# Tech Debt A Master Class

@r0ml

## Outline

- What is "Technical Debt"?
- avoiding it
- implications of "microservices"
- implications of "refactoring"

## What is "Technical Debt"?









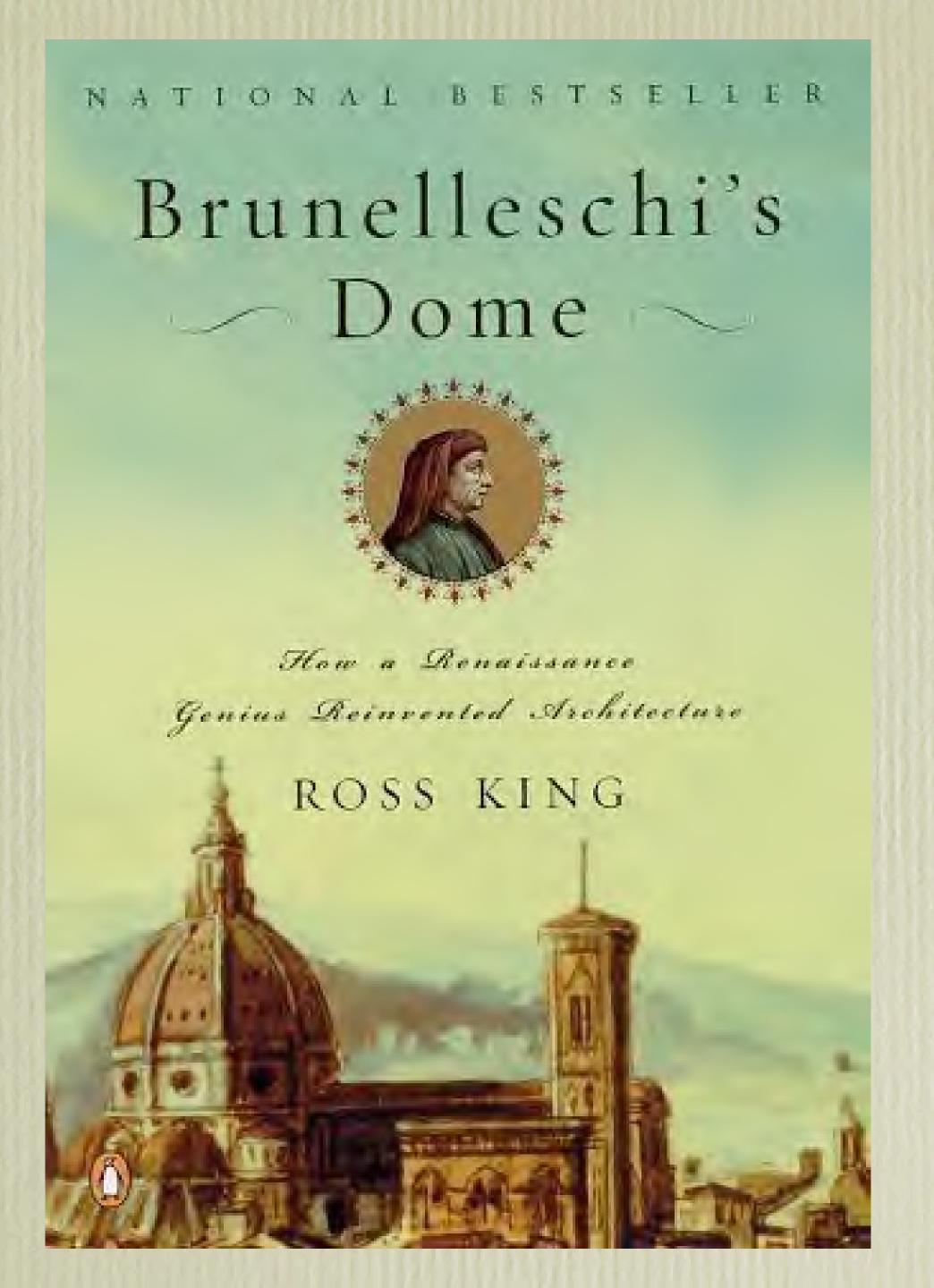
#### Design-Build Delivers: The New Tappan Zee Bridge

You don't have to be a New Yorker to appreciate the engineering, design, construction and political marvel that is the new Tappan Zee bridge. Dubbed the "\$4 Billion Marvel" by The New York Times, the impressive bridge project hosts its first drivers this weekend: "The new bridge is a symphony of statistics. More than 1,000 cylindrical piles were planted into the Hudson riverbed to create 41 pillars to hold up each span. Some 330,000 cubic yards of concrete were poured in construction, including some 6,000 precast concrete road deck panels. The main decks of both spans are held up by 192 cables stretched among eight 419-foot angled towers. The bridge will handle 140,000 cars a day."

None of this could have happened without design-build.

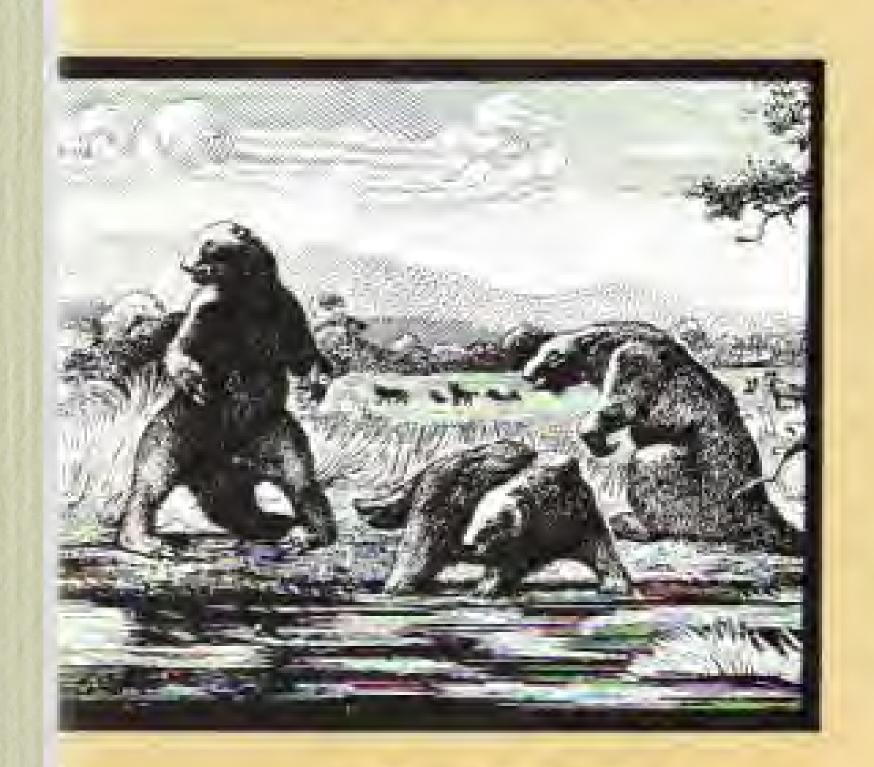
# Design-Build vs. Design-Bid-Build

- https://en.wikipedia.org/wiki/Design-build
- Until 1979, the AIA American Institute of Architects' code of ethics and professional conduct prohibited their members from providing construction services.
- In 1993, the Design-Build Institute of America (DBIA)[4] was formed.
- Design—build is sometimes compared to the "master builder" approach, one of the oldest forms of construction procedure. Comparing design—build to the traditional method of procurement, the authors of Design-build Contracting Handbook noted that: "from a historical perspective the so-called traditional approach is actually a very recent concept, **only being in use approximately 150 years**. In contrast, the design—build concept—also known as the "master builder" concept—has been reported as being in use for over four millennia."[2]



## the mythical man-month

Essays on Software Engineering



Frederick P. Brooks, Jr.

