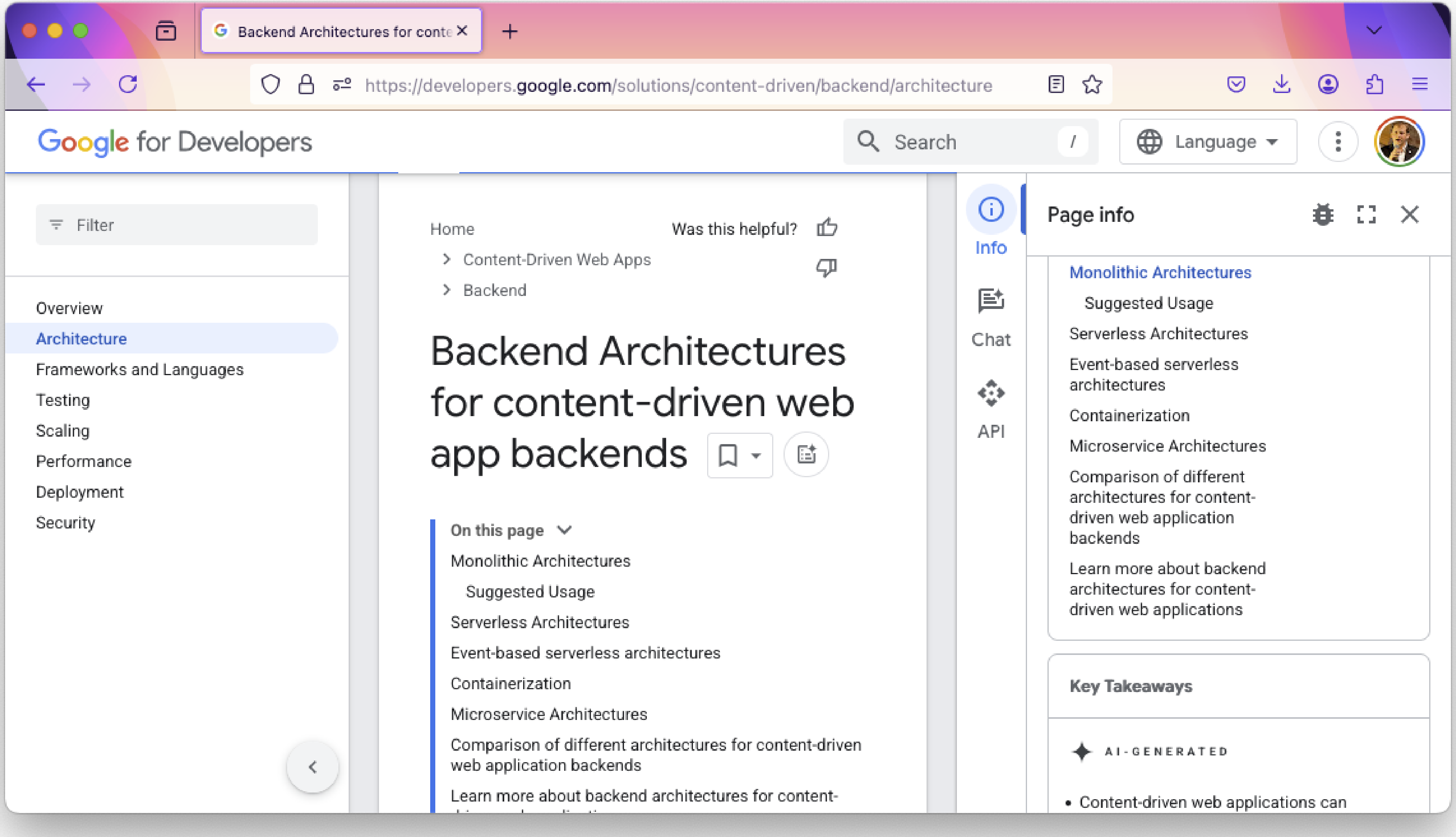
CS 4530: Fundamentals of Software Engineering
Module 3, Lesson 2
Architecting Simple Web Servers

Rob Simmons

Khoury College of Computer Sciences

???



Backend Architectures for conte

https://developers.google.com/solutions/content-driven/backend/architecture

Google for Developers

Search /

Language

Filter

- Overview
- Architecture
- Frameworks and Languages
- Testing
- Scaling
- Performance
- Deployment
- Security

Home > Content-Driven Web Apps > Backend

Was this helpful?

