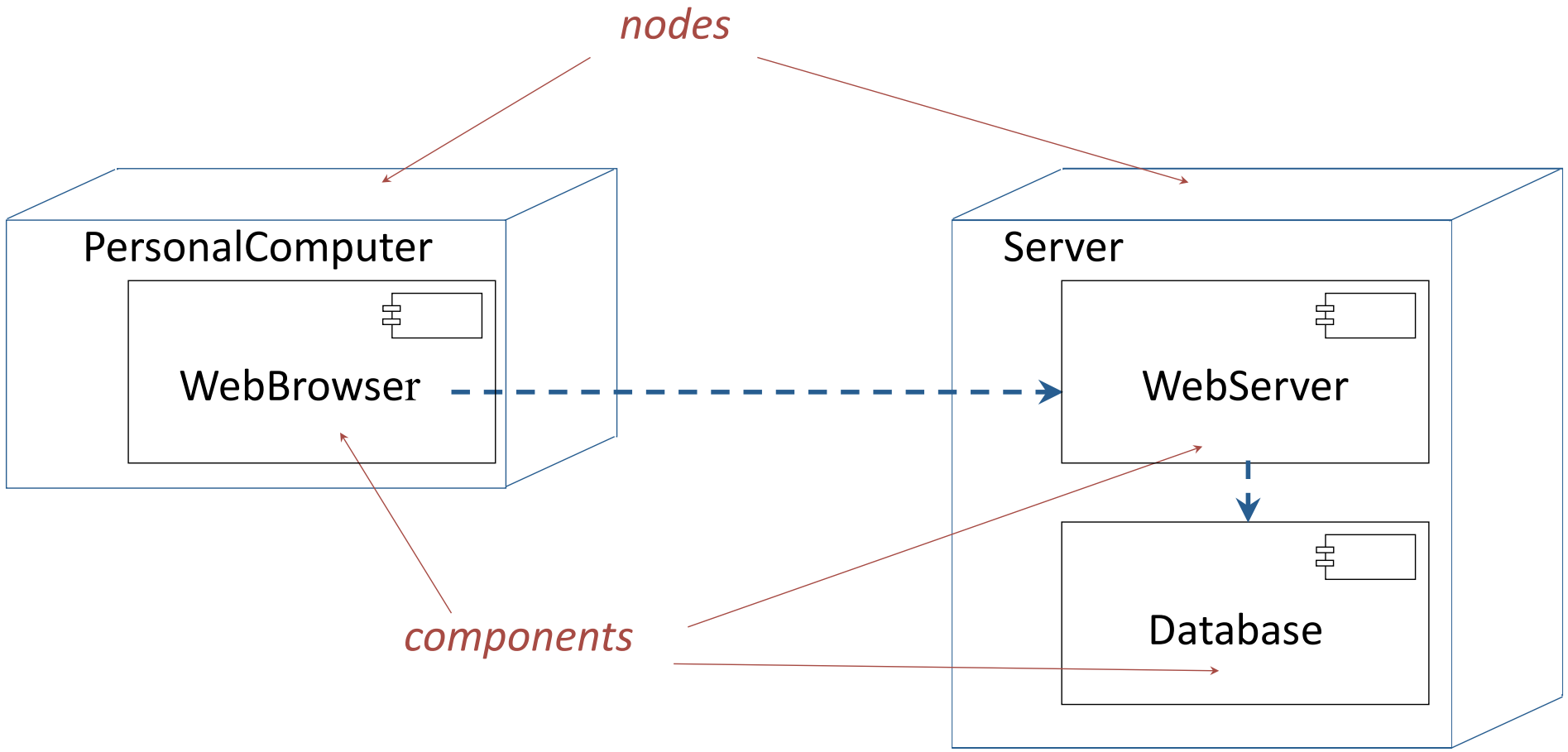
Architectural considerations

- Infrastructure
  - Hardware
  - Operating systems
  - Virtualization
- Interfaces
  - Networks/buses
  - Protocols
- Services
  - Databases
  - Authentication
- Operations
  - Testing
  - Logging/monitoring
  - Backups
  - Rolling deployment
- Product line

Example: interface diagram



Example: deployment diagram

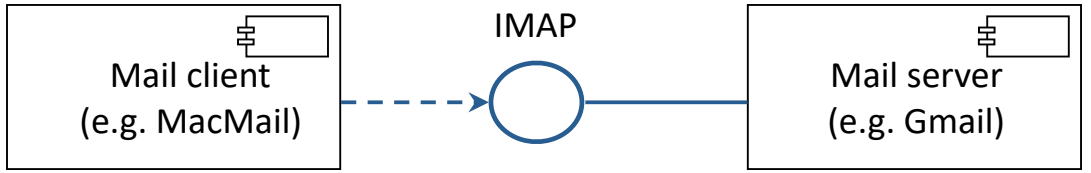


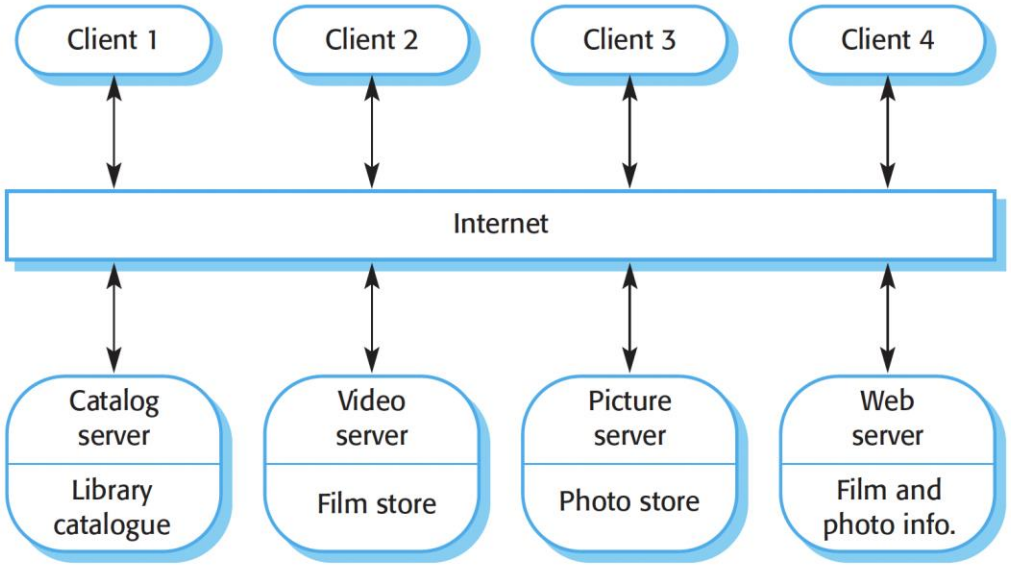
Poll: Account data will be stored in a PostgreSQL database running in a Docker container under Linux on an HPE ProLiant server that also runs the web server. Which architectural diagram is an appropriate place to show these details?

