Design and Modularity

Advanced Software Engineering
Spring 2023

Agenda

- Administrivia
  - Welcome to new students joining the class
  - Upcoming deadline: Feb 7, reflection paper proposal
  - Swapped mining software repositories and open source content
- Quality without a name
- (Some lecture on design)
- Reuse v compression
- (More lecture on design)

Why Design?

- Communicate organization: support human understanding of code
- Control complexity: maintainability, reusability
- Other reasons:
  - Efficiency (of implementation)
  - Have some artifact to check against requirements
  - Developer “happiness” (satisfies developer’s intrinsic motivation)
  - Extendability (add more features in future), interoperability (connect this to other things)
  - Learnability and understandability
  - Testability
- (Note - not all projects will require/be improved by these qualities)

The Quality Without a Name

Christopher Alexander

“The first place I think of when I try to tell someone about this quality is a corner of an English country garden where a peach tree grows against a wall.

The wall runs east to west; the peach tree grows flat against the southern side. The sun shines on the tree and, as it warms the bricks behind the tree, the warm bricks themselves warm the peaches on the tree. It has a slightly dozy quality. The tree, carefully tied to grow flat against the wall; warming the bricks; the peaches growing in the sun; the wild grass growing around the roots of the tree, in the angle where the earth and roots and wall all meet.

This quality is the most fundamental quality there is in anything.”

Image: DALL•E

QWAN and Software

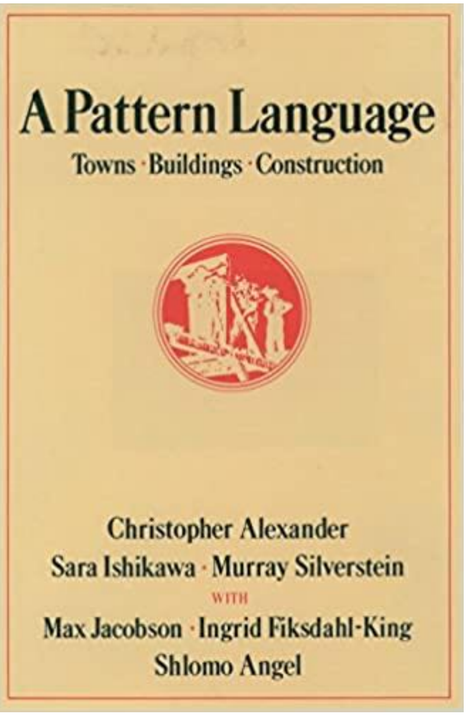
- How does RPG describe this? How would you?
- Have you worked with software in a way that resembles this?
- How do we make software in this manner?
- Is it possible to objectively define “good design” in software?

A Pattern Language: Towns, Buildings, Construction

Christopher Alexander (1977)

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

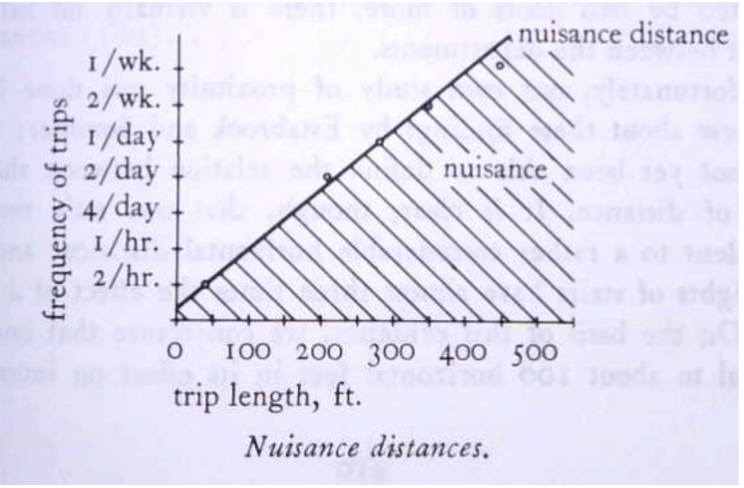
- It is a design language, constructed such that when patterns are shared by a neighborhood, the result is the QWAN
- Introduced this idea to a wide community beyond architects



Christopher Alexander's Patterns

No. 82: Office Connections

"If two parts of an office are too far apart, people will not move between them as often as they need to, and if they are more than one floor apart, there will be almost no communication between the two"



Nuisance distances.

"A Pattern Language: Towns, Buildings, Construction," C Alexander et al. pg 408-409

Example Pattern: Push vs Pull

```
class Producer {
    theData : number
}


class Consumer {
    neededData: number
    doSomeWork () {
        doSomething(this.neededData)
    }
}
```

- How can we get a piece of data from the producer to the consumer?

