CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 3.1 Software Architectures

Jon Bell, John Boyland, Mitch Wand

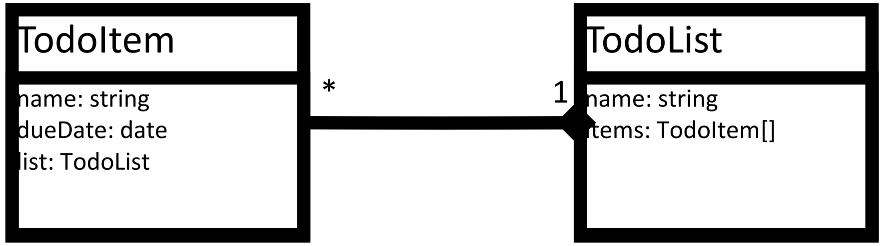
Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
  - explain why software architecture is important
  - list a few of the properties that an architecture may have (the "ilities")
  - describe the basic ideas of the following architectures, with examples and pictures
    - monolithic
    - layered
    - pipeline
    - microkernel
    - event-driven
    - microservice

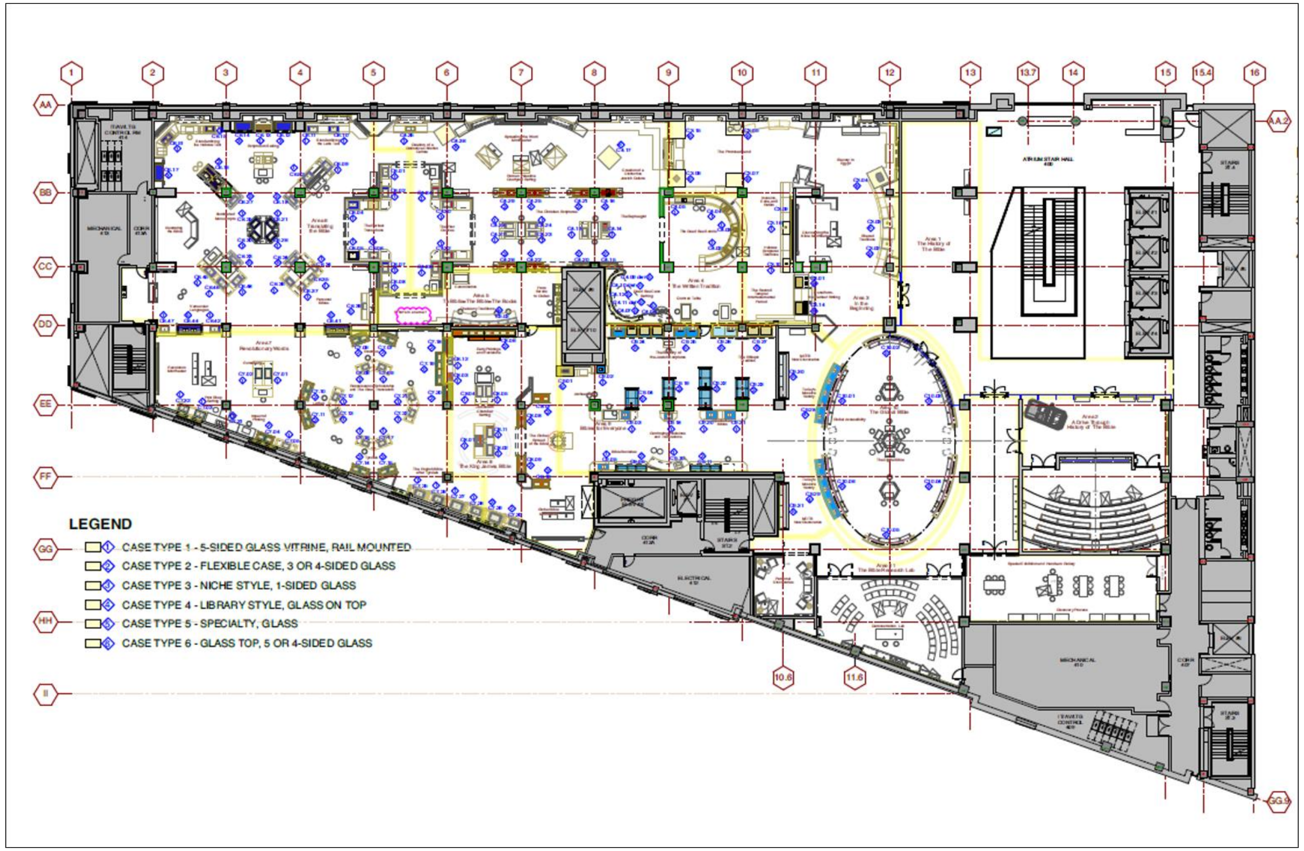
Design in this class so far: the details

- Metaphor: building architecture



Design at larger scales

- Metaphor: building architecture
- How do the pieces fit together? What do we reuse?