## Mythical Man Month

To make a user-friendly system, the system must have conceptual integrity, which can only be achieved by separating architecture from implementation. A single *chief architect* (or a small number of architects), acting on the user's behalf, decides what goes in the system and what stays out.

# Lowering Software Development Costs

- Implementers may be hired only after the architecture of the system has been completed (a step that may take several months, during which time prematurely-hired implementers may have nothing to do).
- Do not develop software. Buy it "off the shelf" instead.

# https://en.wikipedia.org/wiki/ Software\_architect

The software architect concept began to take hold when object-oriented programming or OOP, was coming into more widespread use (in the late 1990s and early years of the 21st century).[citation needed] OOP allowed ever-larger and more complex applications to be built, which in turn required increased high-level application and system oversight.

# this slide intentionally left blank



#### awardcunningham

#### Ward Cunningham

## 1992

"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. *Objects make the cost of this transaction tolerable.* The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."



#### @martinfowler

#### Martin Fowler

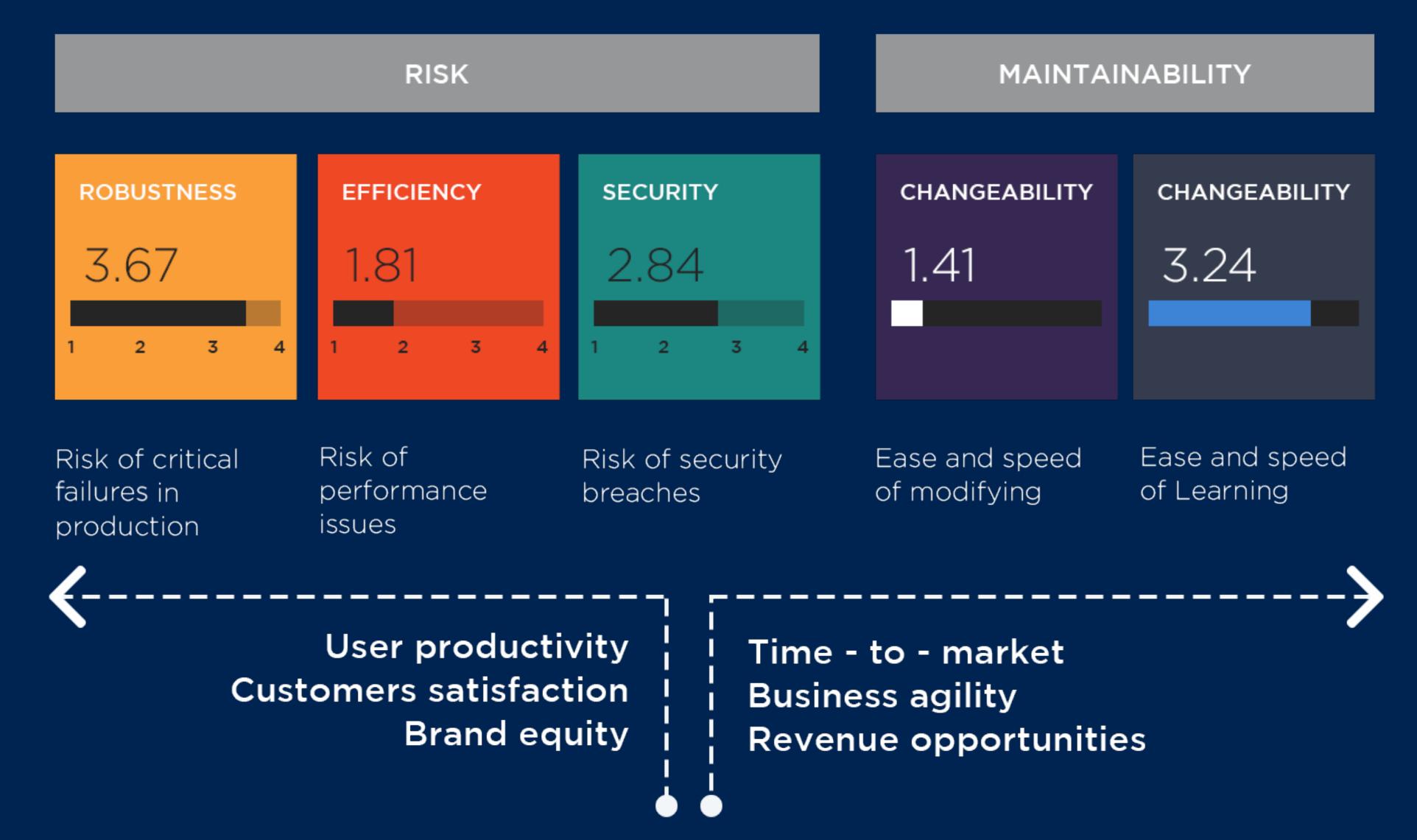
2003

The tricky thing about technical debt, of course, is that unlike money it's impossible to measure effectively. The interest payments hurt a team's productivity, but since we CannotMeasureProductivity, we can't really see the true effect of our technical debt.



# Why does this metaphor fall short?

- "managers don't care about technical debt interest payments."
- it should be about "risk", not "debt"



http://www.castsoftware.com



#### @ralfw

### There is no such thing as technical Debt

- do you know when you got into debt?
- did you ask someone for a loan?
- do you know how much you owe?
- do you know what your payments are?
- who do you owe?
- is anybody threatening you for failure to pay back the loan?

We could summarize the metaphor as follows:

When *taking short cuts* and *delivering code that is not quite right* for the programming task of the moment, a development team incurs Technical Debt. This debt decreases productivity. This loss of productivity is the interest of the Technical Debt.

https://www.agilealliance.org/introduction-to-the-technical-debt-concept/

## What is Technical Debt?

- There are errors of omission and commission.
- An error of omission is not tech debt it is YAGNI. It is MVP. It is narrowing scope in order to release sooner.
- Tech debt is an error of commission. Tech debt is when you put in code you shouldn't have.
- "Creating debt" is called "borrowing"
- The most common error of commission is adding a dependency. When you add a dependency, you are *borrowing* code. You now have a tech debt.
- That's why the fix is "refactoring" and not "implementing that which is missing"

# What gets borrowed?

- The tech that is already there.
- The tech that is "free".

## The tech that is there

- CoffeeScript or JQuery
- PHP or Python
- MySQL or PostgreSQL
- Consul
- Redis
- RedHat or Ubuntu
- Rackspace or AWS

## The tech that is free

- Kafka
- Elm
- Node
- Pillow

## How to Avoid It

It is not necessary to be sure of success, only to be unafraid of failure.

- Datatypes
- Democratization
- Independence
- Variations
- Immutability
- Incrementalism
- Standing your ground against optimization

# Datatypes



#### @AgileAlliance

#### AgileAlliance

#### Refactoring

• Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behaviour preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and **aids to formal reasoning such as type systems**.