PollEv.com/cs5150sp25

Client/Server

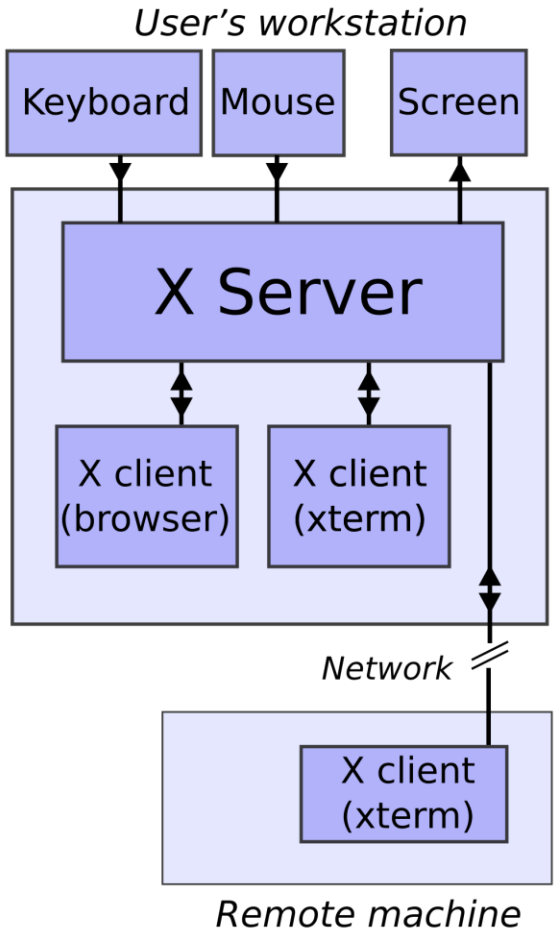
- Control flow in client and server are independent
- Communication follows a protocol
- If protocol is fixed, either side can be replaced independently
- Peer-to-peer: same component can act as both client and server





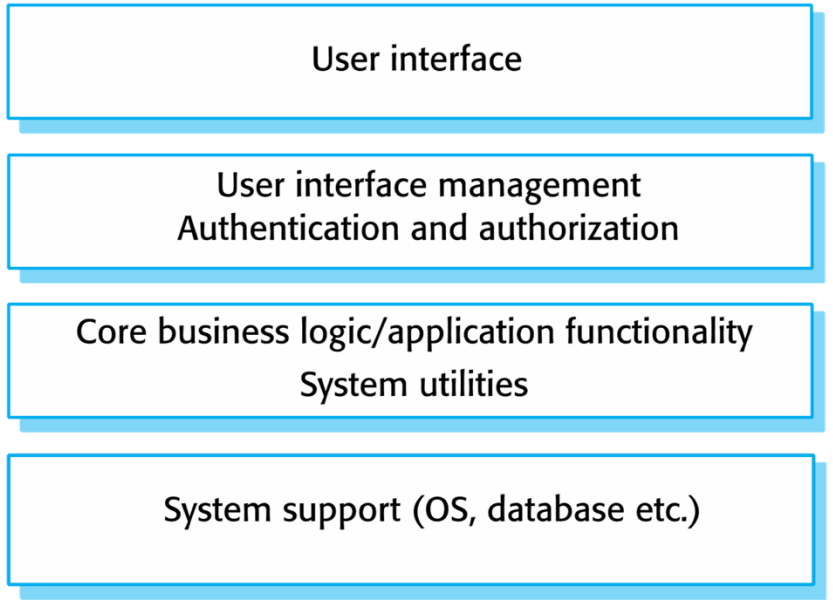
Example: X Window System (X11)

- X server runs on computer w graphic display
- Client application (browser) connects to server via X11 protocol
- Server send input events, client sends drawing commands
- Confusingly, “X11 client” runs on “application server”, while “X11 server” runs on “thin client”



Layered Architecture

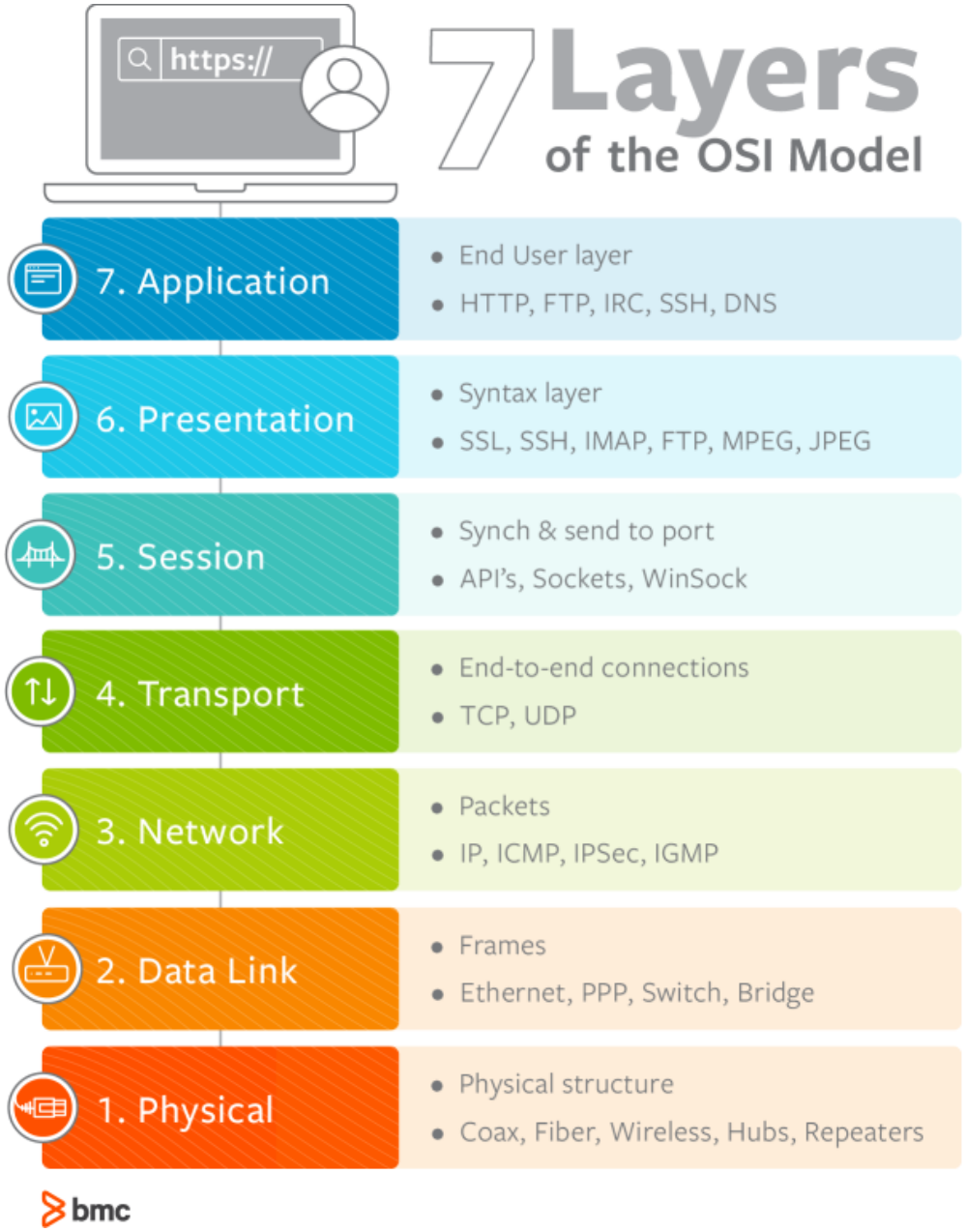
- Partition subsystems into stack of layers
  - Layer provides services to layer directly above
  - Layer relies on services to layer directly below
- Advantage: constrains coupling
- Danger: leaky abstractions
  - Clear separation is difficult
  - May need services of multiple lower layers
  - Performance