Backend Architectures for content-driven web app backends

On this page

- Monolithic Architectures
  - Suggested Usage
- Serverless Architectures
- Event-based serverless architectures
- Containerization
- Microservice Architectures
- Comparison of different architectures for content-driven web application backends
- Learn more about backend architectures for content-

Info

Chat

API

Page info

- Monolithic Architectures
  - Suggested Usage
- Serverless Architectures
- Event-based serverless architectures
- Containerization
- Microservice Architectures
- Comparison of different architectures for content-driven web application backends
- Learn more about backend architectures for content-driven web applications

Key Takeaways

AI-GENERATED

- Content-driven web applications can

Learning Goals for this Lesson

At the end of this lesson, you should be able to

- Explain what “business logic” is
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Explain the difference between “horizontal” and “vertical” scaling
- Know what someone is talking about when they say “microservices”

This example is silly

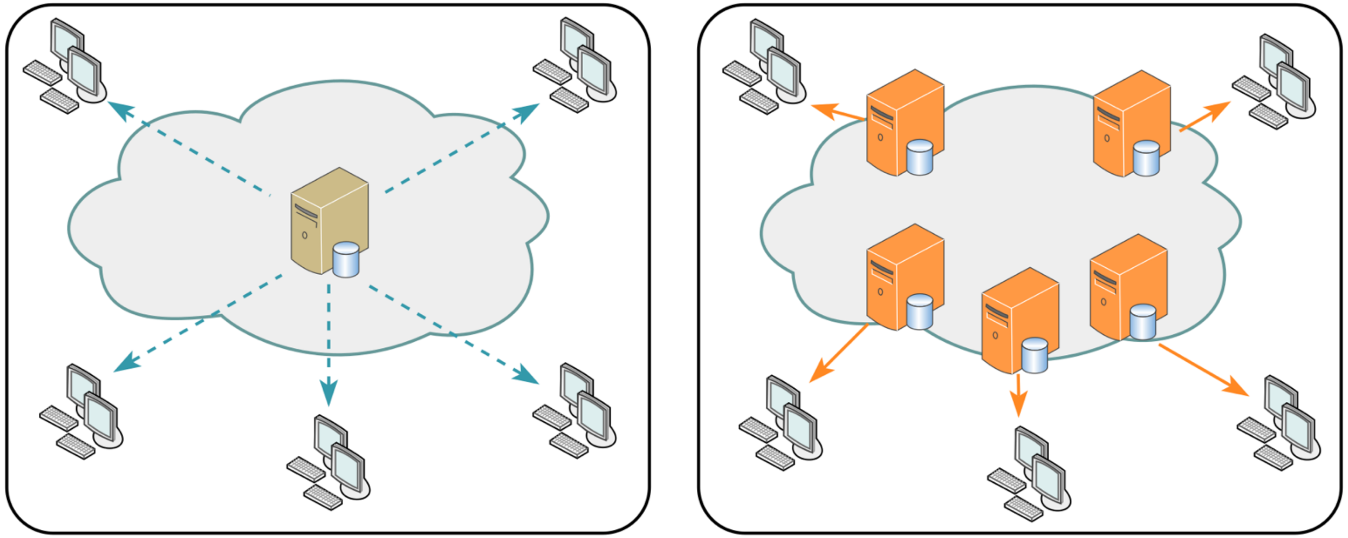
```
import express from 'express’;
import { z } from 'zod’;

type UserAuth = z.infer<typeof zUserAuth>;
const zUserAuth = z.object({
  username: z.string(),
  password: z.string(),
});
let numLogins = 0;
const app = express();
app.use(express.json());
app.post('/api/user/login', (request, response) => {
  const { username, password }: UserAuth = zUserAuth.parse(request.body);
  if (username.toLowerCase() === 'user1' && password === 'sekret') {
    response.send({ success: true, numLogins: numLogins++});
  } else {
    response.send({ error: 'Invalid username or password' });
  }
});
```

numLogins resets whenever you stop running the program

there’s one user and one password and it’s hard-coded

State and statelessness

- Web applications have *state*: they're ultimately storing or modifying *something*
  - Otherwise, maybe don't have a server running Node at all?
  - Content Delivery Networks have put tons of work into solving that distributed systems problem.
  - Static sites are fast & cheap



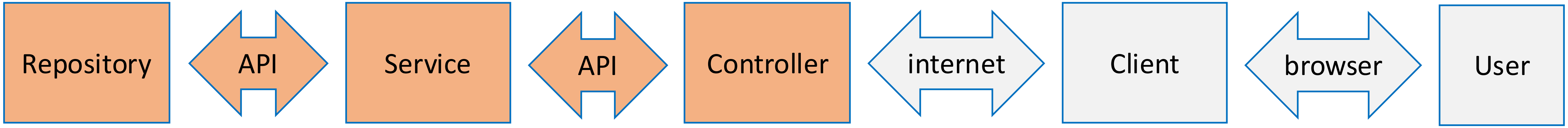
https://en.wikipedia.org/wiki/Content_delivery_network