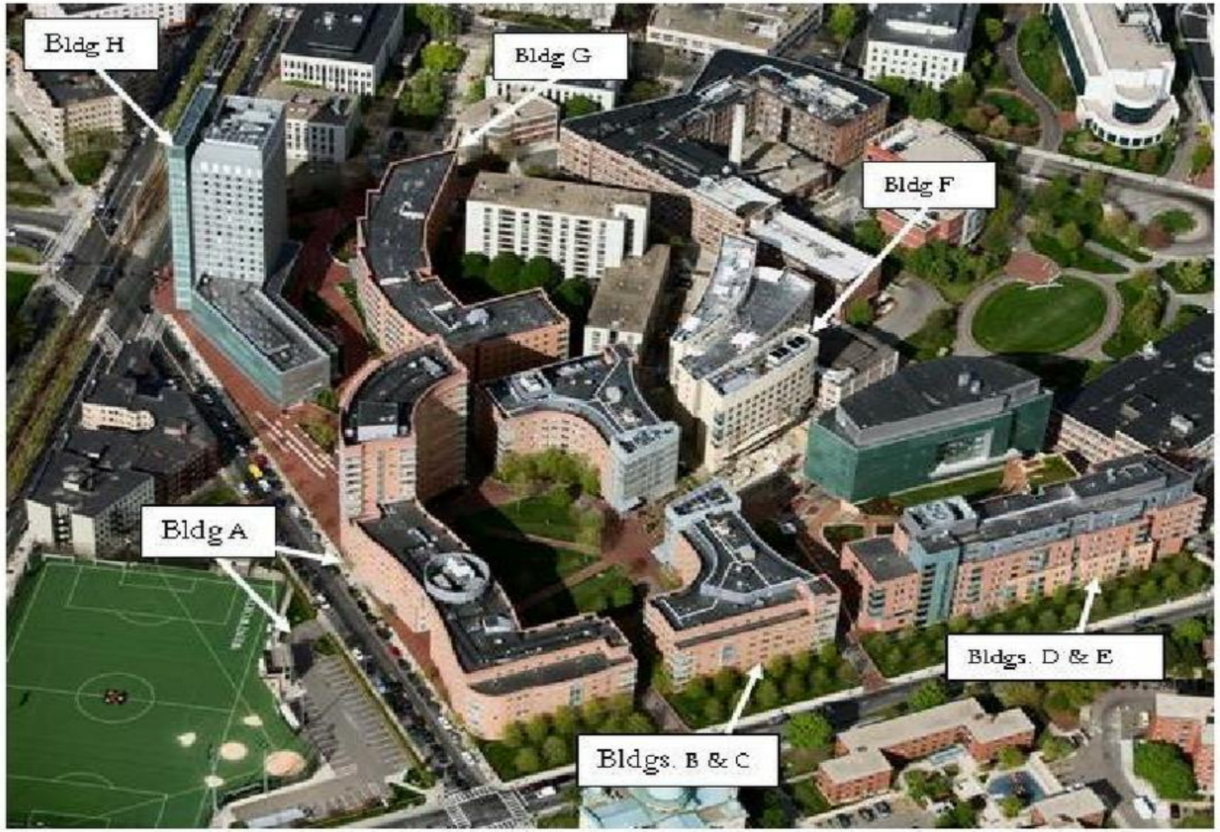


Design at larger scales

- Metaphor: building architecture
- How do the pieces fit together? What do we reuse?
- How do we organize into teams?

Design at larger scales

- How well will our system work in its context?



Goal: Create a high-level model of the system

- Abstract details away into reusable components
- Allows for analysis of high-level design before implementation
- Enables exploration of design alternatives
- Reduce risks associated with building the software

Properties of Software Architectures (the "ilities")

Table 4-1. Common operational architecture characteristics	
Term	**Definition**
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there's a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

from Richards & Ford: Fundamentals of Software Architecture

More ilities...

Table 4-2. Structural architecture characteristics

Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

from Richards & Ford: Fundamentals of Software Architecture

And still more ilities

Table 4-3. Cross-cutting architecture characteristics	
Term	**Definition**
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

from Richards & Ford: Fundamentals of Software Architecture

- We don't have time to study these in any detail, or to try to discuss how any particular architecture might rate on any of them.
- You could write a whole book about that...

Our goal:

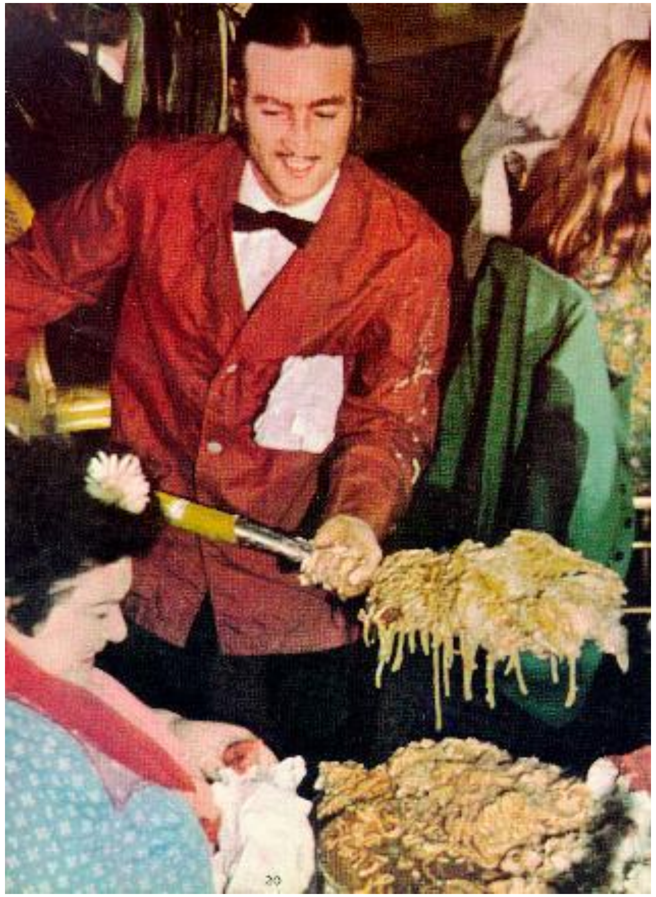
- Just talk about some different top-level organizations.
- Knowing the top-level organization gives you the first clue about
  - how to understand the system
  - where to look for bugs or explain behaviors
  - how to organize into teams
  - how to find modification and extension points

> Remember the overall goal of making software systems understandable by humans.