https://www.agilealliance.org/glossary/refactoring/

#### aids to formal reasoning such as type systems.

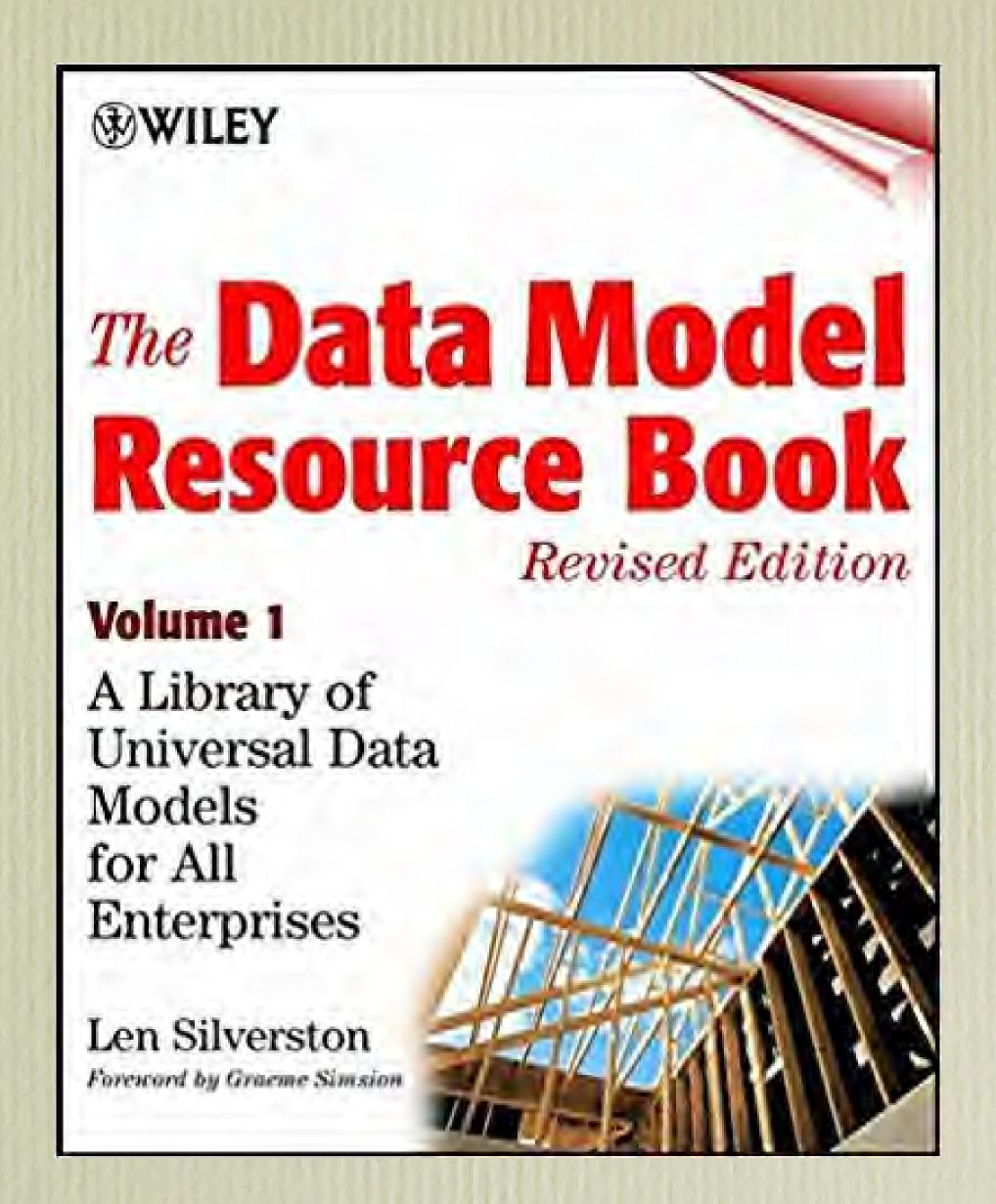
- Typescript
- Go, Rust
- Java, Swift, Kotlin
- Protobuf
- Relational Database Tables

#### @JeffDean

Jeff Dean

## JSON is Tech Debt

- Low-level MapReduce interfaces are in terms of byte arrays
  - Hardly ever use textual formats, though: slow, hard to parse
  - Most input & output is in encoded Protocol Buffer format



## Democratization

# Use Spreadsheets



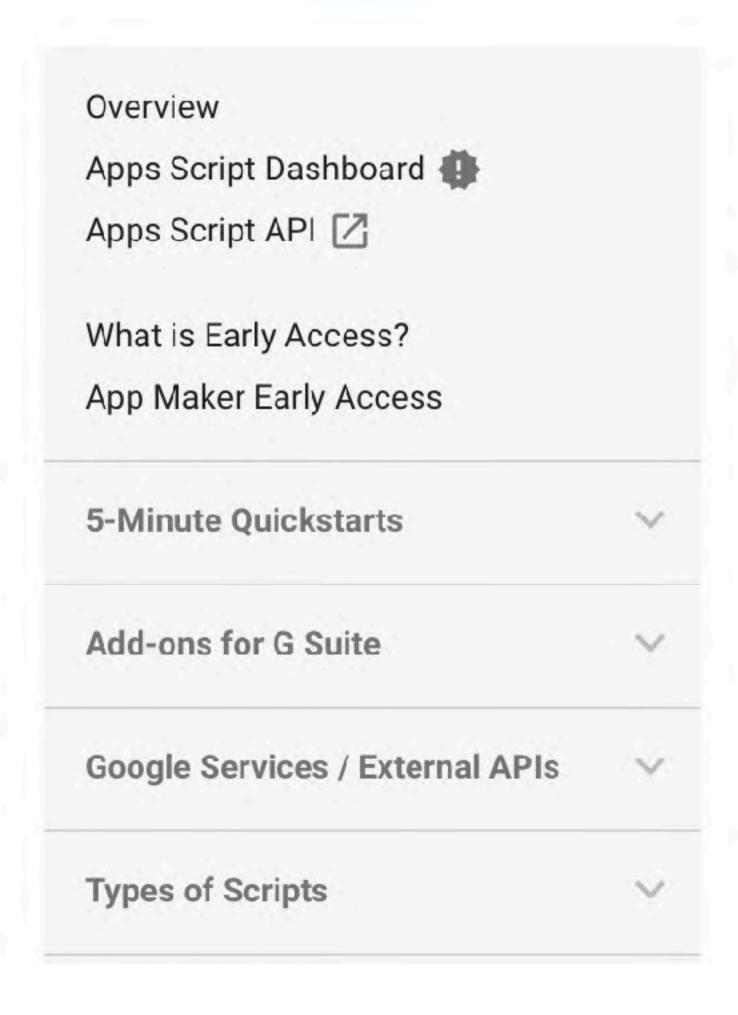
HOME **GUIDES** 

REFERENCE

SAMPLES

SUPPORT

**APPS SCRIPT API** 



#### Extending Google Sheets



Google Apps Script lets you do new and cool things with Google Sheets. You can use Apps Script to add custom menus, dialogs, and sidebars to Google Sheets. It also lets you write custom functions for Sheets, as well as integrate Sheets with other Google services like Calendar, Drive, and Gmail.