Pattern 1: consumer asks producer ("pull")

```
class Producer {
    theData : number
    getData () {return this.theData}
}

class Consumer {
    constructor(private src: Producer) { }
    neededData: number
    doSomeWork() {
        this.neededData = this.src.getData()
        doSomething(this.neededData)
    }
}
```

- The consumer knows about the producer
- The producer has a method that the consumer can call
- The consumer asks the producer for the data

Pattern 2: producer tells consumer ("push")

```
class Producer {
    constructor(private target : consumer) {}
    theData : number
    updateData (input) {
        // ..something that changes theData..
        // notify the consumer about the change:
        this.target.notify(this.theData)
    }
}

class Consumer {
    neededData: number
    notify(val: number) { this.neededData = val }
    doSomeWork () {
        doSomething(this.neededData)
    }
}
```

- Producer knows the identity of the consumer
- The Consumer has a method that producer can use to notify it.
- Producer notifies the consumer whenever the data is updated
- Probably there will be more than one consumer

This is called the Observer Pattern

- Also called "publish-subscribe pattern"
- Also called "listener pattern"
- The object being observed (the "subject") keeps a list of the objects who need to be notified when something changes.
  - subject = producer = publisher
- When a new object wants to be notified when the subject changes, it registers with ("subscribes to") with the subject/producer/publisher
  - observer = consumer = subscriber = listener

Push vs. Pull: Tradeoffs

PULL	PUSH
The Consumer knows about the Producer	Producer knows about the Consumer(s)
The Producer must have a method that the Consumer can call	The Consumer must have a method that producer can use to notify it
The Consumer asks the Producer for the data	Producer notifies the Consumer whenever the data is updated
Better when updates are more frequent than requests	Better when updates are rarer than requests

Design Patterns are Everywhere

Example: Alexander's "Oregon Experiment"

Every particular community will always need to do the same to supplement the general patterns from *A Pattern Language*. The patterns are:

UNIVERSITY POPULATION
OPEN UNIVERSITY
STUDENT HOUSING DISTRIBUTION
UNIVERSITY SHAPE AND DIAMETER
UNIVERSITY STREETS
LIVING LEARNING CIRCLE
FABRIC OF DEPARTMENTS
DEPARTMENTS OF 400
DEPARTMENT SPACE
LOCAL ADMINISTRATION
STUDENT COMMUNITY
SMALL STUDENT UNIONS
PARKING SPACES
CLASSROOM DISTRIBUTION
FACULTY STUDENT MIX
STUDENT WORKPLACE
REAL LEARNING IN CAFES
DEPARTMENT HEARTH

9. Living Learning Circle:
Students who want to live closely related to the university want their housing integrated with the university; yet most on-campus housing provided today is zoned off from academic departments. Therefore: Provide housing for 25 per cent of the student population within the 3000 for inner university diameter. Do not zone this housing off from academic departments…"

"The Oregon Experiment," C Alexander et al. pg 114

Design Patterns are Everywhere

- Every time you read a blog post or web page with some code illustrations, you are using a design pattern:
  - a piece of code to solve a particular problem
  - and which needs to be adapted to your particular situation.
- Four guys in the 90’s wrote a book that lists a lot of patterns, including the “observer” pattern

Christopher Alexander on OOD Patterns

Keynote address at OOPSLA, 1996

> “So far, as a lay person trying to read some of the works that have been published by you in this field, it looks to me more as though mainly the pattern concept, for you, is an inspiring format that is a good way of exchanging fragmentary, atomic, ideas about programming. Indeed, as I understand it, that part is working very well. But these other two dimensions, (1) the moral capacity to produce a living structure and (2) the generativity of the thing, its capability of producing coherent wholes—I haven't seen very much evidence of those two things in software pattern theory. Are these your shortcomings? Or is it just because I don't know how to read the literature?”

Talk: https://www.youtube.com/watch?v=98LdFA-_zfA
Transcript: http://www.patternlanguage.com/archive/ieee.html
Image: DALL•E

Show Trial Of The Gang Of Four (A joke?)

“Patterns once held out the promise of freeing object-oriented architects, designers, and programmers from the oppression of the cartoon/CargoCultists and methodology mongers that theretofore held sway over them. They aimed to help fulfill the elusive promise of object-oriented reuse, and help designers reap the benefits of an undiscovered bounty of collective object-oriented architectural experience. Have patterns come close to achieving these utopian goals, or are Vlissides, Johnson, Helm, and Gamma the GoF that failed?”

“The Accused have promoted a cult of personality, and brought about the establishment of a cottage industry of consultants, trainers, and sundry acolytes to interpret their abstruse musings.