Architecture #0: Monolithic

- A single app, with no particular organization
- Also known as: "spaghetti code"
- May still have useful interfaces for some degree of encapsulation and modularity.
  - but is there a method to the madness?

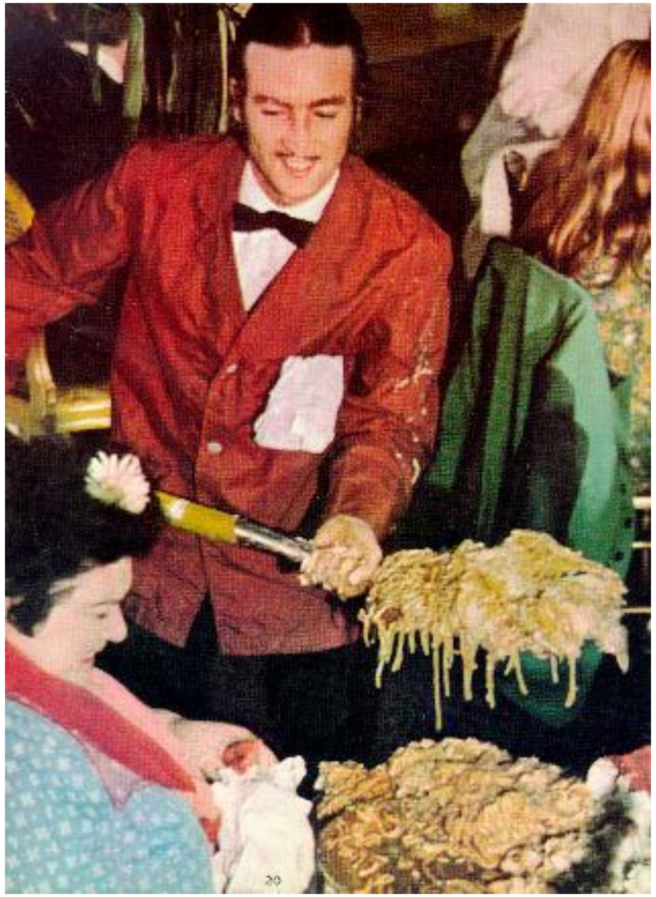
Shakespeare, *Hamlet*. The exact quote is: "Though this be madness, yet there is method in't" (Polonius, Act 2, Scene 2)



Brian Foote and Joe Yoder

Architecture #0: Monolithic

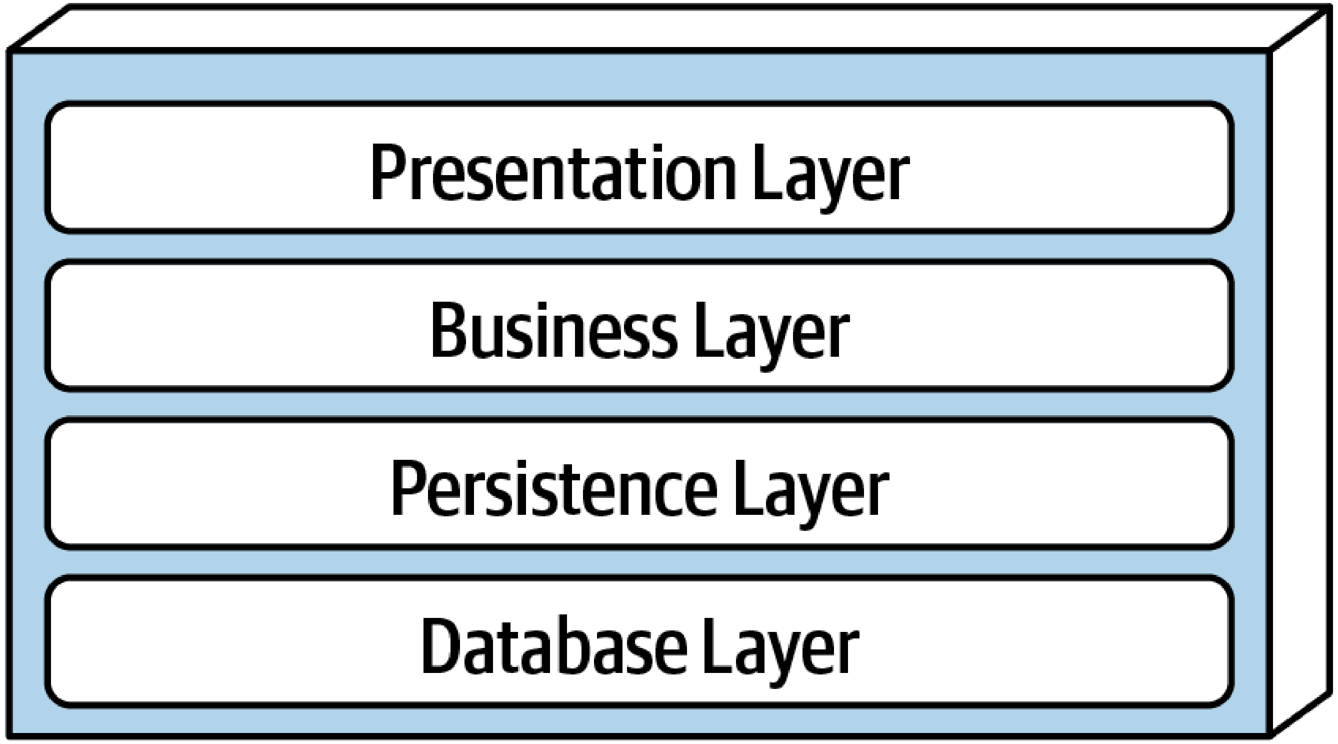
- OK for single-developer, short-lived projects
- But
  - what happens if you want to add a new developer
  - what happens if you need to come back to the code later?