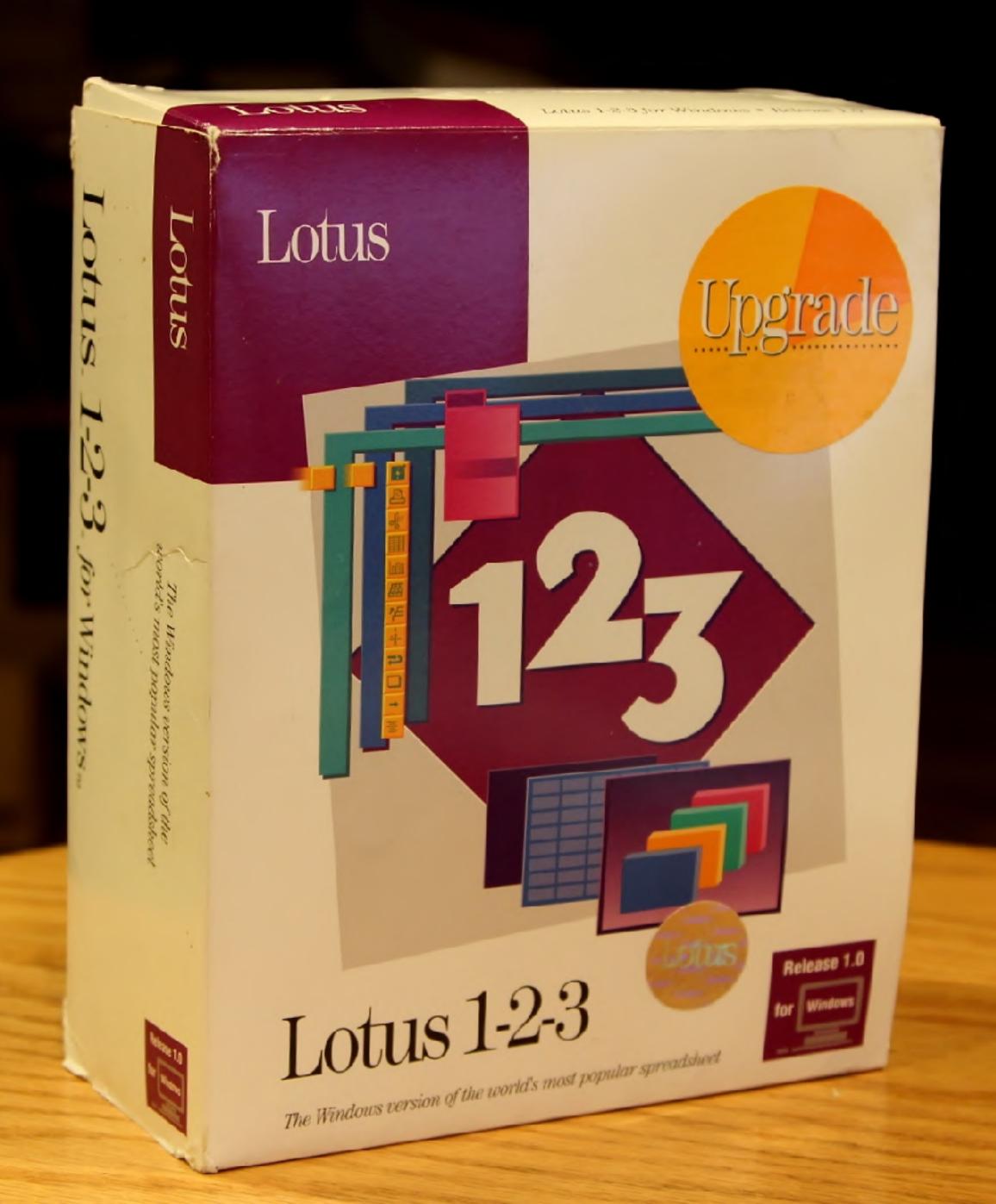
Most scripts designed for Google Sheets manipulate arrays to interact with the cells, rows, and columns in a spreadsheet. If you're not familiar with arrays in JavaScript, Codecademy offers a great training module for arrays. (Note that this course was not developed by and is not associated with Google.)

For a quick introduction to using Apps Script with Google Sheets, see the 5-minute quickstart guide for Macros, Menus, and Custom Functions.



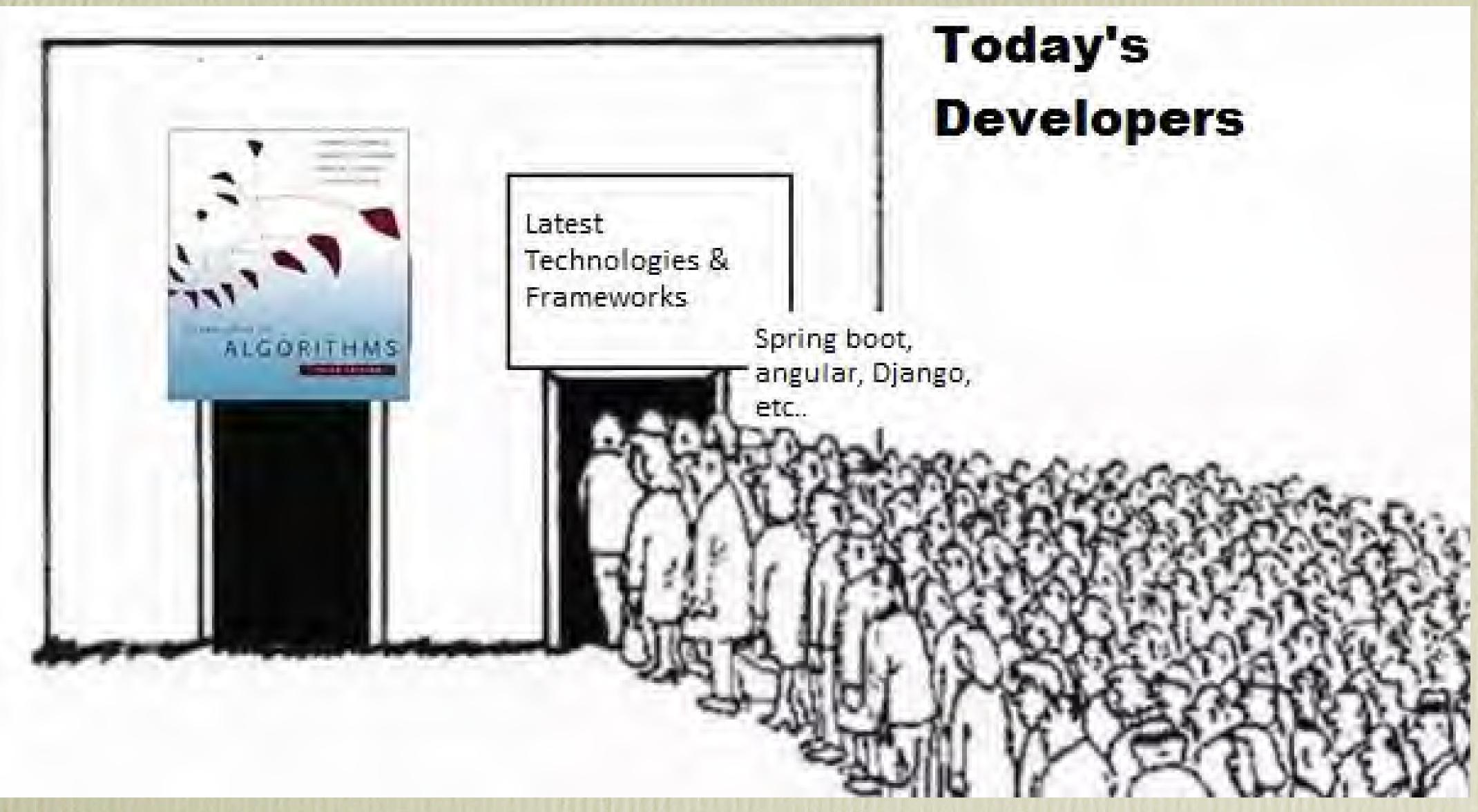


Microsoft



Independence (1)





### Don't Have Dependencies

- Adding an unnecessary dependency is manufacturing technical debt
- All dependencies are unnecessary

```
$ wc requirements.txt
160 178 2777 requirements.txt
```

\$ wc package.json
145 311 4661 package.json



### Only the Giants Survive

- There can be  $n^2$  interactions among n components
- Reusable components take 3-5 times longer to write than non-reusable ones. (Good ones take 50% to 200% longer to write than mediocre ones)
- Platforms require someone taking responsibility for coherence and stability.
- Big components (over five million lines of code) work. The cost of documentation is too high for smaller components.
- Reusable components often consume 100 times the resources of custom built components. (the reason today's PC is not 10,000 times more featureful than the Xerox Alto)

## Before using a library

- just copy-and-pasting the function you need into your code
- implement it yourself
- if during the course of the implementation, you realize that the problem is much more complex than you anticipated, then you now understand the reason for the library
- if the implementation is not that complicated, then you are done.

# An Analysis of Errors in a Reuse-Oriented Development Environment (Thomas, Delis, Basili — 1997)

- There is a clear benefit from reuse in terms of reduced error density when the reuse is verbatim or via slight modification. However, reuse through slight modification only shows about a 59% reduction in total error density, while verbatim reuse results in more than a 90% reduction compared to newly developed code.
- A number of studies have reported *higher defect/error densities in smaller components* than in larger components (Basili and Perricone, 1984; Shen et al., 1985; Lind and Vairavan, 1989; Miiller and Paulish, 1993).

http://www.cs.umd.edu/~basili/publications/journals/J70.pdf

Immutability



### The Important Question

- I very frequently get the question: 'What's going to change in the next 10 years?' And that is a very interesting question; it's a very common one.
- I almost never get the question: 'What's not going to change in the next 10 years?'
- And I submit to you that that second question is actually the more important of the two because you can build a business strategy around the things that are stable in time.