Sommerville, *Software Engineering*

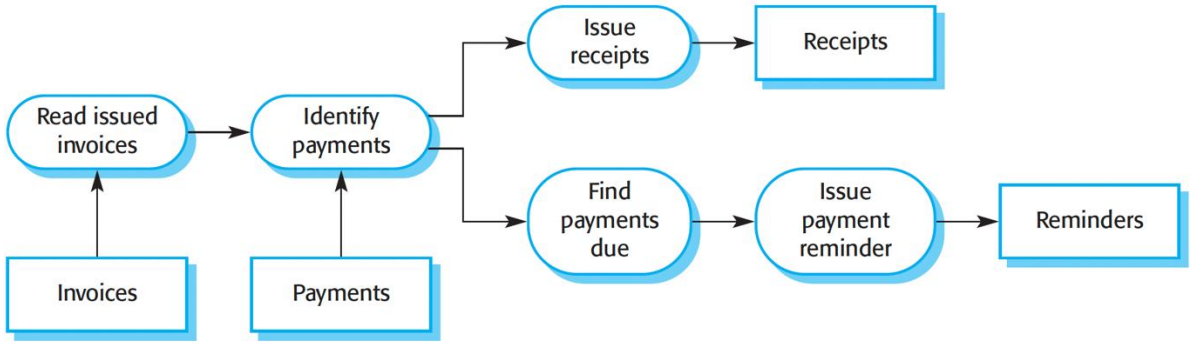
Example

- OSI Reference Model
- Used for network protocols (TCP/IP)

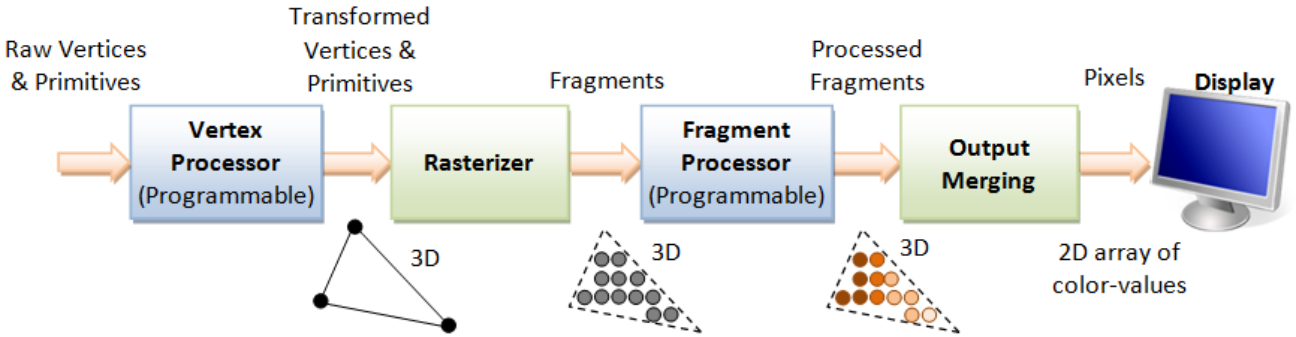


Pipe and Filter

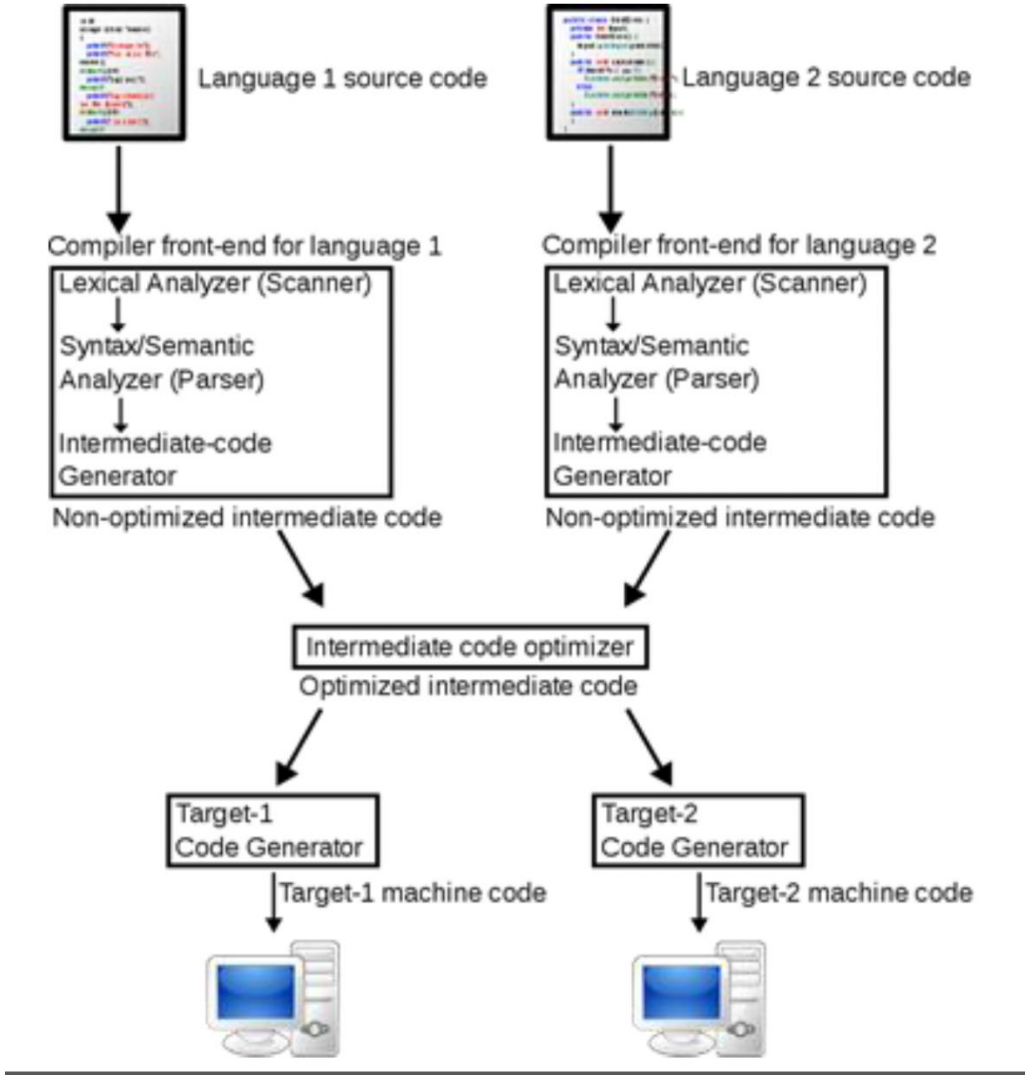
- Transformation components process inputs to produce outputs
  - Subsystems coupled via data exchange
  - Good match for data flow models
  - May be dynamically assembled
  - Limited user interaction
- Applications:
  - Compilers
  - Graphics shaders
  - Signal processing
- Caveats:
  - Awkward to handle events (interactive systems)
  - Rate mismatches if branches merge