Brian Foote and Joe Yoder

Architecture #1: Layered

- Each layer depends on services from the layer or layers below
- Organize teams by Layer
  - different layers require different expertise
- When the layers are run on separate pieces of hardware, they are sometimes called "tiers"



Layered Architecture (contd)

- Typical organization for operating systems
- Layers communicate through procedure calls and callbacks (sometimes called "up-calls")

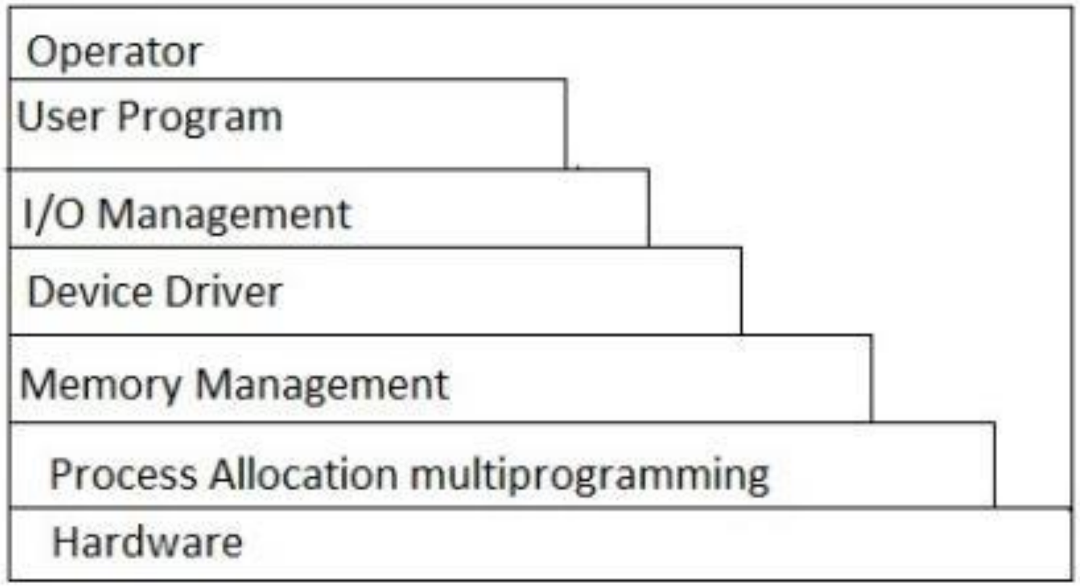
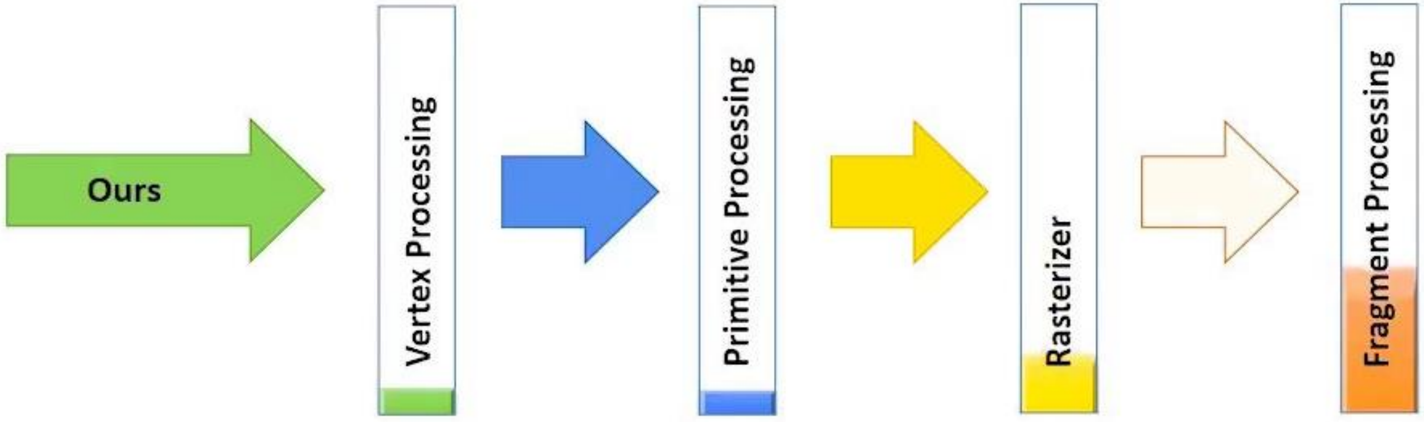
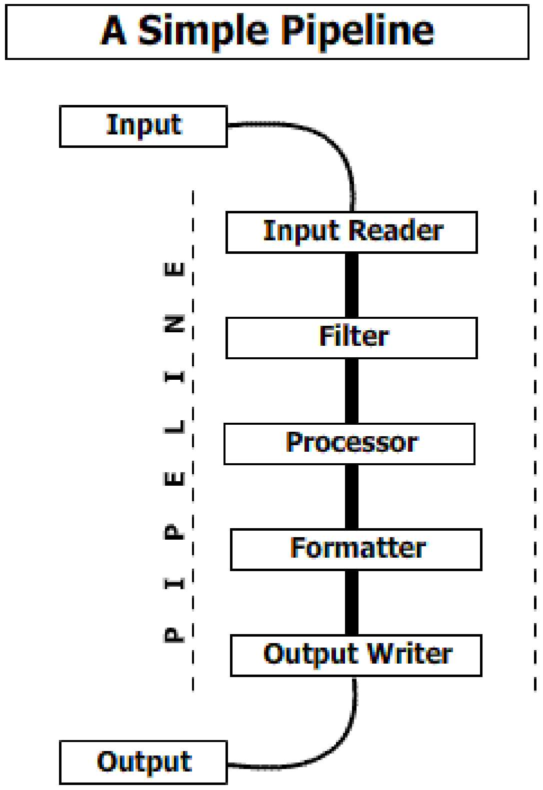