The Accused, by distilling hard-won design expertise into patterns, have encouraged novices to act like experts.”

https://wiki.c2.com/?ShowTrialOfTheGangOfFour

Example Pattern: Circuit Breaker

- Context: Applications need to connect to remote APIs to access data. Sometimes there are connection problems.
- Problem to solve: When there is a connection problem, making more connections only makes matters worse
- Pattern: Use a “circuit breaker” that, after 3 failures, will fail-fast

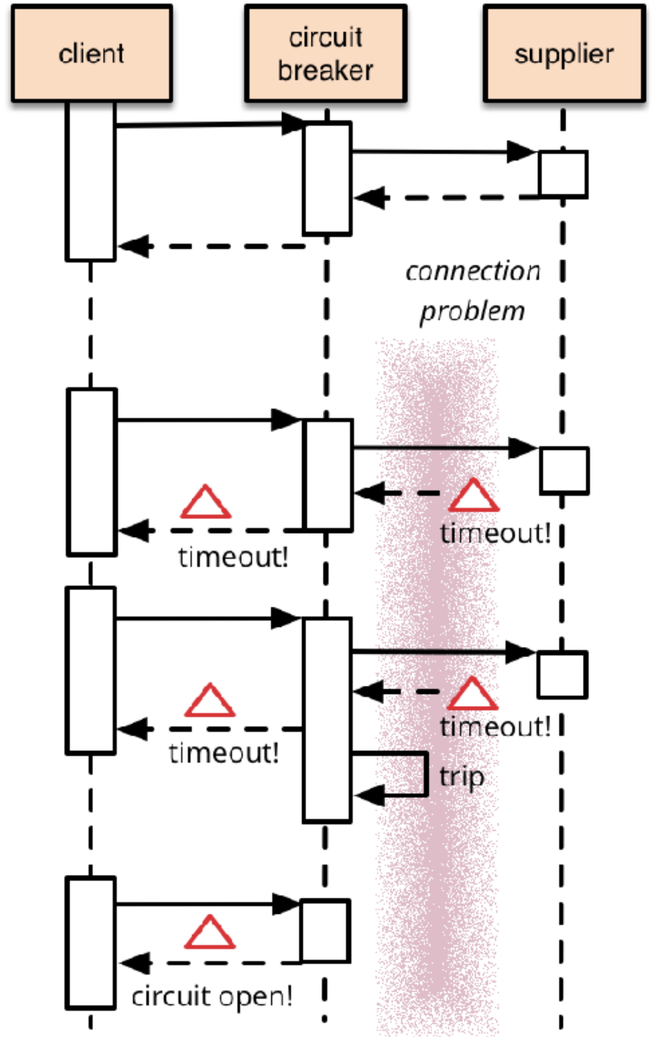
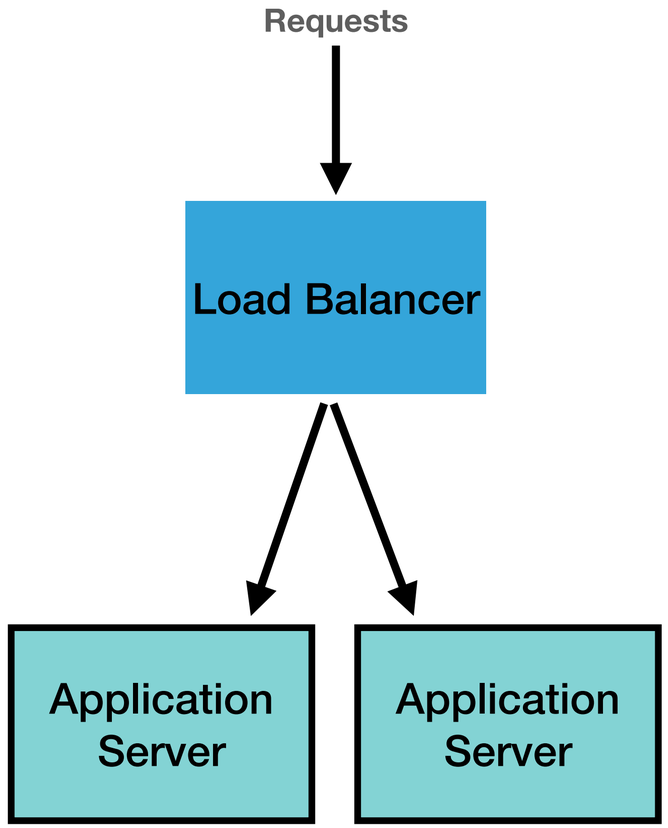