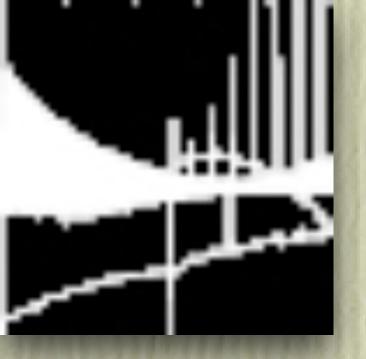
### The Important Question

- In our retail business, we know that customers want low prices, and I know that's going to be true 10 years from now.
- They want fast delivery; they want vast selection.
- It's impossible to imagine a future 10 years from now where a customer comes up and says, 'Jeff I love Amazon; I just wish the prices were a little higher,' [or] 'I love Amazon; I just wish you'd deliver a little more slowly.'
- Impossible.
- And so the effort we put into those things, spinning those things up, we know the energy we put into it today will still be paying off dividends for our customers 10 years from now.
- When you have something that you know is true, even over the long term, you can afford to put a lot of energy into it."

You can use an eraser on the drafting table or a sledgehammer on the construction site.

— Frank Lloyd Wright

# Aspire to build software that will never change.



#### @hashbreaker

#### Daniel J. Bernstein

#### Some thoughts on security after ten years of qmail

In March 1997, I took the unusual step of publicly offering \$500 to the first person to publish a verifiable security hole in the latest version of qmail: for example, a way for a user to exploit qmail to take over another account. My offer still stands. Nobody has found any security holes in qmail. I hereby increase the offer to \$1000.



#### @hashbreaker

#### Daniel J. Bernstein

#### Some thoughts on security after ten years of qmail

- eliminating bugs
  - enforcing explicit data flow
  - simplifying integer semantics
  - avoid parsing
- eliminating code
  - identifying common functions
  - automatically handling temporary errors

- reusing network tools
- reusing access controls
- reusing the file system
- eliminating trusted code
  - isolating single source transformations
  - delaying multiple-source merges

Perhaps the Objective should not be to write crappy code that you will fiddle with forever

#### Incrementalism

# Don't use anything new

#### What is "new"

- New to me? Or New New
- Is Java "new" to a Python shop?
- Is Swift "new" for an iOS app?
- Is AWS Lambda "new" if you haven't used it before.

# The Lindy Effect

the future life expectancy of a technology or an idea is proportional to its current age



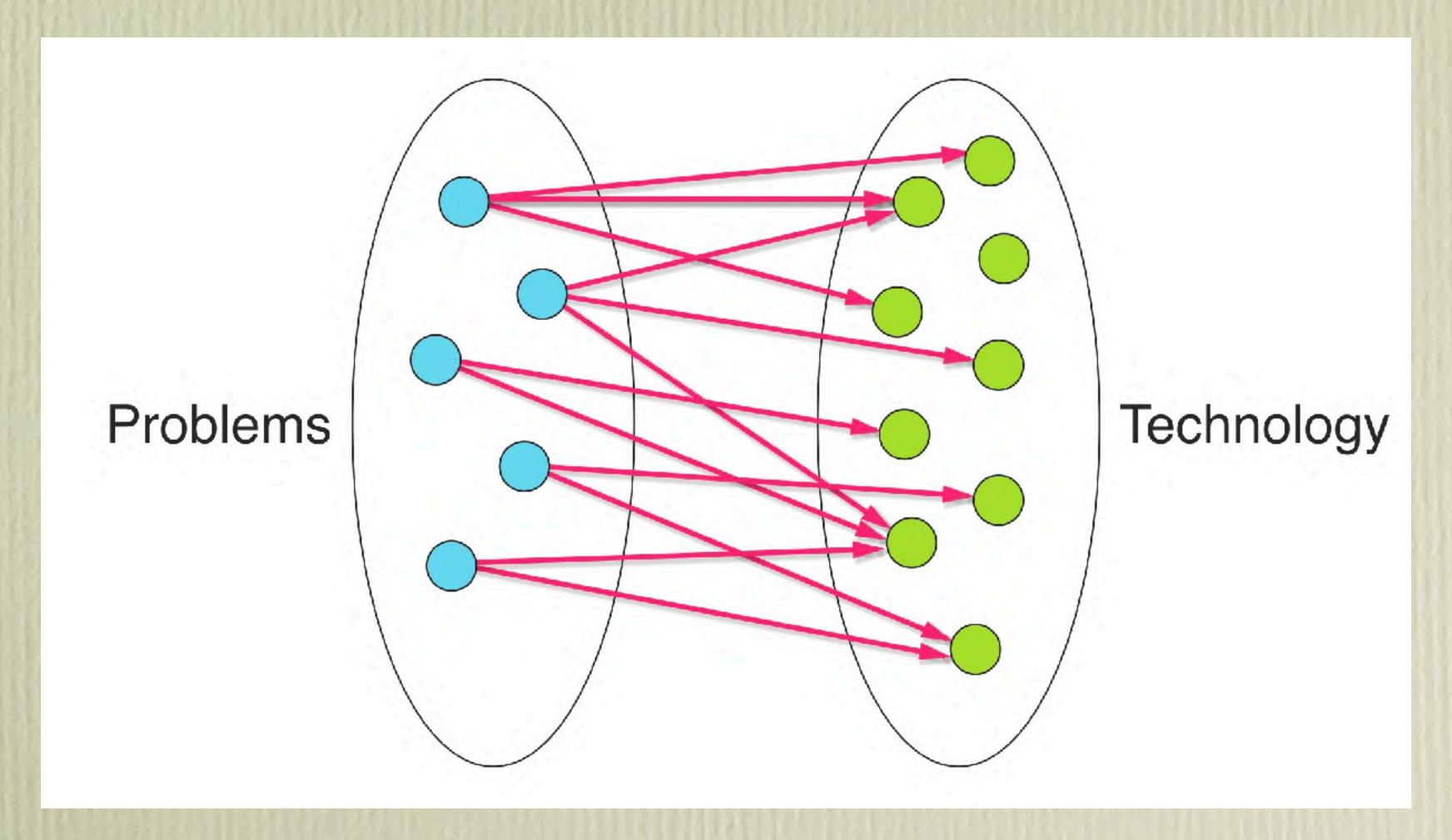
#### @wohali

#### Joan Touzet

- AIM: dead.
- ICQ: dead.
- MSN: dead.
- Y! chat: dead.
- Google XMPP/Jabber: dead.
- IRC survived.
- Remember that.