State and statelessness

- A web server or web service should be *stateless*
  - Every REST request should be indifferent to whether the node application has been running for several hours or five seconds
  - Our silly application, and the IP1 code, is *not* stateless (why?)
- If the web server is going to be stateless, and the web application has state, the server has to phone a friend:
  - Access the filesystem
  - Query a database
  - Initiate some other remote procedure call to another server
- Common case: a *database* is the point of centralization
  - Centralization (& hierarchical centralization) is a cheat code for making distributed systems managable

Three parts of a web server

- The **repository** is the only part that stores state
  - I think it would be clearer if we called it the “database” tbh
- The **service** doesn’t know how we connect to the client
  - HTTP? REST? WebSockets? The service shouldn’t know!
- The **controller** doesn’t know how we store data
  - Are we actually stateless, or storing things in memory?
  - MongoDB? PostgresQL? SQLite? A file on the hard drive?



Service interface

```
import {
  StudentID,
  Student,
  Course,
  CourseGrade,
  Transcript,
} from './types.ts';
export interface StudentService {
  addStudent(studentName: string): Student;
  getTranscript(id: Student): Transcript;
  deleteStudent(id: Student): void;
  addGrade(id: Student, course: string, courseGrade: CourseGrade): void;
  getGrade(id: Student, course: string): CourseGrade;
  populateNames (studentName: string): Student[];
}
```

Service interface

- Everything we saw from the transcript server is the business logic — the most boring name possible for “the interesting stuff that a web server does that isn’t just reading from a database”
    - “Is this person an authenticated user?” — usually not business logic
    - “Does this user have permission to access student records” — business logic!
    - “Do new grades go at the front or back of the list” — business logic!

Testing

- We can test at both the service layer and the controller layer
  - What are the pros and cons of each?
- Sometimes we’ll want to test the service layer and/or controller layer without the repository layer!
  - We’ll come back to this.

Web Applications are Distributed Systems

Distributed systems are hard!

- Web applications are designed to only be *kinda* difficult-to-build distributed systems
- Most of this lecture is bad advice if you're Google, Netflix, or Amazon

Web applications are distributed systems *because*

1. You don't live in the cloud
2. **Scalability: Netflix needs at *least* two computers**

Scaling & the database bottleneck

- Web services often start on a single computer
- Stateless web servers make it possible to *horizontally* scale your web service as you get more users: add more cheap stateless web servers!
  - AWS will be delighted to help, only real limit is money
- Centralized databases tend towards *vertical* scaling: move your database to a more powerful computer
  - This has limits