Figure: Martin Fowler

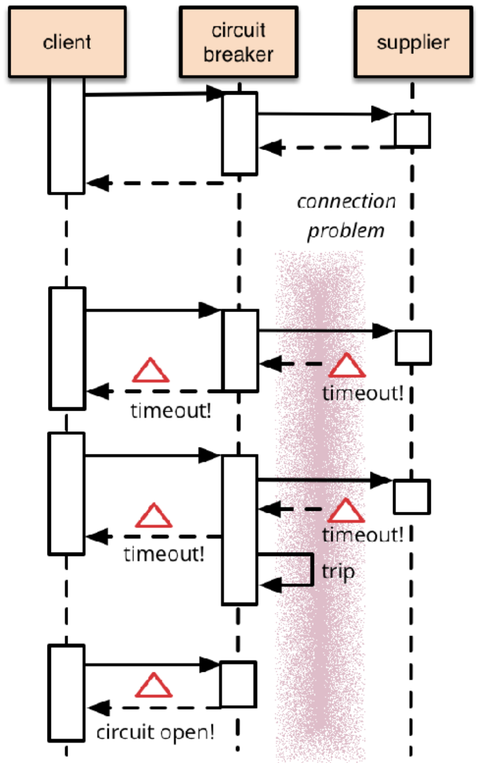
Example Pattern: Horizontal Scaling

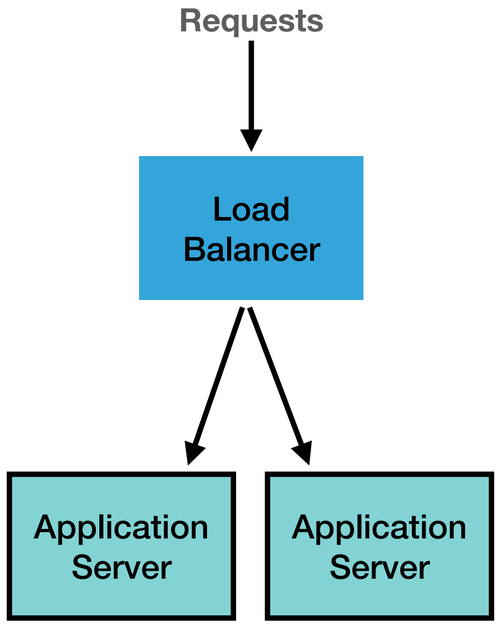
- Context: An application can only process so many requests per-second. Each request is stateless (independent from each other).
- Problem to solve: when there are too many requests for one server to process, we still need to be able to process them
- Solution: Deploy multiple servers, using a load balancer to route requests in a round-robin manner
- See also: Stateful load balancing; canary deployments



Patterns inspire modularity

- Why modularity?
  - Reuse… or… compression?
- How to define module boundaries?
  - Experience… or… Patterns?
- Common problems + common solutions = infrastructure?
  - Database, load balancer, cache, logger, authentication/authorization, message queues, etc…



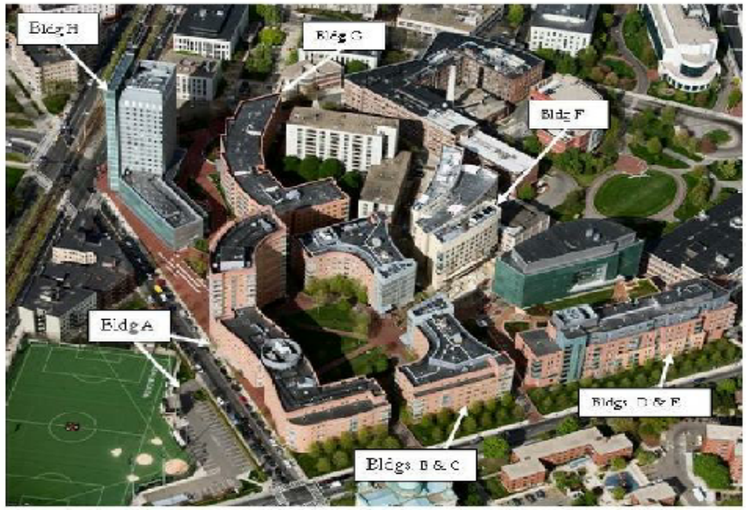


Compression and Reuse

- In what ways *can* we reuse code?
  - Object-oriented
  - Modules
  - Libraries
  - …?
- Why compare reuse and compression?
- How to design abstractions? What are the desirable qualities for abstractions?