Examples

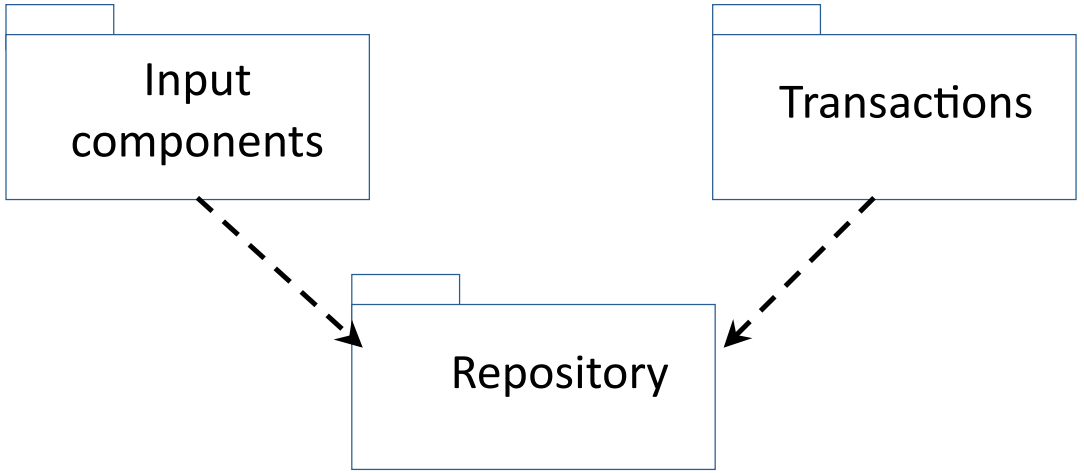


3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.



Repository

- Couple subsystems via shared data
  - Repository may need to support atomic transactions
- Advantages:
  - Components are independent (low coupling)
  - Centralized state storage (good for backups)
  - Changes propagated easily
- Dangers:
  - Bottleneck / single point of failure



Poll: Consider a real-time data processing system continuously collecting and analyzing log files from multiple distributed servers. The system should filter, aggregate, and store logs while allowing administrators to query historical data efficiently. Which architectural pattern would be most appropriate, and why?

PollEv.com/cs5150sp25

Lecture goals

- Identify common architectural styles (continued)
  - Three tier architecture
  - Model-view-controller
- Encapsulate deployments using virtualization