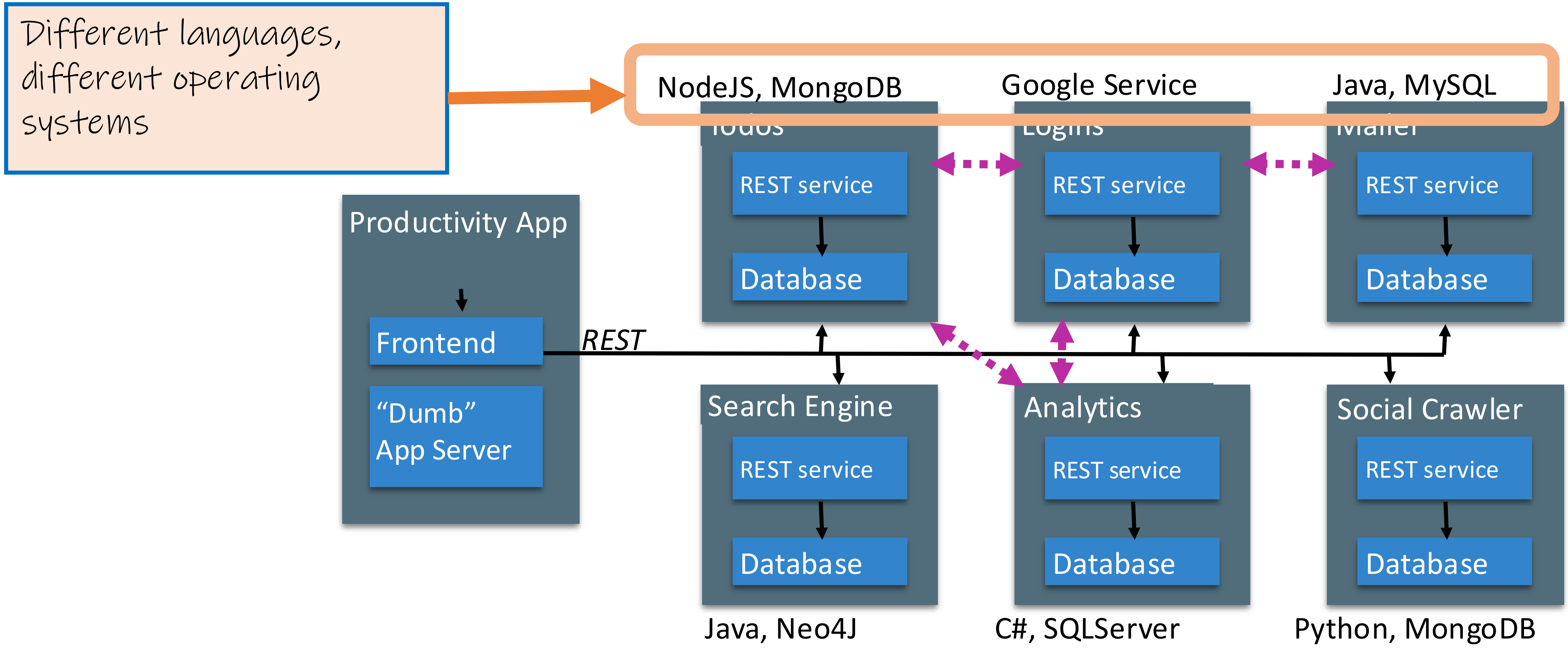
Scaling & the database bottleneck

- Most applications want to do expensive but periodic data analysis on the database
- Database *read-only-replicas* are an easy solution here — seconds to minutes behind reality (and can add reliability in case of failure!)

Scaling & the database bottleneck

- If you’ve got a bunch of data (or computation) that can handled separately and independently, you can put that somewhere else and have two independent databases
  - Chat and game information could be in separate places
  - Games could have their business logic running on different servers, written in different programming languages, and accessed (by the server the client is connected to) through their own REST API!
  - This way lies microservices