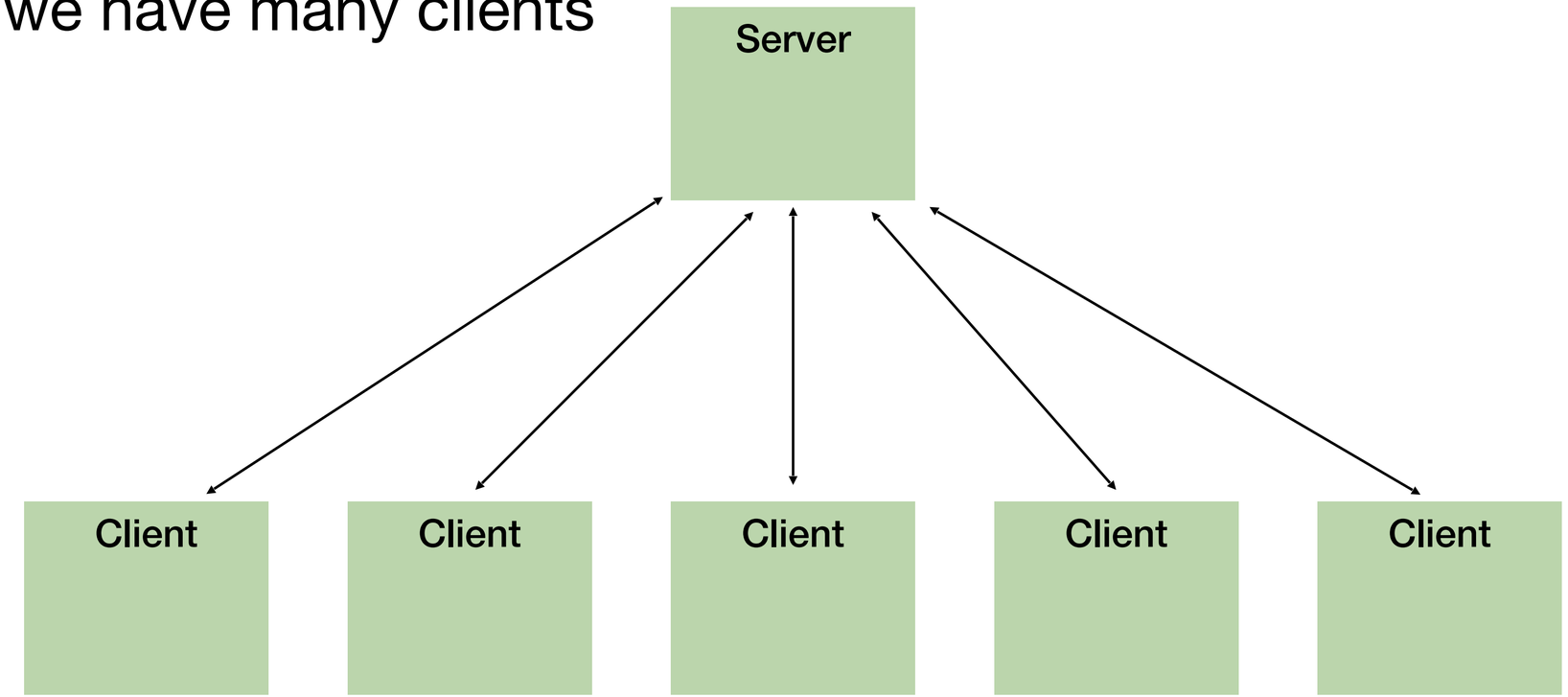
Distributed Software Architectural Patterns

- Goal: abstract details away into reusable components
- Enables exploration of design alternatives
- Allows for analysis of high-level design before implementation
- Match system requirements to quality attributes of common architectural patterns



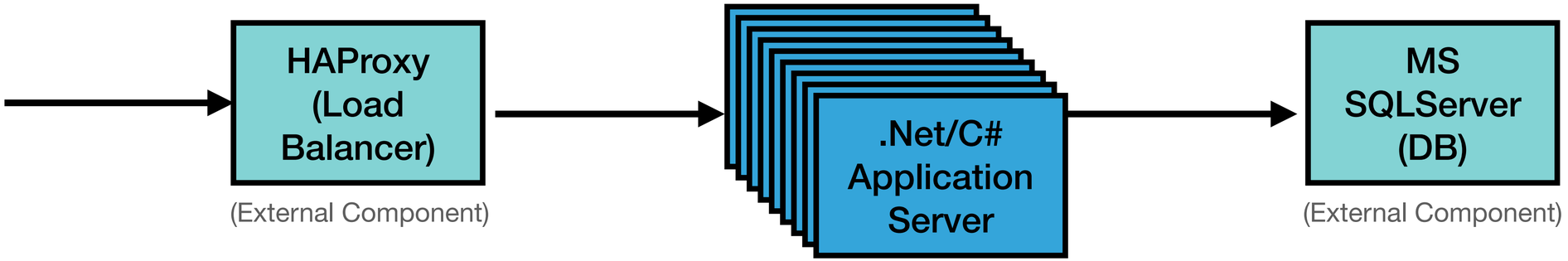
The Monolith Relies on a Single Server

- Simplest answer to consistency problem: have only one server, one source of truth
- Still “distributed” in that we have many clients
- Sacrifices:
  - Scalability
  - Performance
  - Fault tolerance