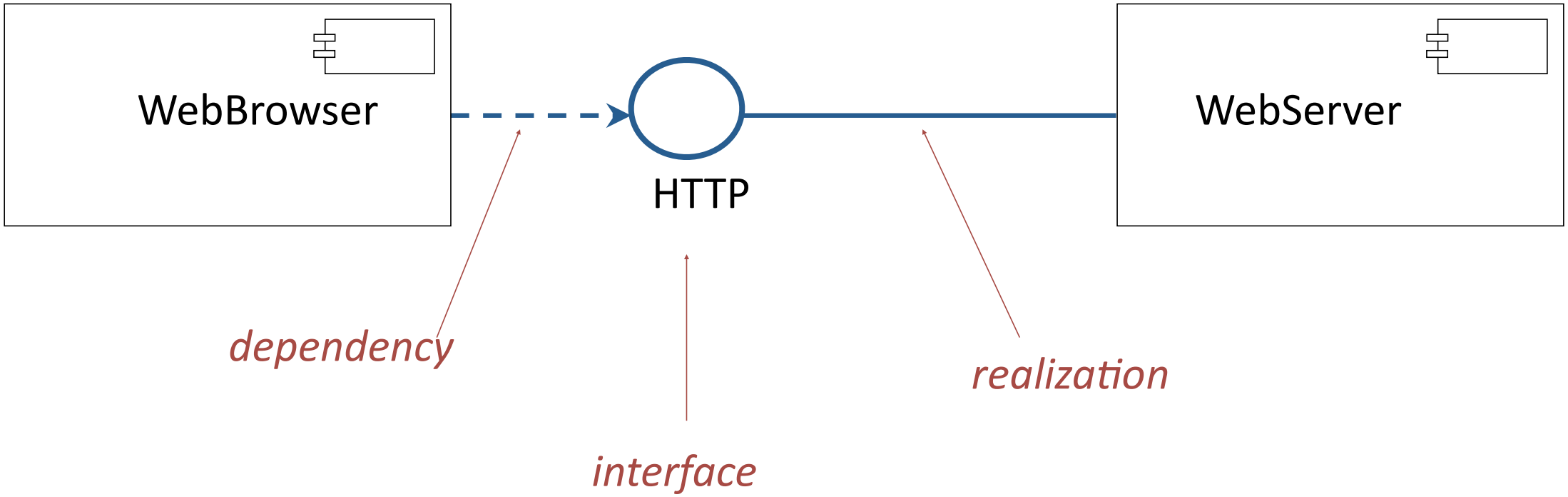
Architectural styles

… continued from Lecture 7

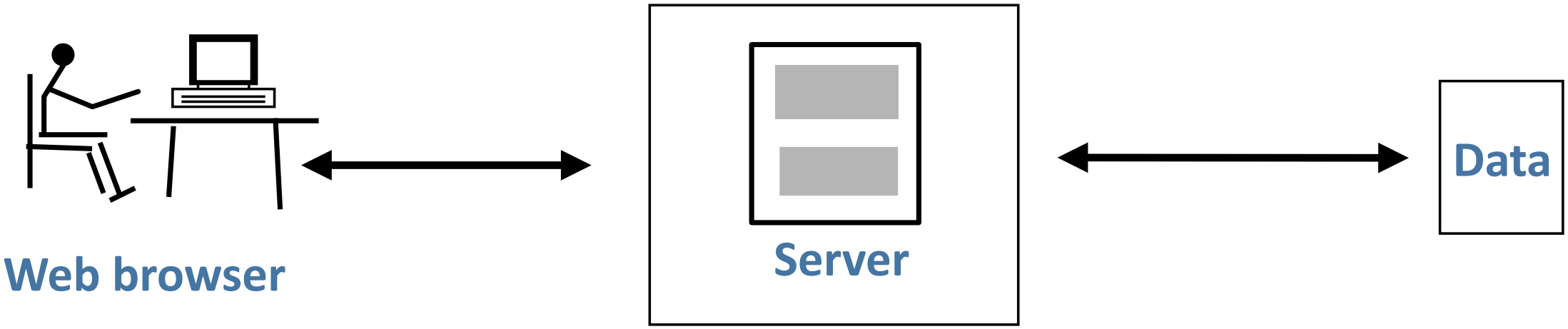
Three tier architecture

- Extension of client/server model
- Commonly used for small-medium websites
  - Classic example: LAMP stack for web applications (Linux, Apache, MySQL, PHP/Python)

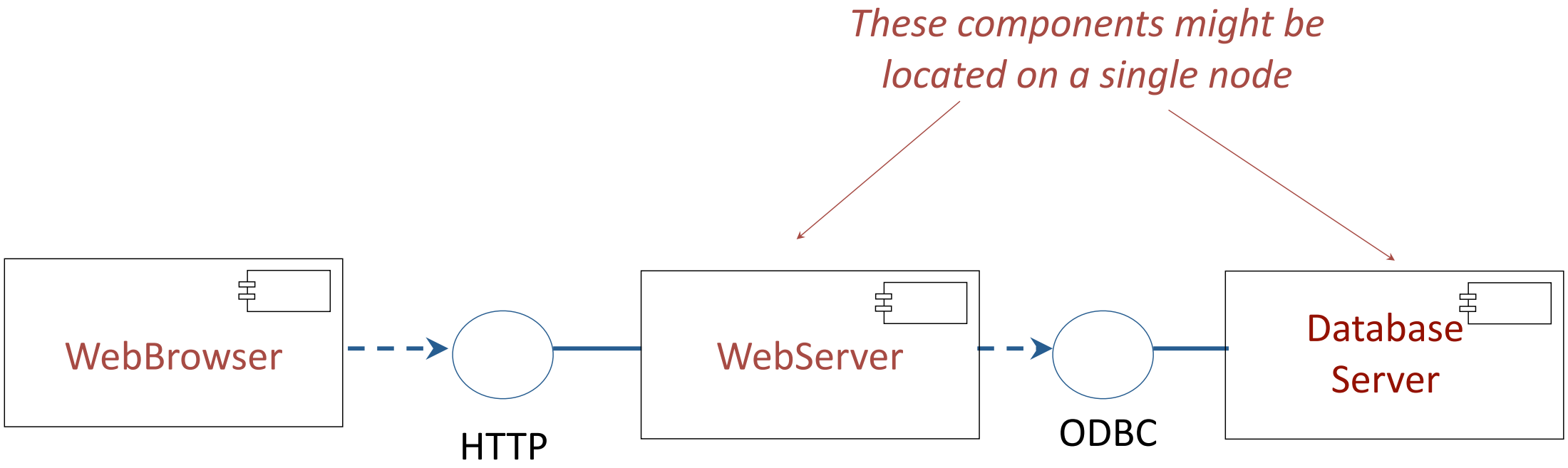
Basic website (client/server)