fig:- layered Architecture

Architecture #2: Pipeline

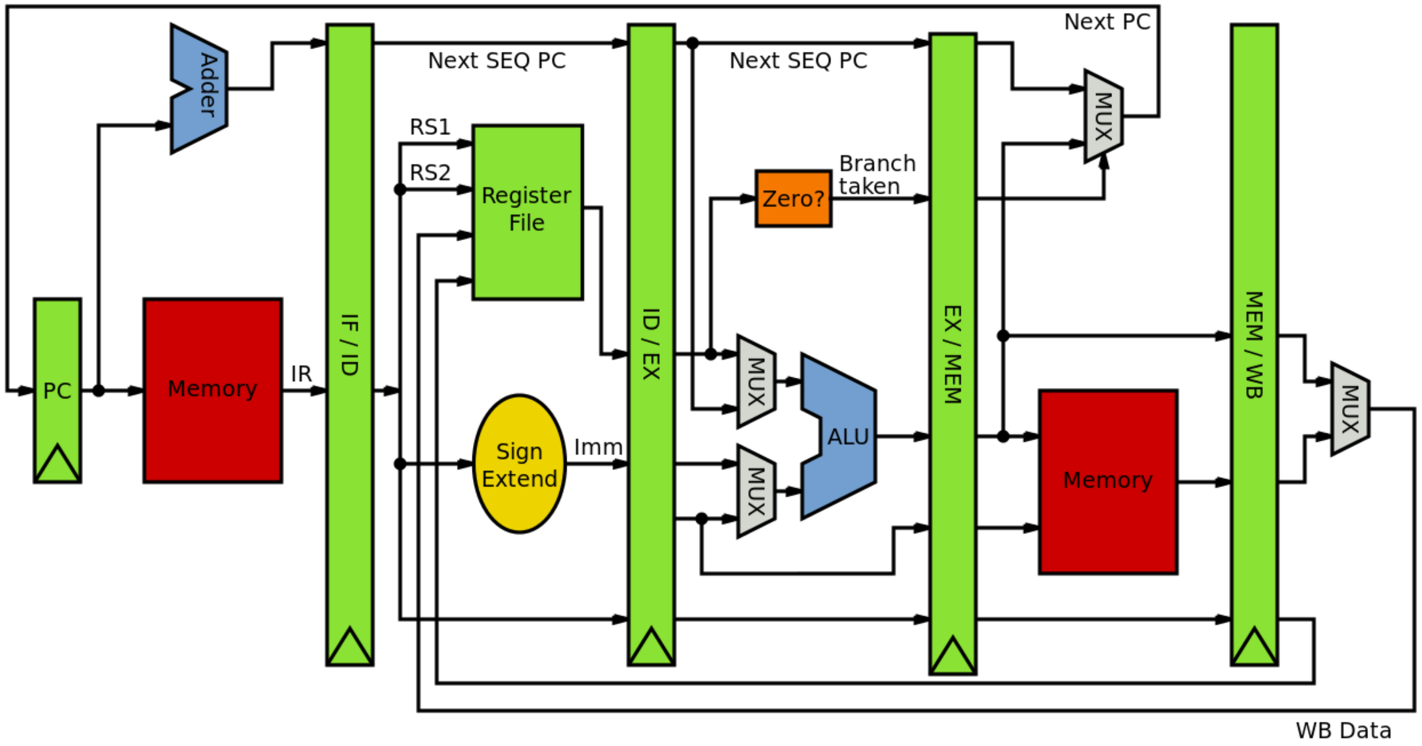
- Good for complex straight-line processes, eg image processing





Also good for visualizing hardware

Instruction Fetch	Instruction Decode Register Fetch	Execute Address Calc.	Memory Access	Write Back
IF	ID	EX	MEM	WB

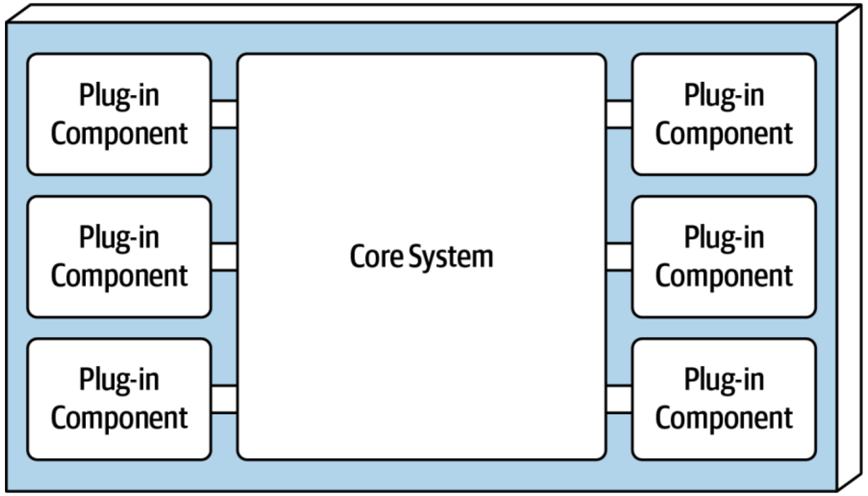


How do the stages communicate?