Stack Exchange is a Monolithic Application

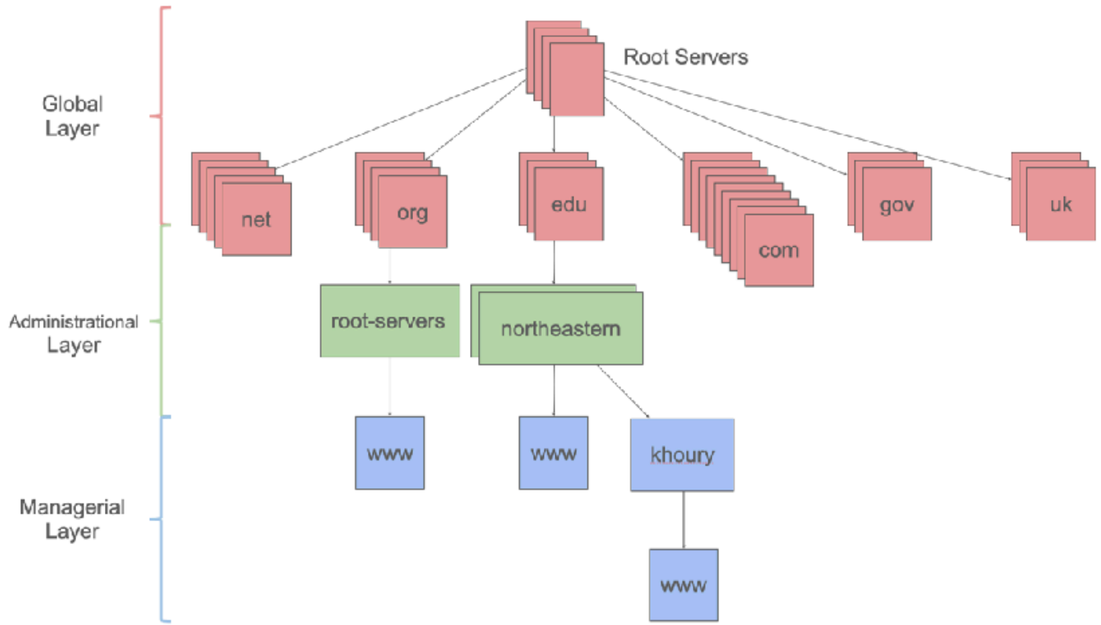
(Stack Overflow, etc)



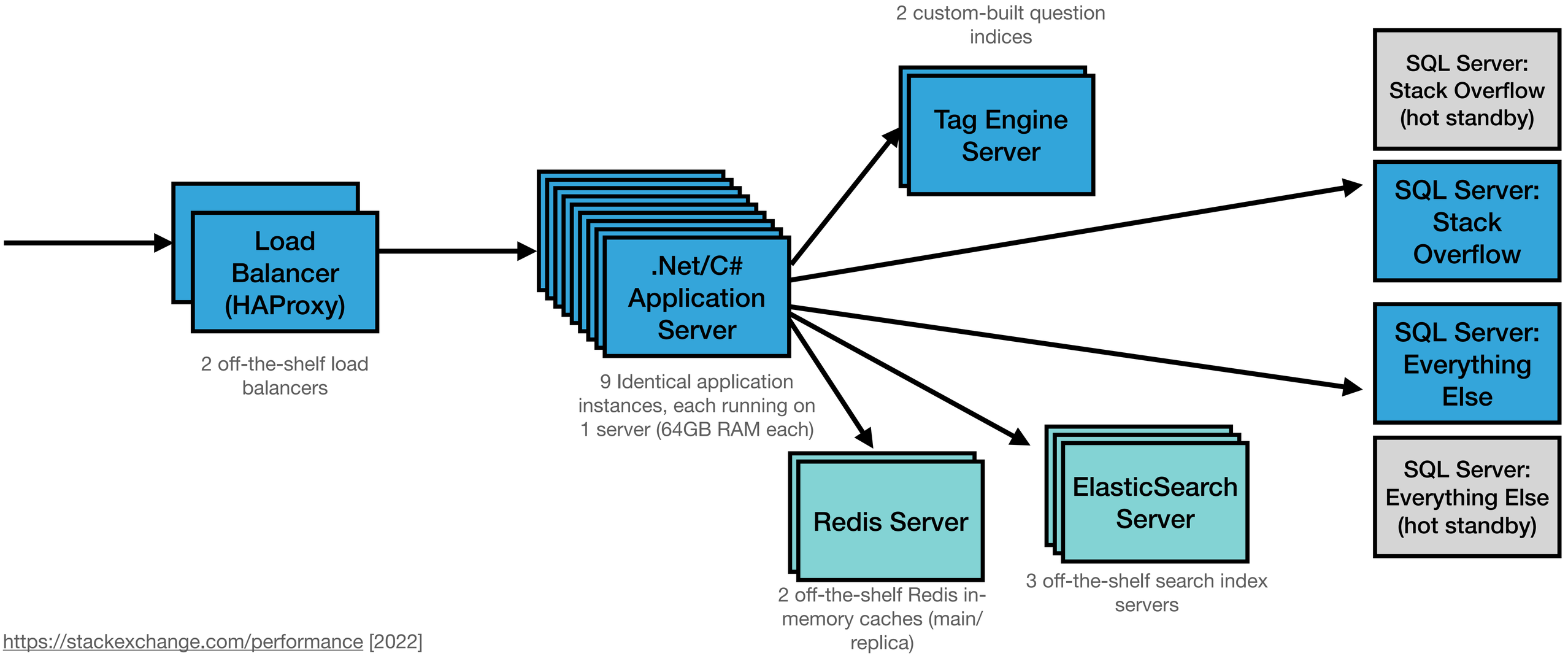
https://stackexchange.com/performance [2022]
https://hanselminutes.com/847/engineering-stack-overflow-with-roberta-arcoverde

Tiered Architectures Partition Responsibility

- Key idea: Partition the system into distinct tiers based on responsibilities
- Each tier scales independently of the others - .com need not know about .org
- Satisfying a single request may require multiple tiers
- DNS is a tiered architecture
  - Example: scale .com differently from .gov