Extension: data store



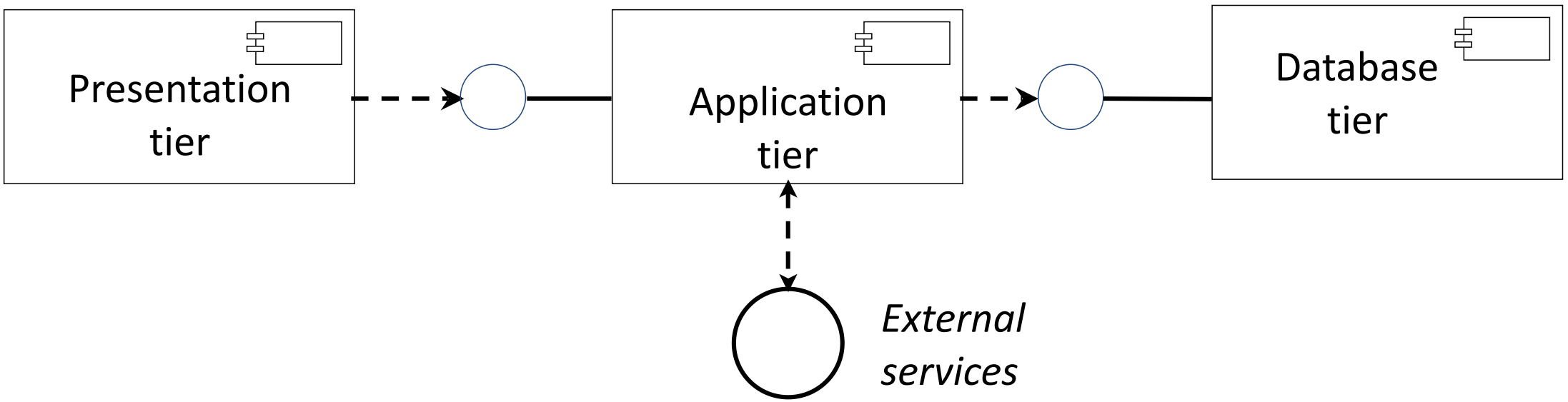
Component diagram



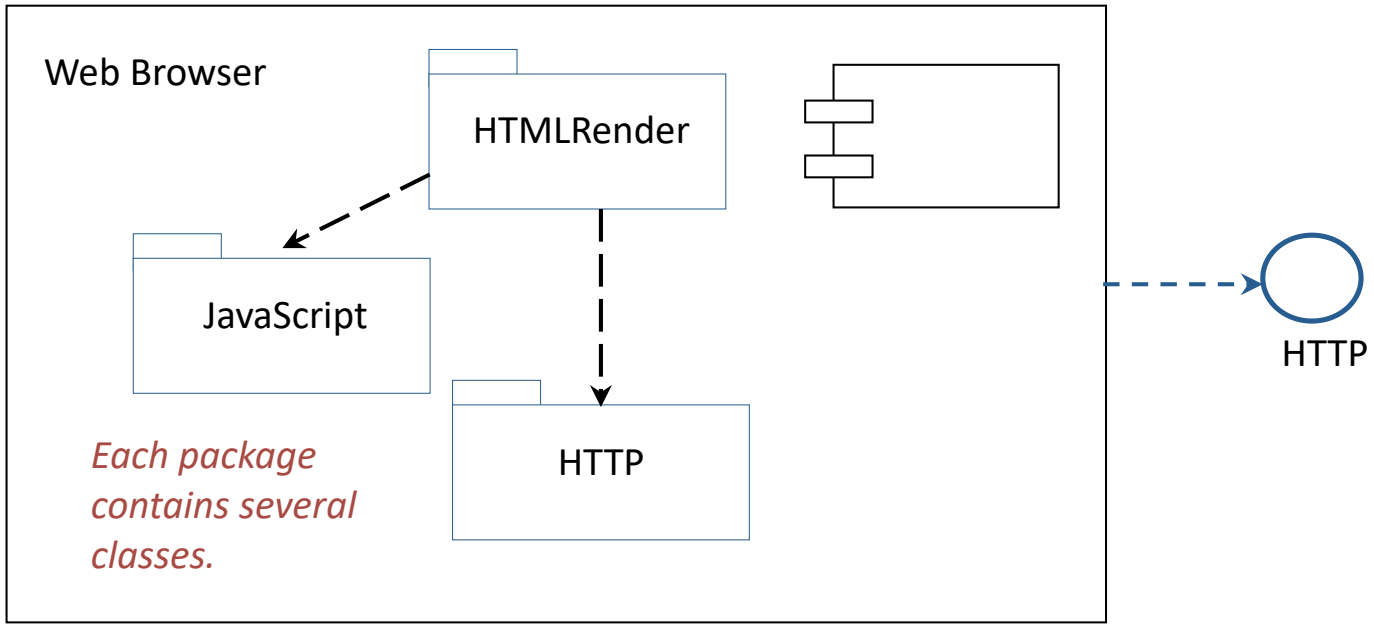
Significance of components (replaceable binary elements):

- Any web browser can access the website
- Database can be replaced by another that supports the same interface

Three tier architectural style



Presentation tier complexity



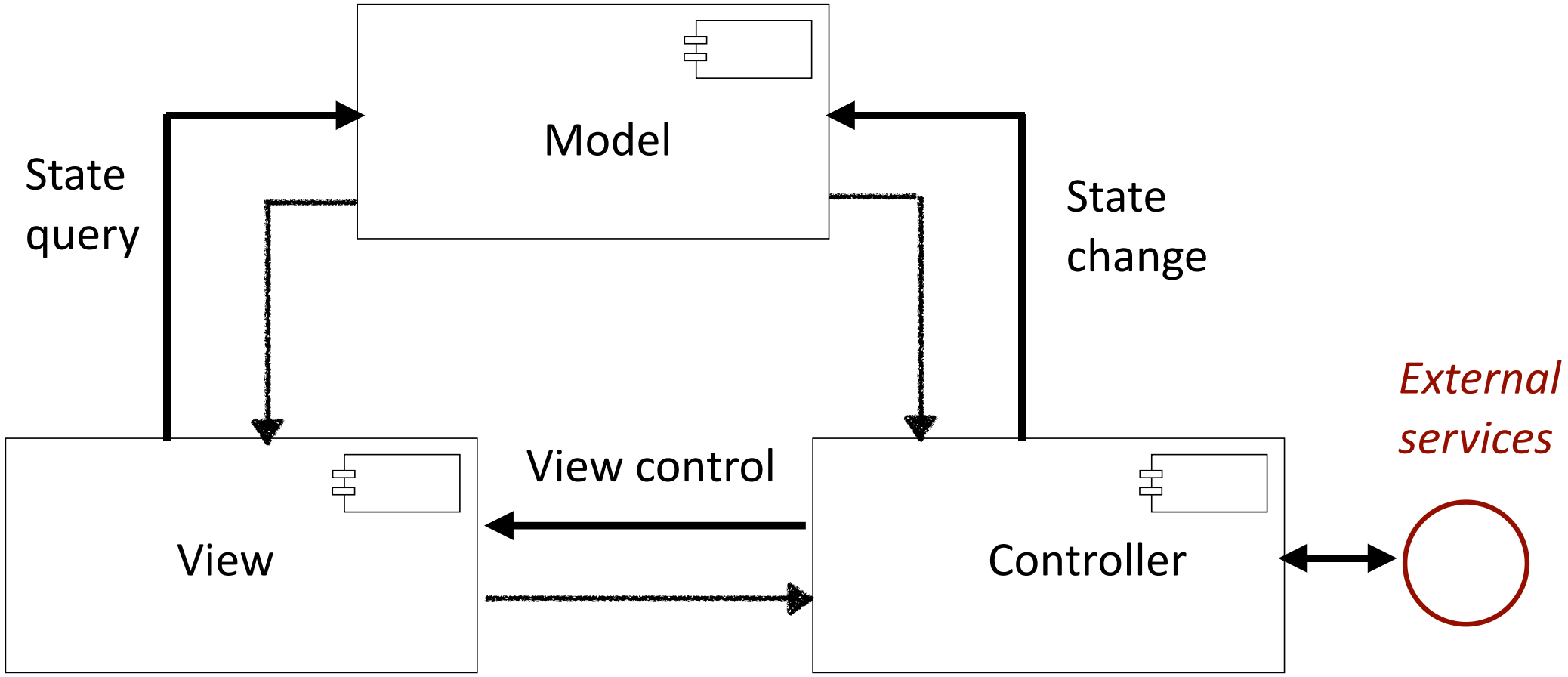
Presentation tier may house internal complexity, but as long as it supports the same interface, it is still a binary-replaceable component

Model-View-Controller

- Beware: many variations
  - Some are architectural styles: system-level responsibilities partitioned into different components
    - Example: **Play Framework** for building web apps
  - Some are program design patterns: functionality divided between different classes
    - Focus on reusable controls
    - Example: **Swing widgets**
    - Variation on which logic is widget-level vs. form-level (MVC vs. MVP)
    - Variation on which classes communicate directly (MVC vs. MVA)
    - Variations in model storage (domain objects, DB record sets, immutable store)

Read more: https://martinfowler.com/eaaDev/uiArchs.html

Component diagram



Features of MVC

- Separated presentation
  - Decouple model and view (replaceable components)
  - Multiple (possibly simultaneous) views supported

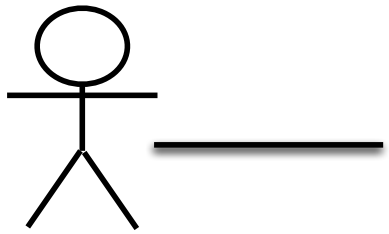
Example: "mission control" terminal

(based on a past CS 5150 project)

- A vehicle (unmanned aircraft) is flown by a pilot interfacing with a computer terminal on the ground
- Vehicle communicates with ground station via radio signals
  - Actuation commands (uplink): change throttle, angle flaps, etc.
  - Sensor measurements (downlink): air speed, GPS position, actuator settings, etc.