- That's the next-level decision
  - data-push (each stage invokes the next)
  - demand-pull (each stage demands data from its predecessor)
  - queues? buffers?
  - ??

Architecture #3: Plugins ("microkernel")

- System consists of a small core (the "microkernel") for essential functions, and lots of hooks for adding other services
- Highly extensible
- Plug-ins can be designed by small, less-experienced teams– even by users!
- Connection methods may vary



Plugin Examples

- Many examples:
  - Visual Studio Code (internal org. + extension marketplace)
  - emacs (emacs-lisp + hooks)
  - git clients

```
$ ls .git/hooks
applypatch-msg.sample       pre-applypatch.sample       pre-rebase.sample
commit-msg.sample           pre-commit.sample           pre-receive.sample
fsmonitor-watchman.sample   prepare-commit-msg.sample   update.sample
post-update.sample          pre-push.sample
```

Express.js uses a microkernel architecture

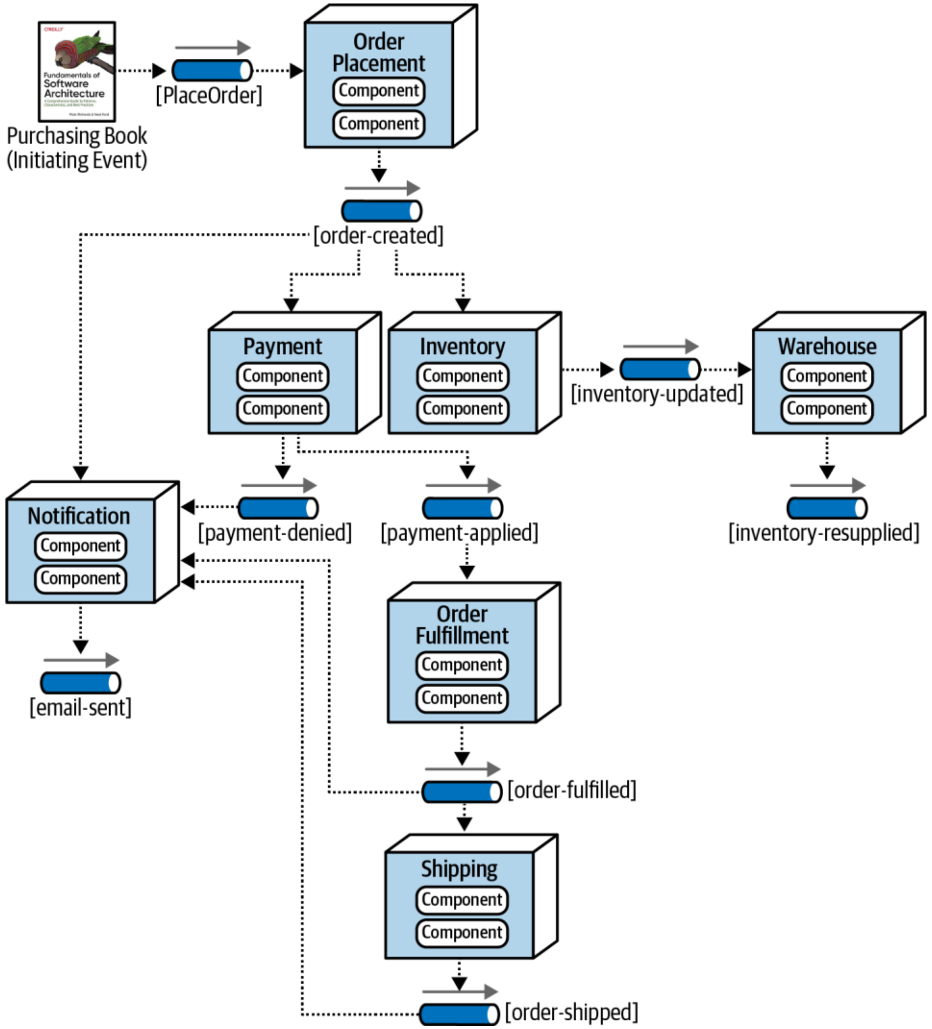
- express.js depends on plug-ins:

```
app.get('/transcripts', (req,res) => {
  console.log('Handling GET/transcripts')
  let data = db.getAll()
  console.log(data)
  res.status(200).send(data)
})
```

app.get is a hook that adds a handler to the server. The handlers are ordered (the first matching handler is executed), and can be pipelined, so a handler can invoke another handler if desired.