Microservices

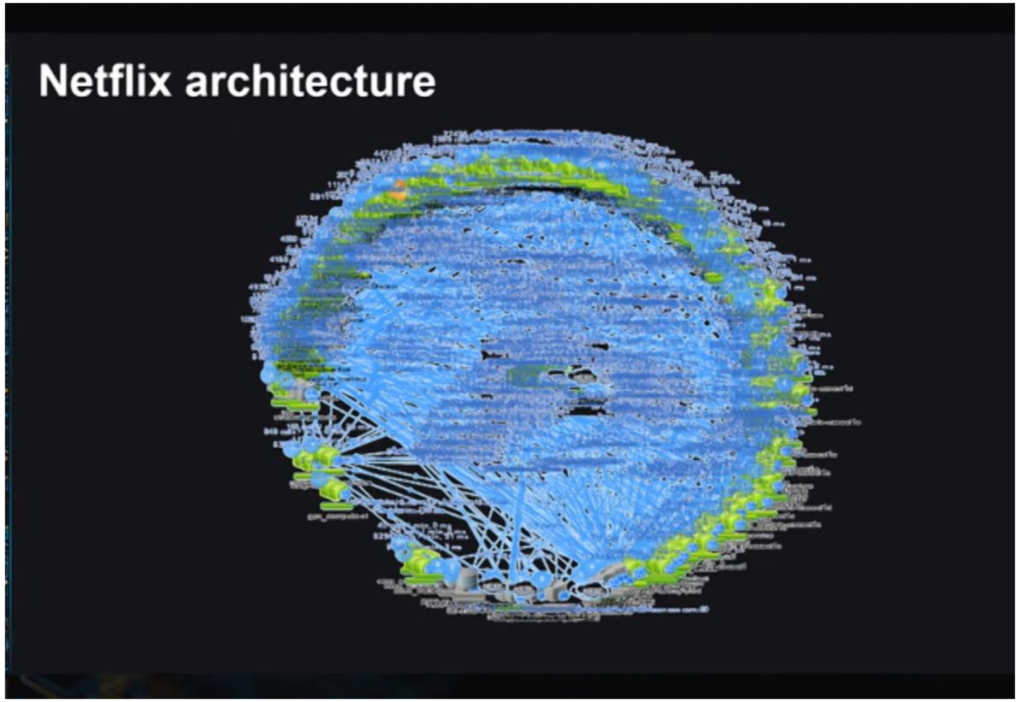


Different languages, different operating systems

Microservices

Netflix is the microservices darling

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

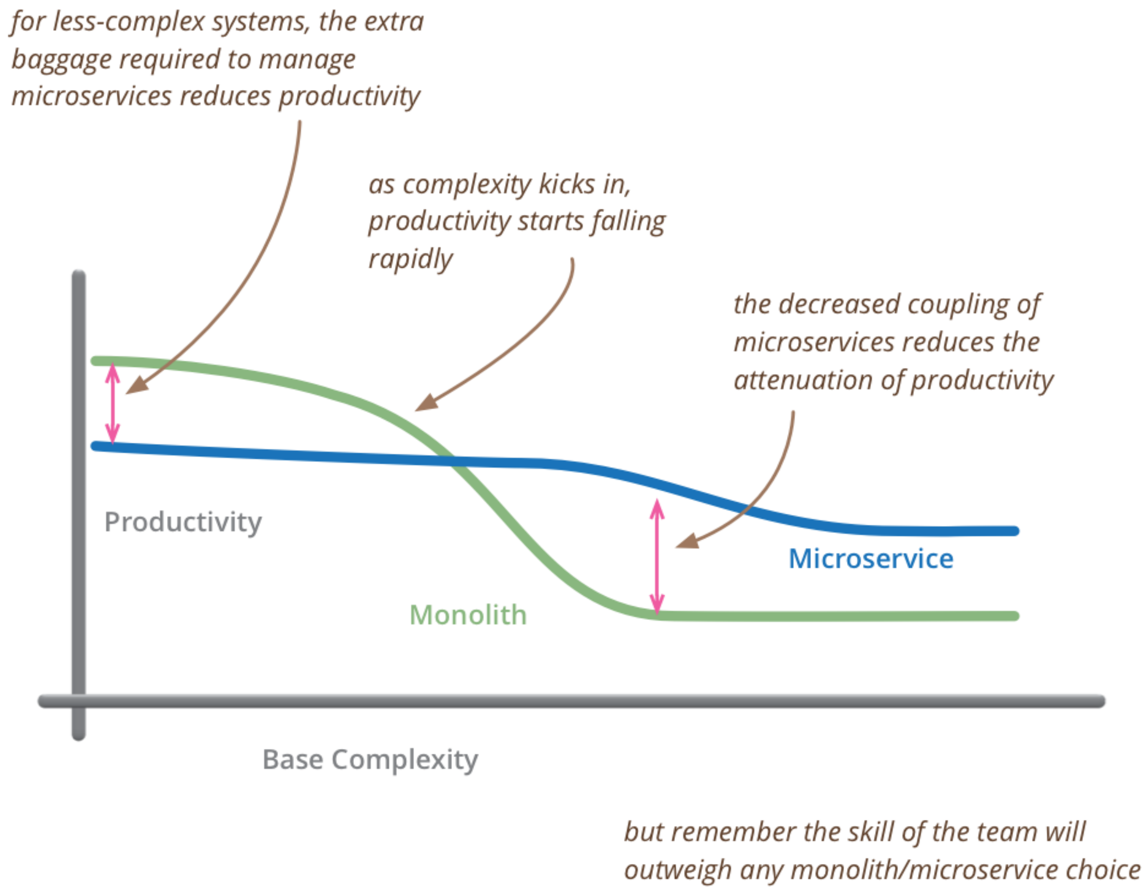


https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

Microservices

The opposite of “microservices” is “monolith”

higher is better



for less-complex systems, the extra baggage required to manage microservices reduces productivity

as complexity kicks in, productivity starts falling rapidly

the decreased coupling of microservices reduces the attenuation of productivity

but remember the skill of the team will outweigh any monolith/microservice choice

https://martinfowler.com/microservices/

Review

- Strategy.town is a monolithic application
- Personal self-assessment: I put a bit too much business logic in the controller layer (service layer doesn’t quite do enough)
- You’ll start IP2 with a proper repository
  - MongoDB is the database used for repository layer
  - Starter code *mostly* stateless, you’ll make it fully stateless
  - The controller doesn’t have to change!*

*we’ll talk about one very big exception tomorrow

Review

At the end of this lesson, you should be able to

- Explain what “business logic” is
- Describe the fundamental differences between the three layers of the controller, service, and repository layers in a C-S-R architecture
- Explain the difference between “horizontal” and “vertical” scaling
- Know what someone is talking about when they say “microservices”

Learning objectives for this lesson

- By the end of this lesson, you should be able to…
  - Explain what made single-threaded web servers an attractive alternative to connection-pool-based web servers
  - Identify a few pitfalls of writing single-threaded applications with cooperative concurrency
  - Understand the difference between programming with callbacks, “classic” promises, and async/await
  - Look at code diffs on GitHub and glean insights