Example: View

- Graphical user interface shows model properties (sensor readings, derived state) as instrument dials and provides input widgets for commanding actuators



Pilot

Example: Model

- Maintains record of state of vehicle (speed, fuel, …)
- Computes derived properties of vehicle (rate of turn, predicted trajectory, …)
- Updates state in response to actions from controller (e.g. new telemetry received)
- Provides view with information to be displayed to user

Different vehicles will need different models but might not require new views or controllers.

Example: Controller

Scenario: Pilot wishes to change flap angle to 20 deg to increase lift and accommodate a slower speed for landing.

1. View sends message to controller: `setFlaps(20)`
2. Controller sends radio command to vehicle: `setFlaps(20)`
3. Vehicle acts on command and replies to controller: `flapsSet(20)`
4. Controller relays telemetry to model: `flapsSet(20)`
5. Model updates state and recomputes stall speed
6. Model notifies view of new state (flap setting, stall speed)
7. View displays new state in user interface

View

- Presents application state and controls to user
- Typically subscribes to model for notifications of state changes
  - "Observer pattern"
- Responsible for rendering to a particular interface
  - Drawing graphics, generating HTML, printing text
- Sends user input to controller
- A single model can support multiple views
  - Example: web app, native app

Model

- Records state of application and notifies subscribers
  - Responds to instructions to change state (from controller)
- Does not depend on either controller or view
- State may be stored in objects or databases
- May be responsible for some application logic (e.g. input validation)

Controller

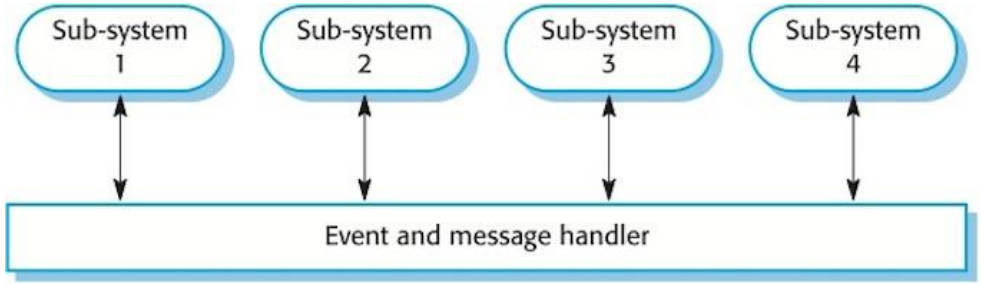
- Manages user input and navigation
- Defines application behavior
- Maps user actions to changes in state (model) or view
- May interact with external services via APIs
- May be responsible for some application logic (e.g. input validation)

- Variety in distribution of duties between model and controller

Deployment vs. component diagrams

- Example: server-side web application
  - View consists of templates (executed on server), stylesheets, JavaScript (executed on client)
  - Client browser responsible for rendering, input handling
- Execution of view elements may be distributed across multiple nodes, but from developer's perspective, they form one component

Publish-subscribe



- Event-driven control
  - Application responds to external stimuli and timeouts
  - No centralized orchestration
- Very loose coupling – components communicate via message broker
  - Easy to extend
  - Difficult to analyze (observer pattern)
    - No control over what (if any) code responds to an event
    - Potential for conflicts (multiple components respond in incompatible ways)
    - Potential for silently dropped events
    - Call stacks may not reflect causality

Closing remark

- Beware software architectures that resemble corporate hierarchy
  - Refactoring more disruptive than reorgs
  - Be aware of and accommodate political context, but architecture should serve the application more than the developer

Virtualization

Deployment concerns

- Dependency conflicts
- Configuration, data sprawl
- OS portability
- Unintended interactions
  - Filesystem has same problems as global variables

- Solution: Encapsulation; but...
  - Deploying on separate machines risks under-utilization

Virtual machines

- Multiple OS instances running on one machine
  - Real hardware is managed by host OS or hypervisor
- Improves hardware utilization, reduces cost
  - Avoids energy consumption by redundant hardware
- Stateful – still risks data sprawl
  - Address with automated administration
- High overhead – software redundancy

- **Examples**: VMware, VirtualBox, Xen, Hyper-V

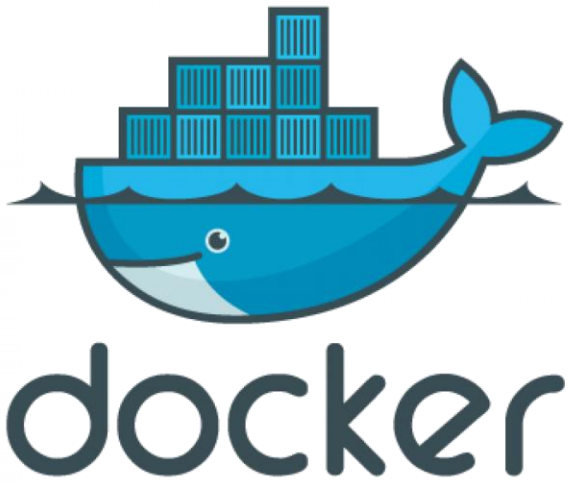
System configuration management

- Automate deployments
  - Installing dependencies
  - Configuring OS
  - Configuring application
- Combat sprawl

- **Examples**: Ansible, Puppet, Chef, Vagrant

Containers

- Trade OS heterogeneity for reduced redundancy
- Still isolate filesystem, network without duplicating OS
- Lightweight – new instances start quickly
  - Improves elasticity
- Often encapsulates a single application
- Often treated as stateless (don't write to filesystem)



- Examples: Docker, LXC

"Serverless"