Architecture #4: Event-Driven Architecture

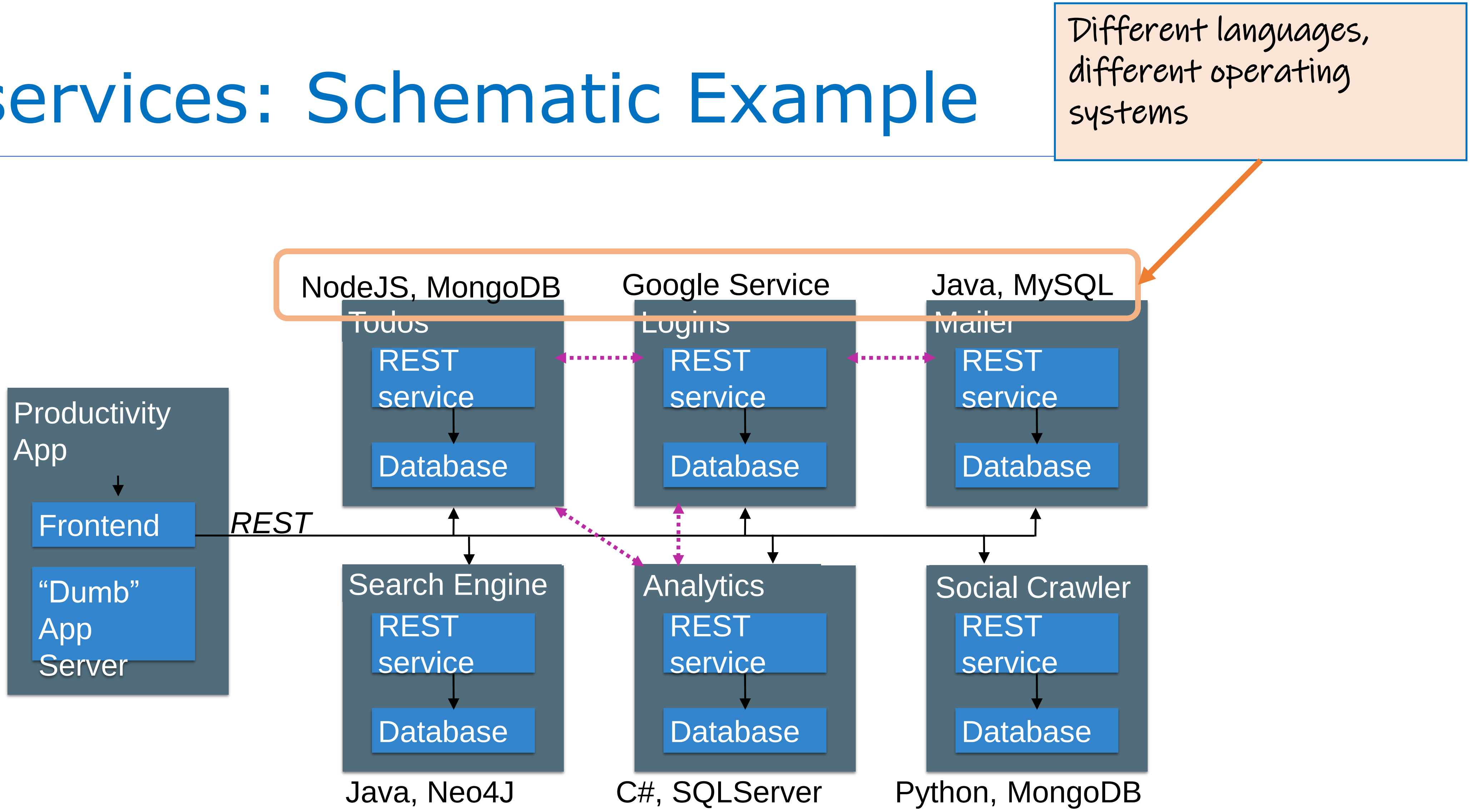
- Metaphor: a bunch of bureaucrats shuffling papers
- Each processing unit has an in-box and one or more out-boxes
- Each unit takes a task from its inbox, processes it, and puts the results in one or more outboxes.
- Stages are typically connected by asynchronous message queues.
- Conditional flow



Architecture #5: Microservices

- Overall task is divided into different components
- Each component is implemented independently
- Each component is
  - independently replaceable,
  - independently updatable
- Components can be built as libraries, but more usually as web services
  - Services communicate via HTTP, typically REST (see lesson 3.3)