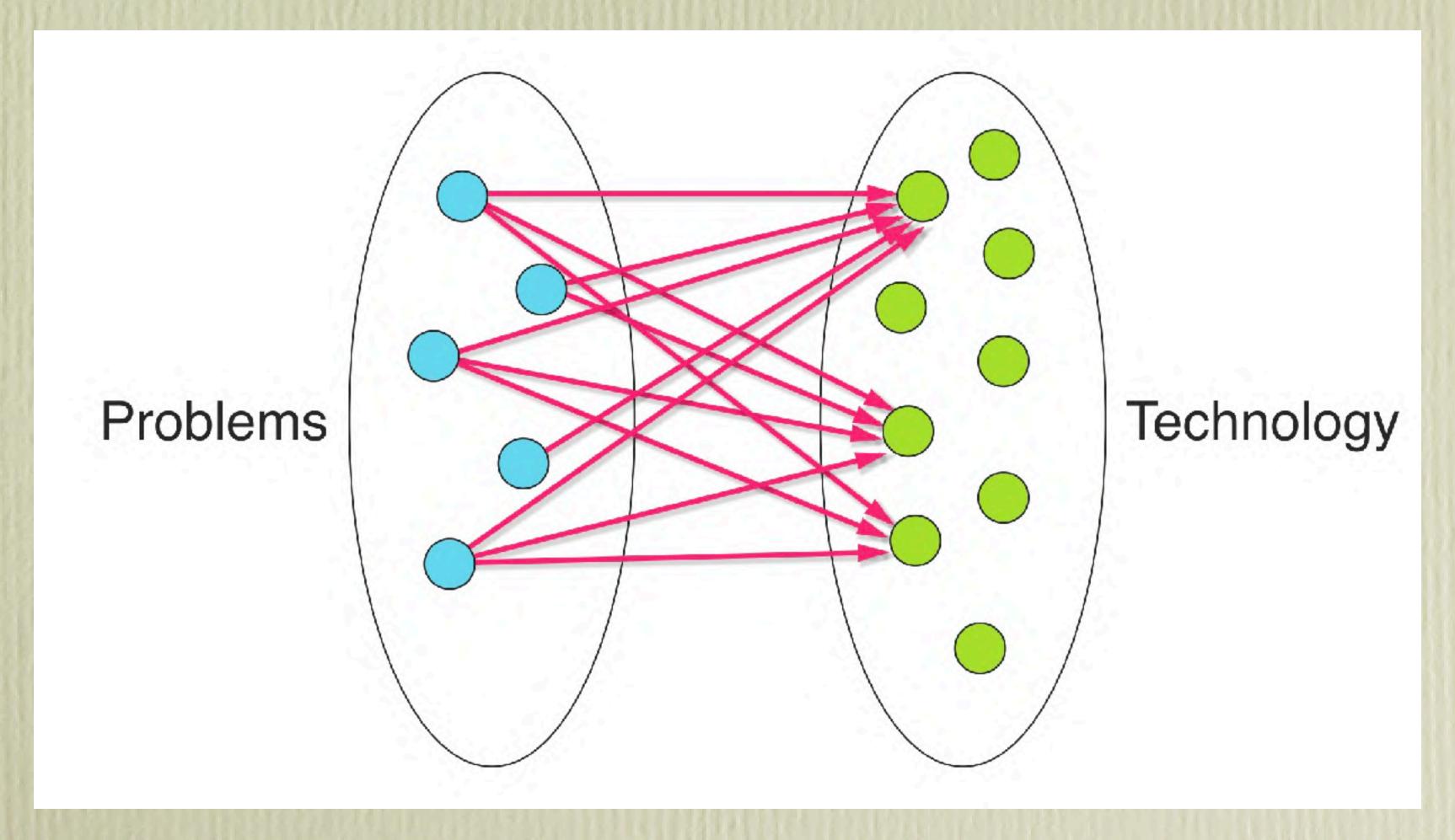
# Choose Boring Technology



http://mcfunley.com/choose-boring-technology



# Choose Boring Technology



http://mcfunley.com/choose-boring-technology



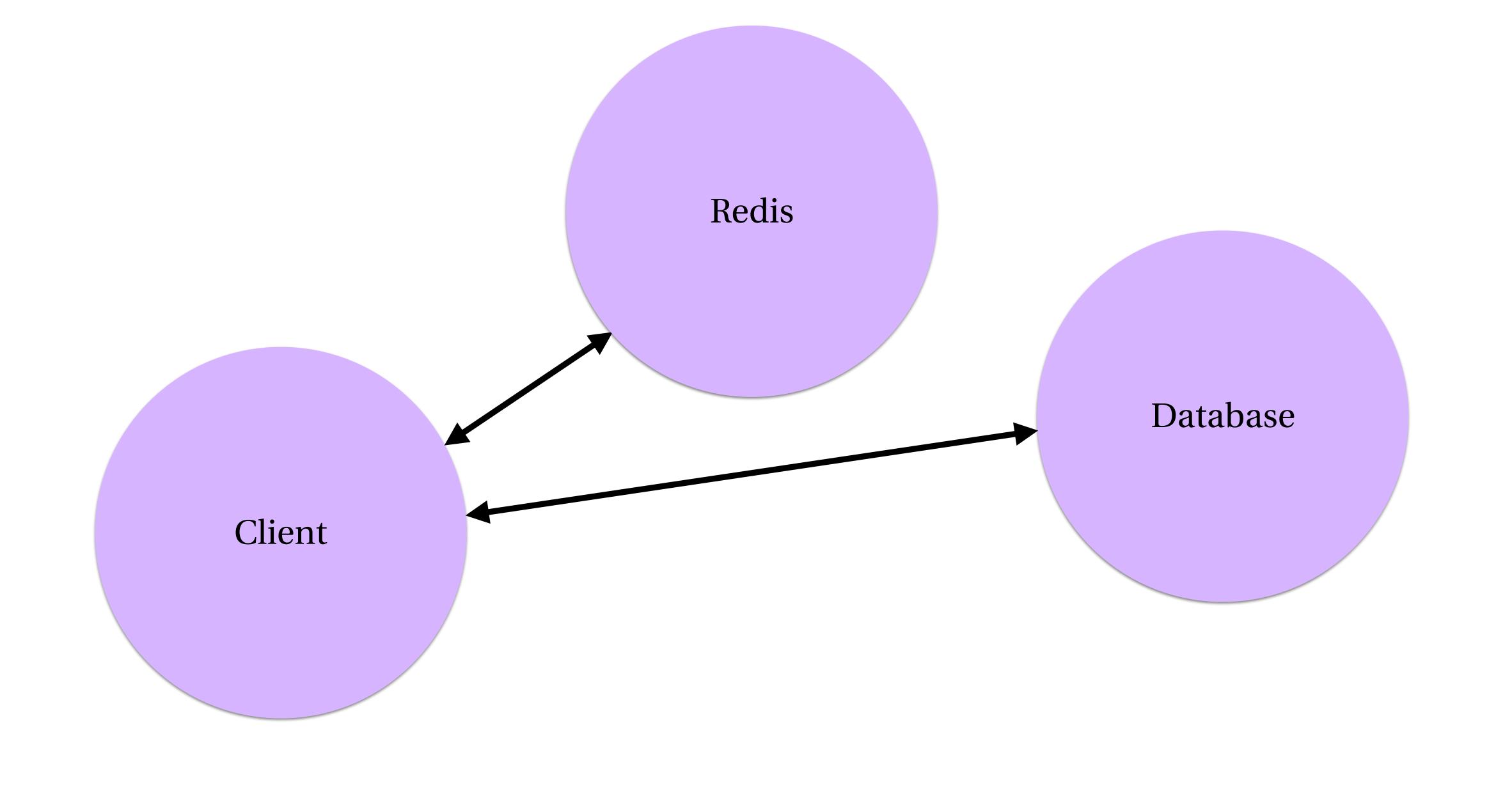
#### Master Your Tools

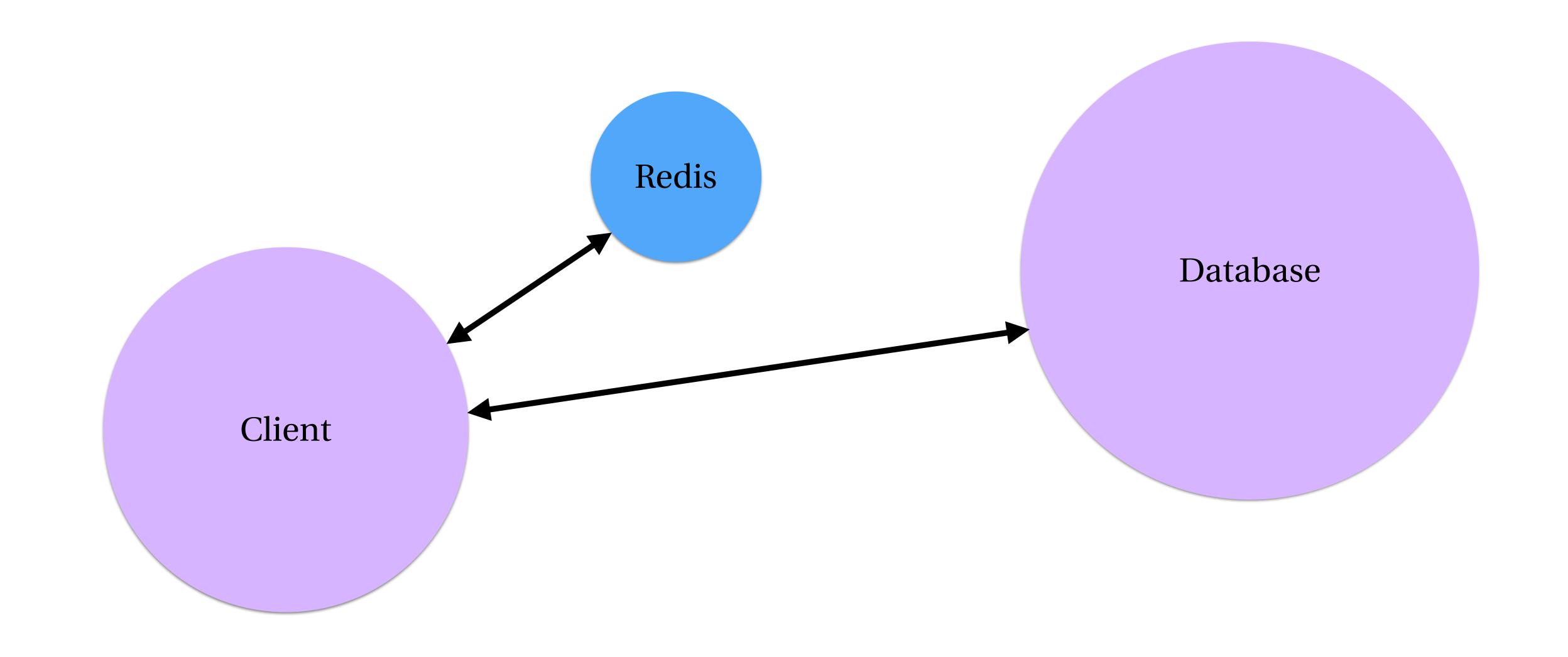
- The new thing won't be better, you just aren't aware of all of the ways it will be terrible yet.
- You should probably be using the tool that you hate the most. You hate it because you know the most about it.

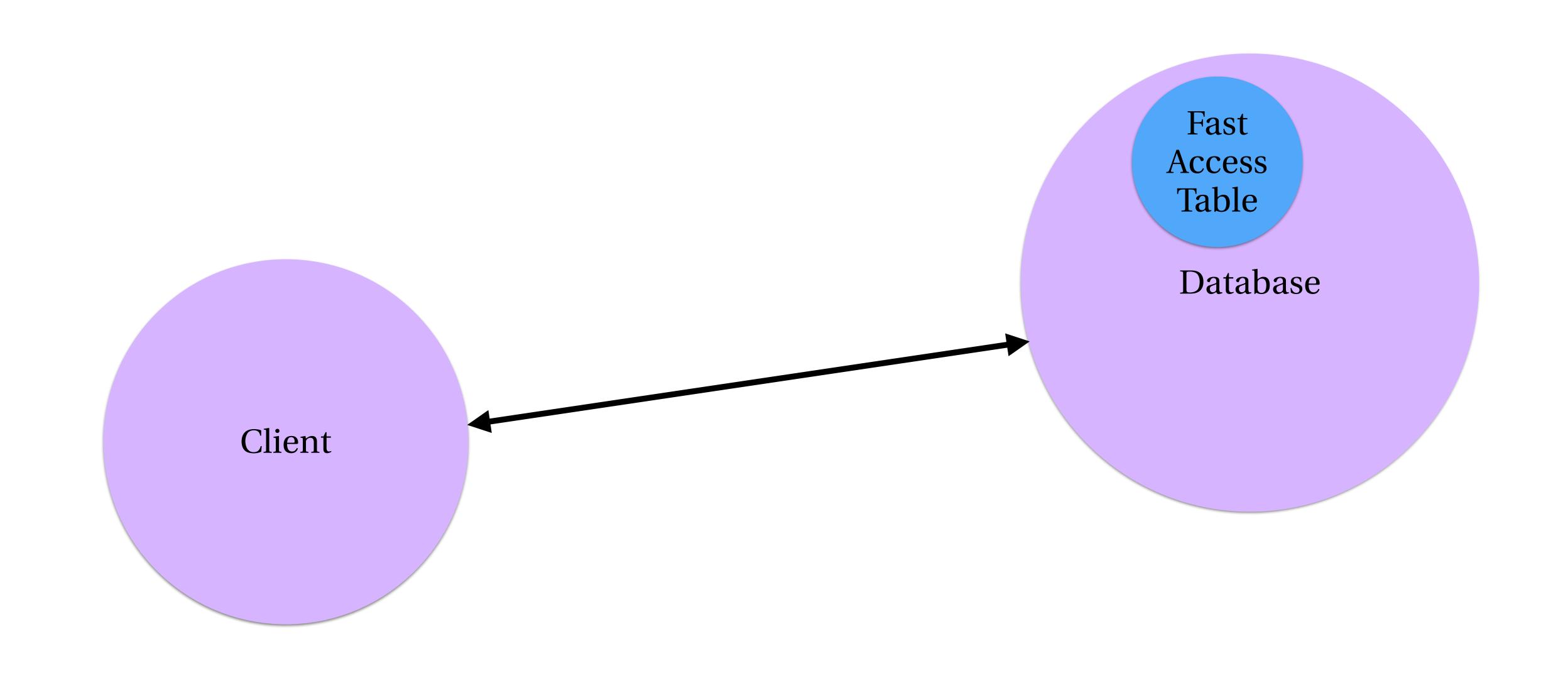
http://boringtechnology.club

# For the Love of God, Montresor! Don't Optimize

# Optimize without adding dependencies







#### https://sc5.io/posts/amazon-aws-lambda-data-caching-solutions-compared/

Cache backend	Read time (1)	Read time (2)	Read time (3)	Read time (4)	Read time (5)
ElastiCache Memcached	3 ms	3 ms	1 ms	2 ms	1 ms
ElastiCache Redis	4 ms	4 ms	1 ms	2 ms	1 ms
RDS Aurora	9 ms	9 ms	1 ms	10 ms	1 ms
RDS Postgres	3 ms	3 ms (*)	1 ms	16 ms	1 ms (*)
RDS MySQL	13 ms	8 ms	1 ms	17 ms	1 ms
SimpleDB	59 ms	56 ms	20 ms	105 ms	19 ms
DynamoDB	60 ms	58 ms	46 ms	56 ms	45 ms

<sup>(1) 1</sup> read within Lambda function, no concurrency

<sup>(2) 1</sup> read within Lambda function, 20 concurrent Lambda invocations

<sup>(3) 1000</sup> reads, no concurrency

<sup>(4) 1000</sup> reads with concurrency of 20 parallel reads within Lambda function

<sup>(5) 1000</sup> reads, no concurrency within Lambda function, 20 concurrent Lambda invocations

<sup>(\*)</sup> RDS Postgres (using default settings) threw errors on some requests when running 20 concurrent Lambda invocations.