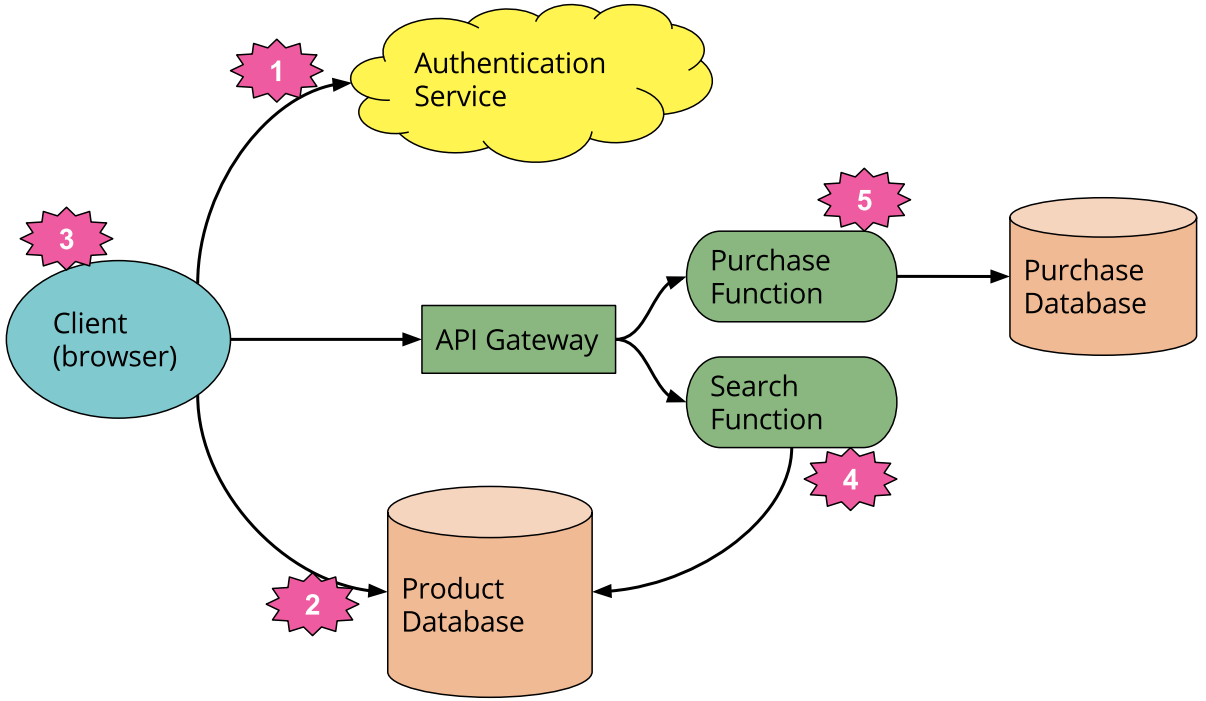
- Computation nodes are stateless, ephemeral, and event-triggered
  - Data store services still persist state, but are application-agnostic
- Application decomposed into event-handler functions
  - Event dispatch, container lifetime managed by platform
- Examples: Amazon Lambda, Azure Functions

Azure Functions

AWS Lambda

https://martinfowler.com/articles/serverless.html

Three-tier vs. serverless

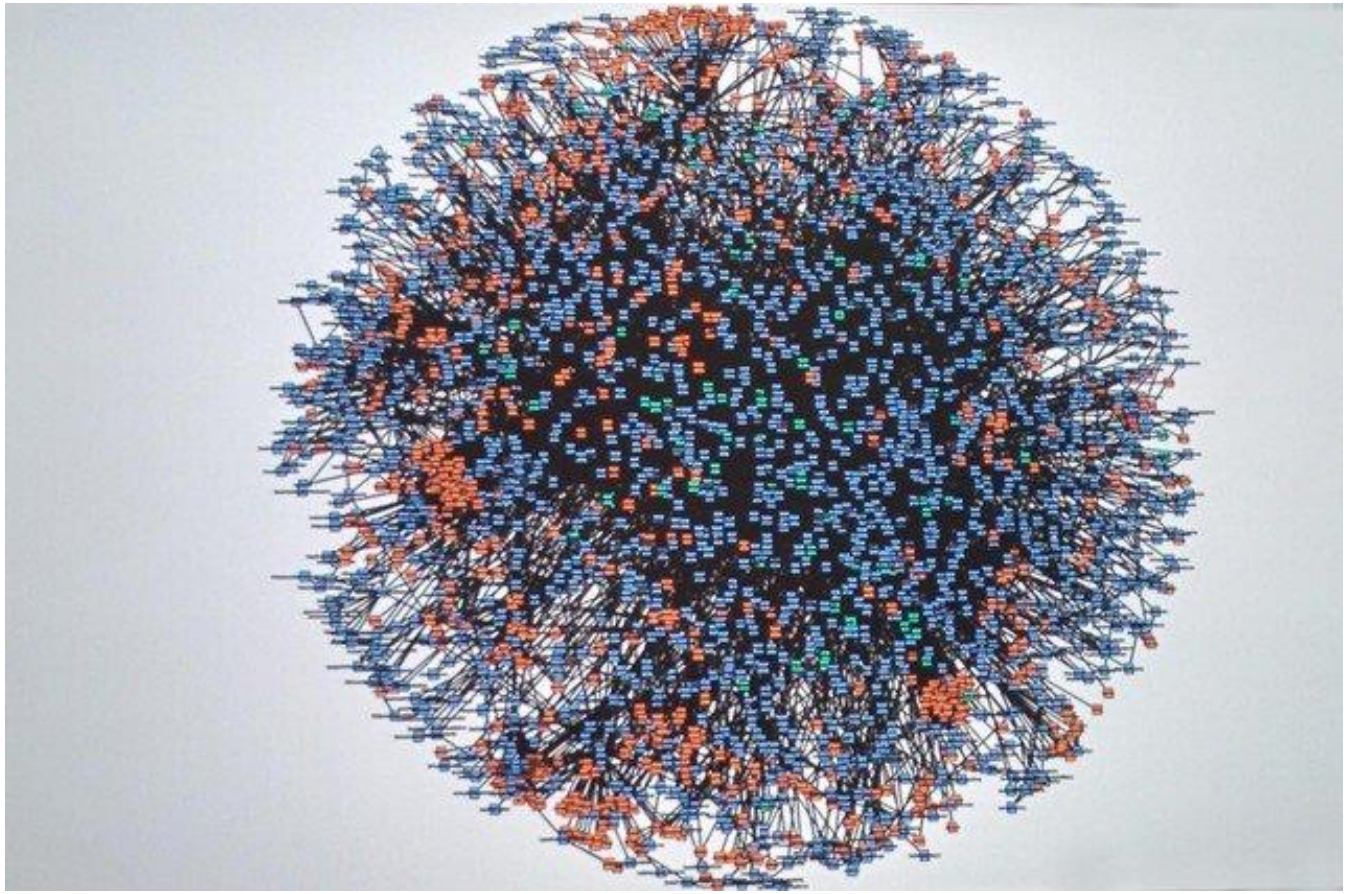


https://martinfowler.com/articles/serverless.html

Microservices

- Components encapsulate services and expose them via standard interfaces. Are ideally binary-replaceable
  - In practice, many frameworks for managing modular applications are language-specific (e.g., OSGi for Java)
  - OOP abstractions like objects, methods are complicated at language boundaries and distributed deployment
- Microservices constrain component definition to **reduce coupling**
  - Language-agnostic protocols (e.g., RESTful HTTP)
  - Independently deployable
- Advantage: More scalable, fault tolerant, rapid roll out
- Disadvantage: Complex monitoring, more points of failure, network delays, testing is challenging
- Examples: Netflix, Amazon, Uber

Amazon’s Microservices Architecture (2008)



https://x.com/Werner/status/7416735145671434 24

Software Architecture Resources

- An Introduction to Software Architecture: David Garlan and Mary Shaw
- Software Engineering, Ian Sommerville: Chapter 6
- https://martinfowler.com/architecture/