Microservices: Schematic Example

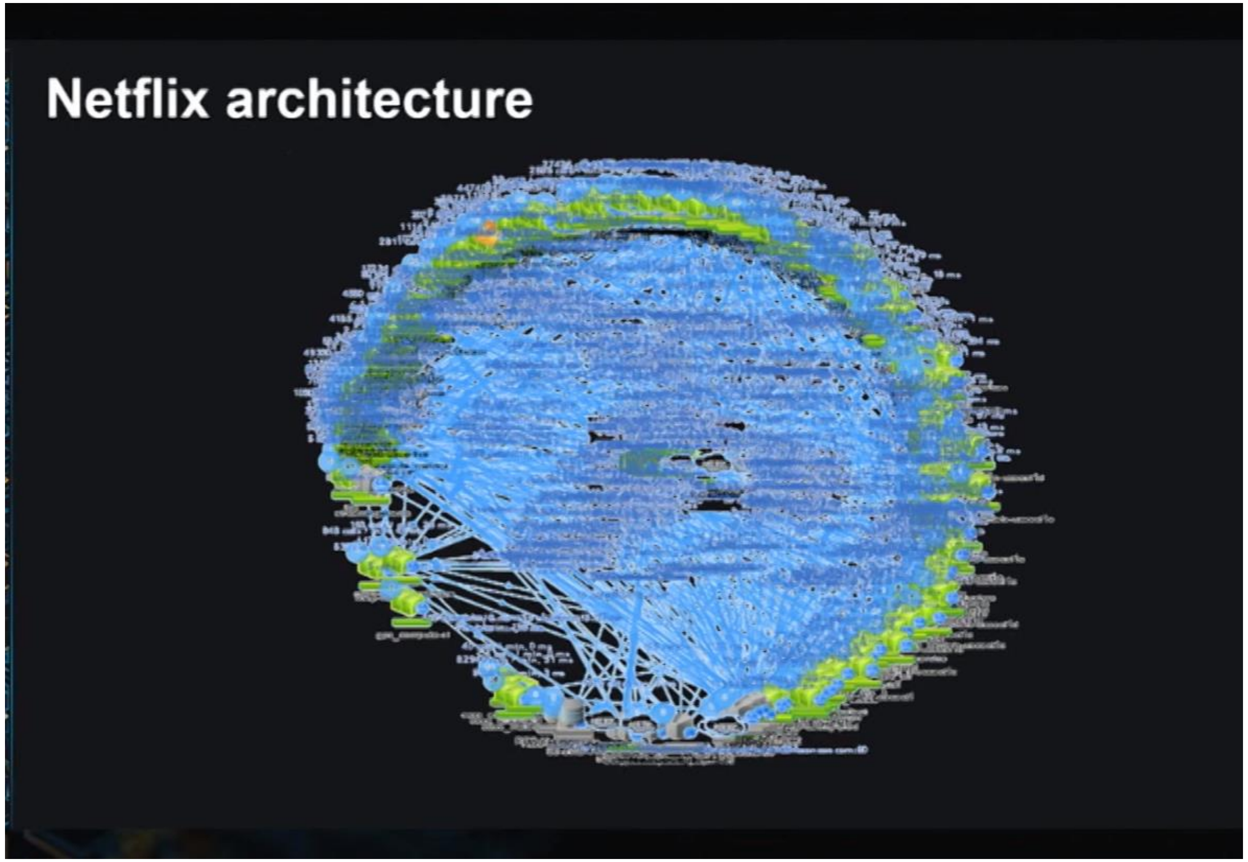


Microservice Advantages and Disadvantages

- Advantages
  - services may scale differently, so can be implemented on hardware appropriate for each (how much cpu, memory, disk, etc?). Ditto for software (OS, implementation language, etc.)
  - services are independent (yay for interfaces!) so can be developed and deployed independently
- Disadvantages
  - service discovery?
  - should services have some organization, or are they all equals?
  - overall system complexity

Microservices are (a) highly scalable and (b) trendy

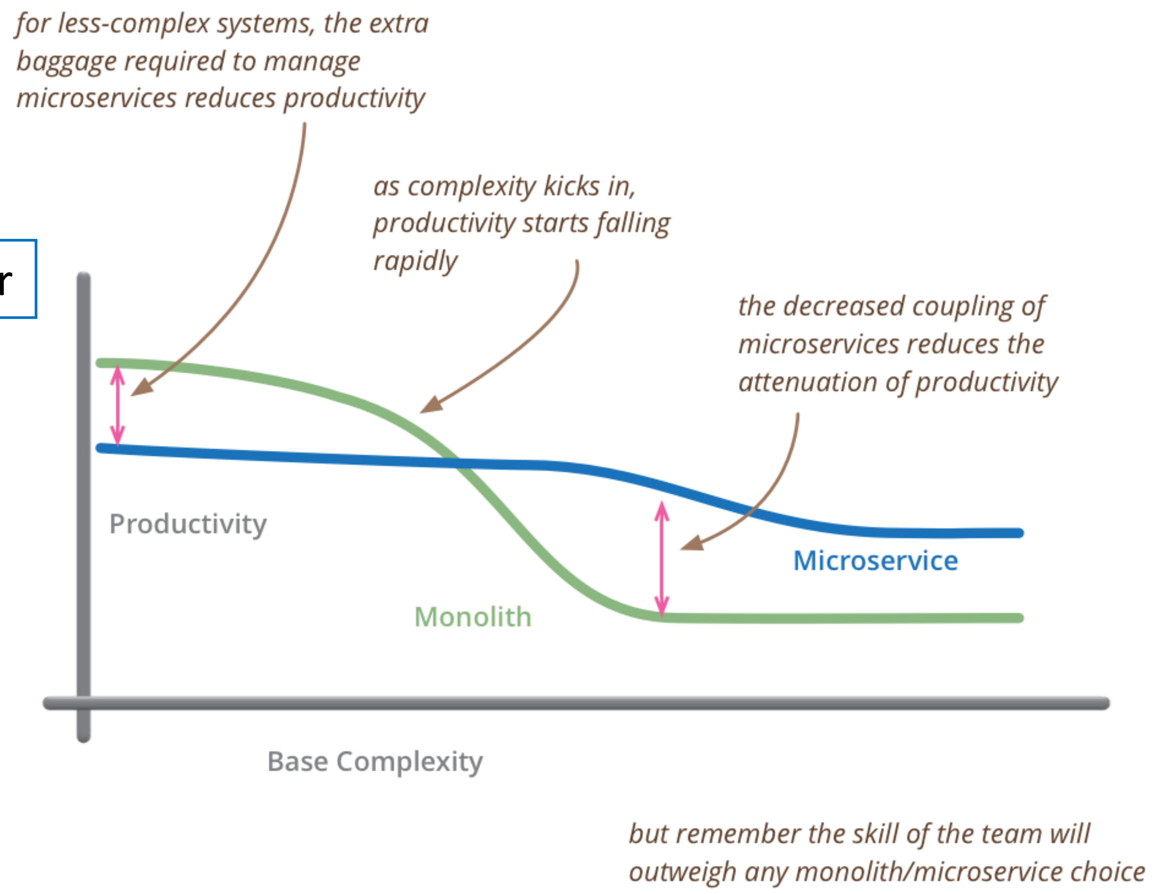
- Microservices at Netflix:
  - 100s of microservices
  - 1000s of daily production changes
  - 10,000s of instances
  - BUT:
  - only 10s of operations engineers



https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

Microservices vs Monoliths

higher is better



- Martin Fowler's Microservices Guide - https://martinfowler.com/microservices/

Review: Learning Objectives for this Lesson

- You should now be able to:
  - explain why software architecture is important
  - list a few of the properties that an architecture may have (the "ilities")
  - describe the basic ideas of the following architectures, with examples and pictures
    - monolithic
    - layered
    - pipeline
    - microkernel
    - event-driven
    - microservice

Next steps...

- In the remaining lessons of this week, we will learn about http, RESTful protocols, and express.js, with the goal of building a small but non-trivial REST server in express.js.