Stack Overflow is Really a Tiered Monolith



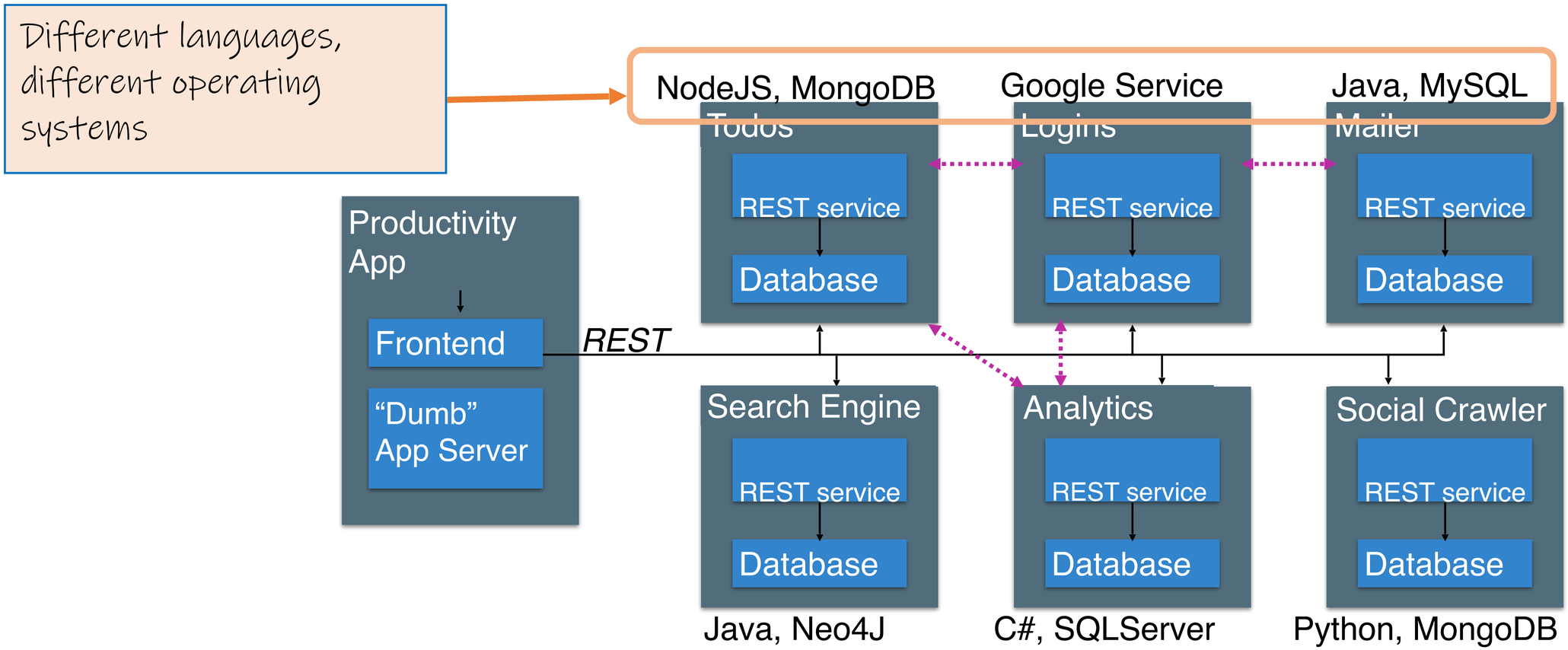
https://stackexchange.com/performance [2022]
https://hanselminutes.com/847/engineering-stack-overflow-with-roberta-arcoverde

Microservice Architectures

(Or: service-oriented architectures)

- Organize implementation around components (responsibilities)
- 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 well-defined protocol like REST

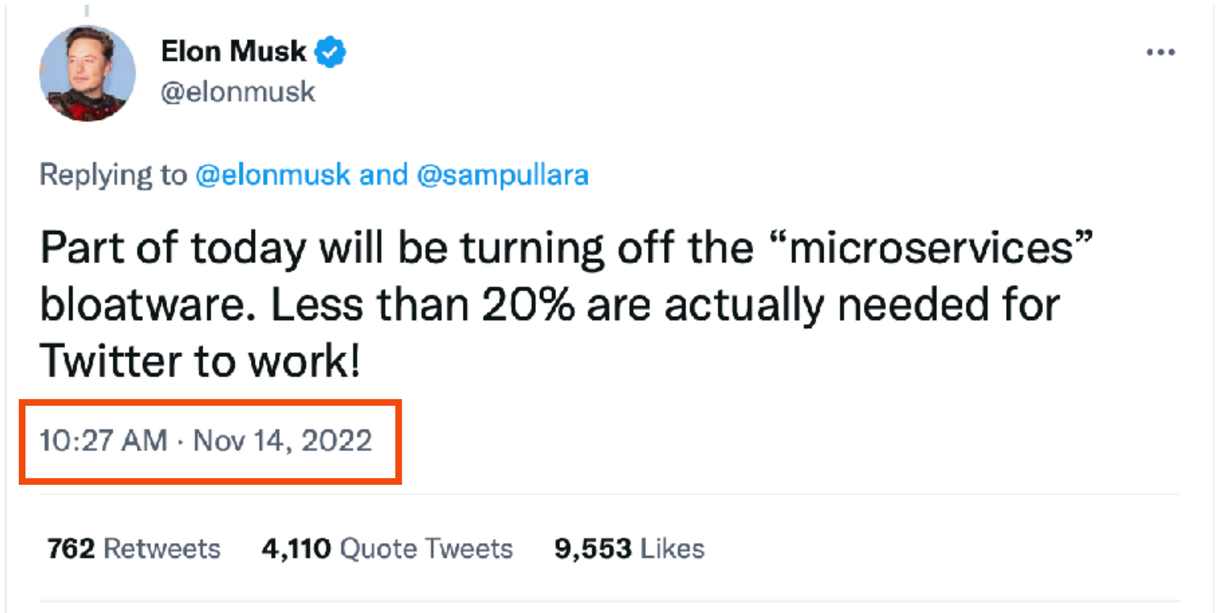
Microservices: Schematic Example



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… developer environments/experience?

Microservices and Twitter

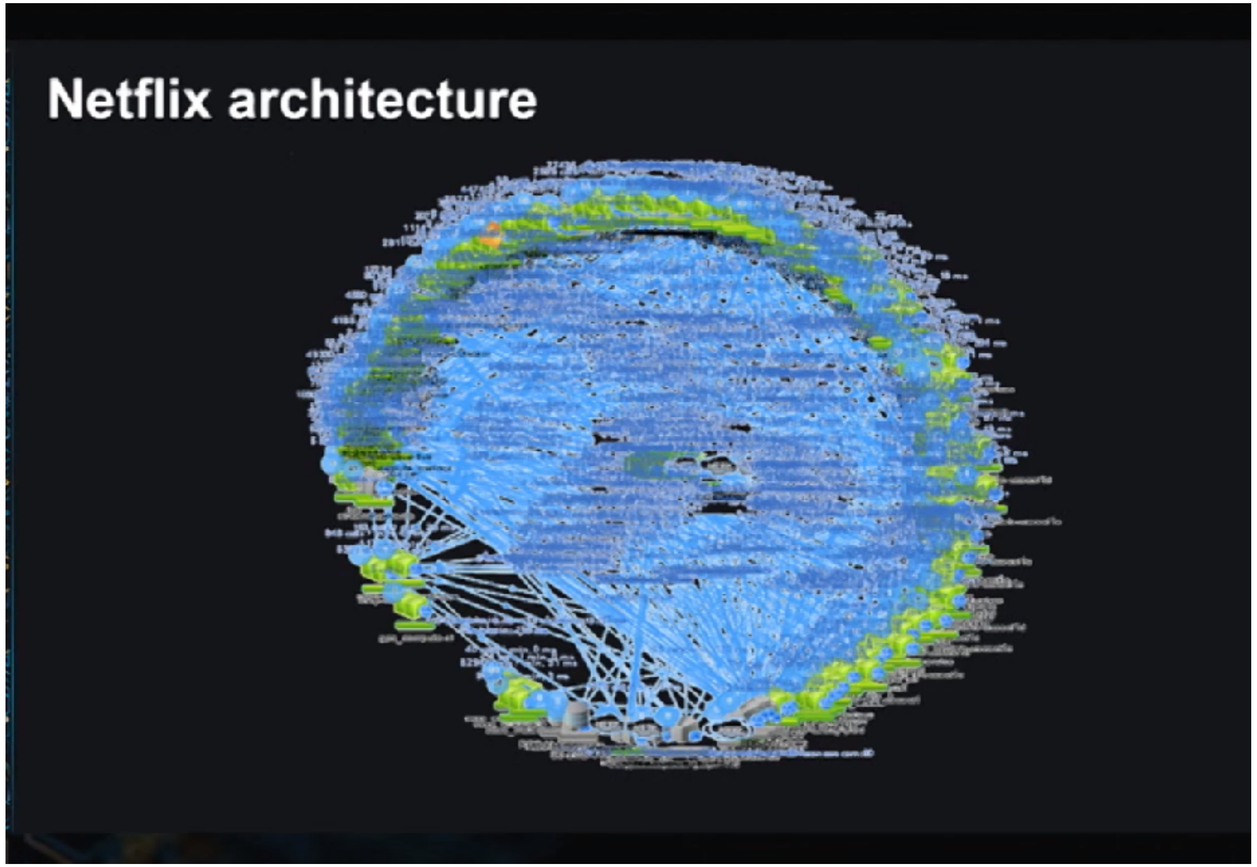


https://twitter.com/elonmusk/status/1592177471654604800

https://www.theverge.com/2022/11/15/23460940/twitter-2fa-two-factor-authentication-microservices

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

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