#### http://code.jjb.cc/benchmarking-postgres-vs-redis

So I threw together a simple benchmark. I generated 1000 key/value pairs, and then accessed random values 100,000 times. I also benchmarked a third system: in-memory access to a ruby hash. The results:

Postgres: 8.9 seconds

Redis: 5.3 seconds

Memory: 0.01 seconds

# Aspire to One Mechanism for Data Persistence

HTTP/2

# Thinking Through the Implications of Microservices

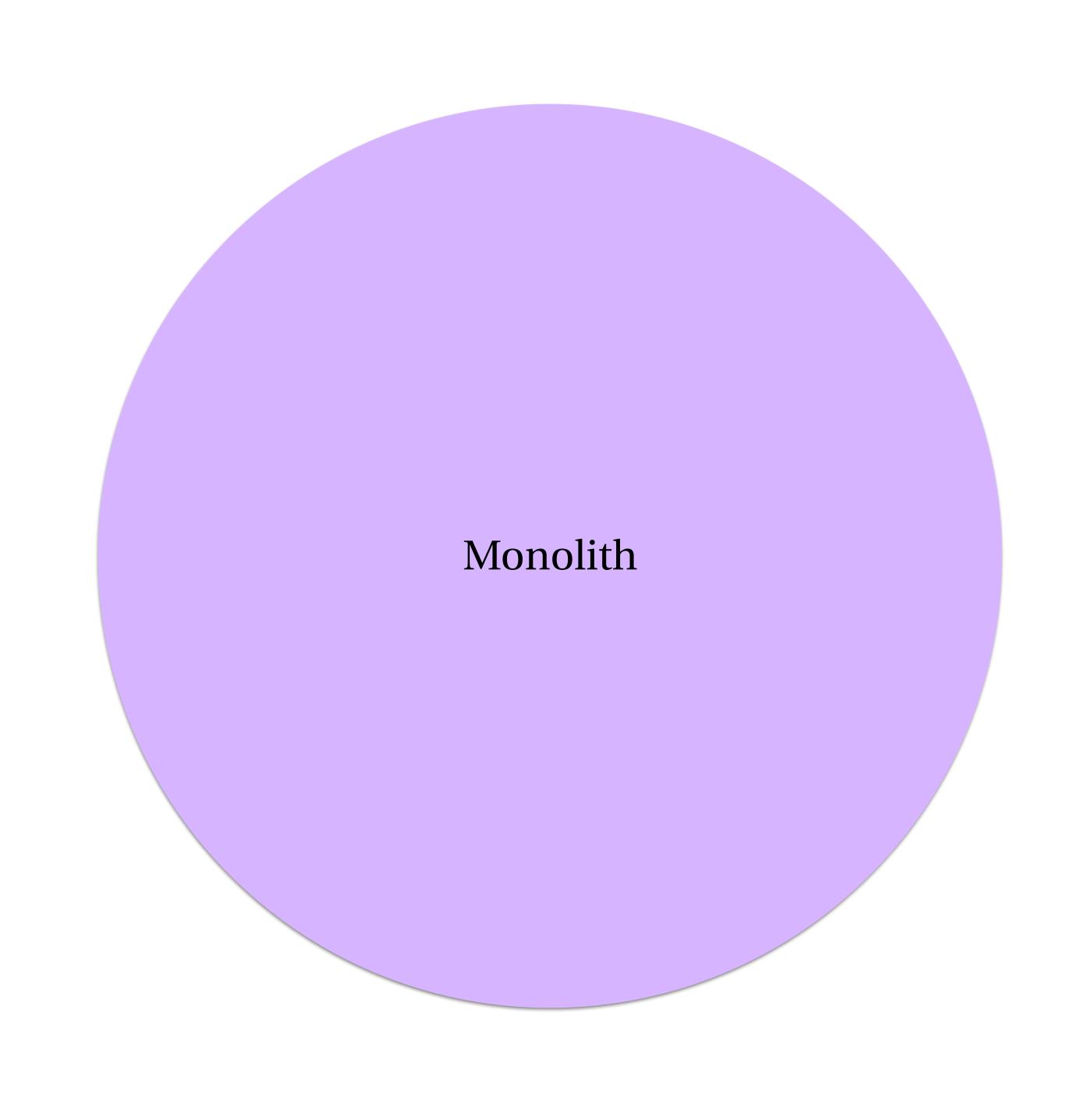
"It is never worth a first-class man's time to express a majority opinion. By definition, there are plenty of others to do that."

— Hardy

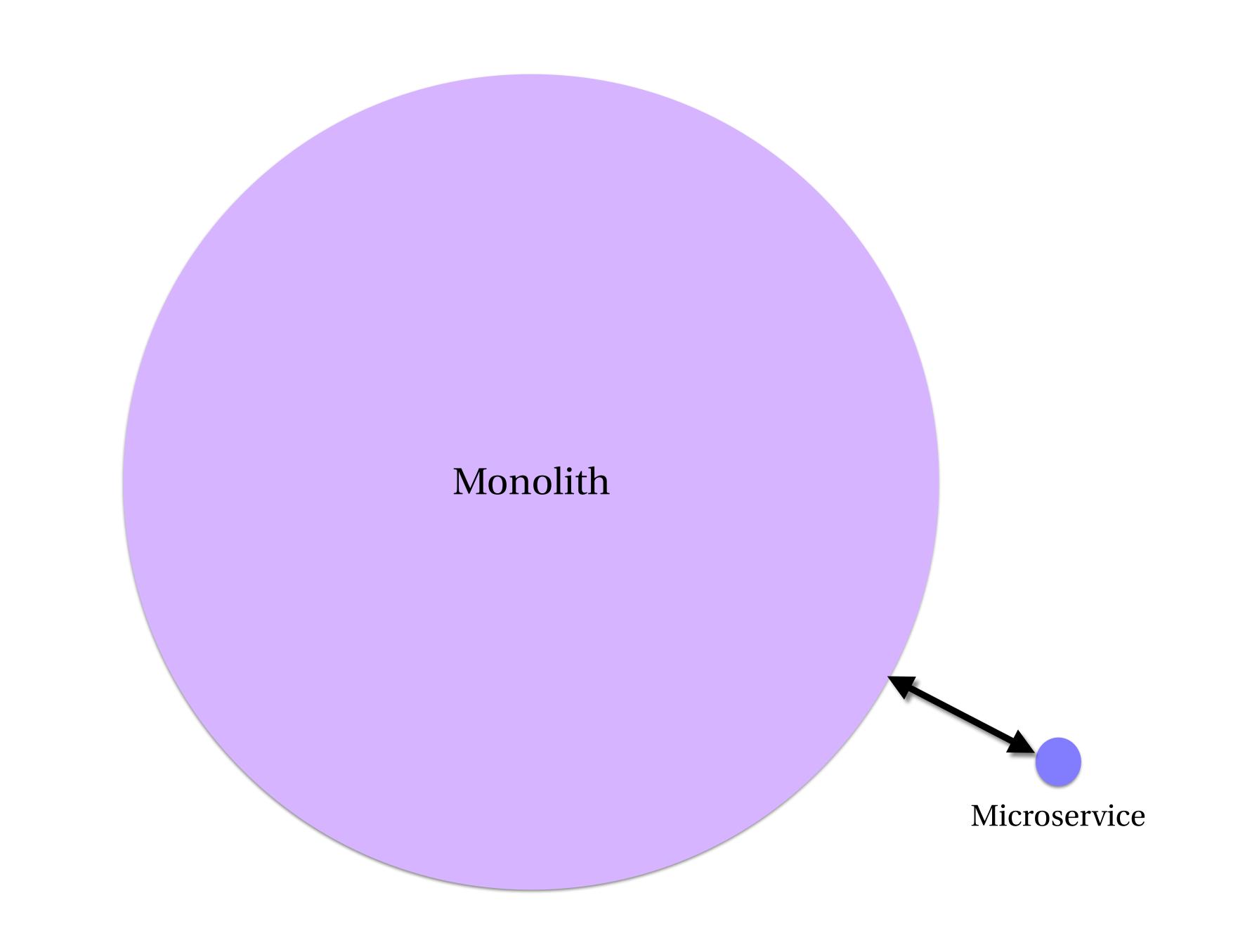
# The Nature of the Firm

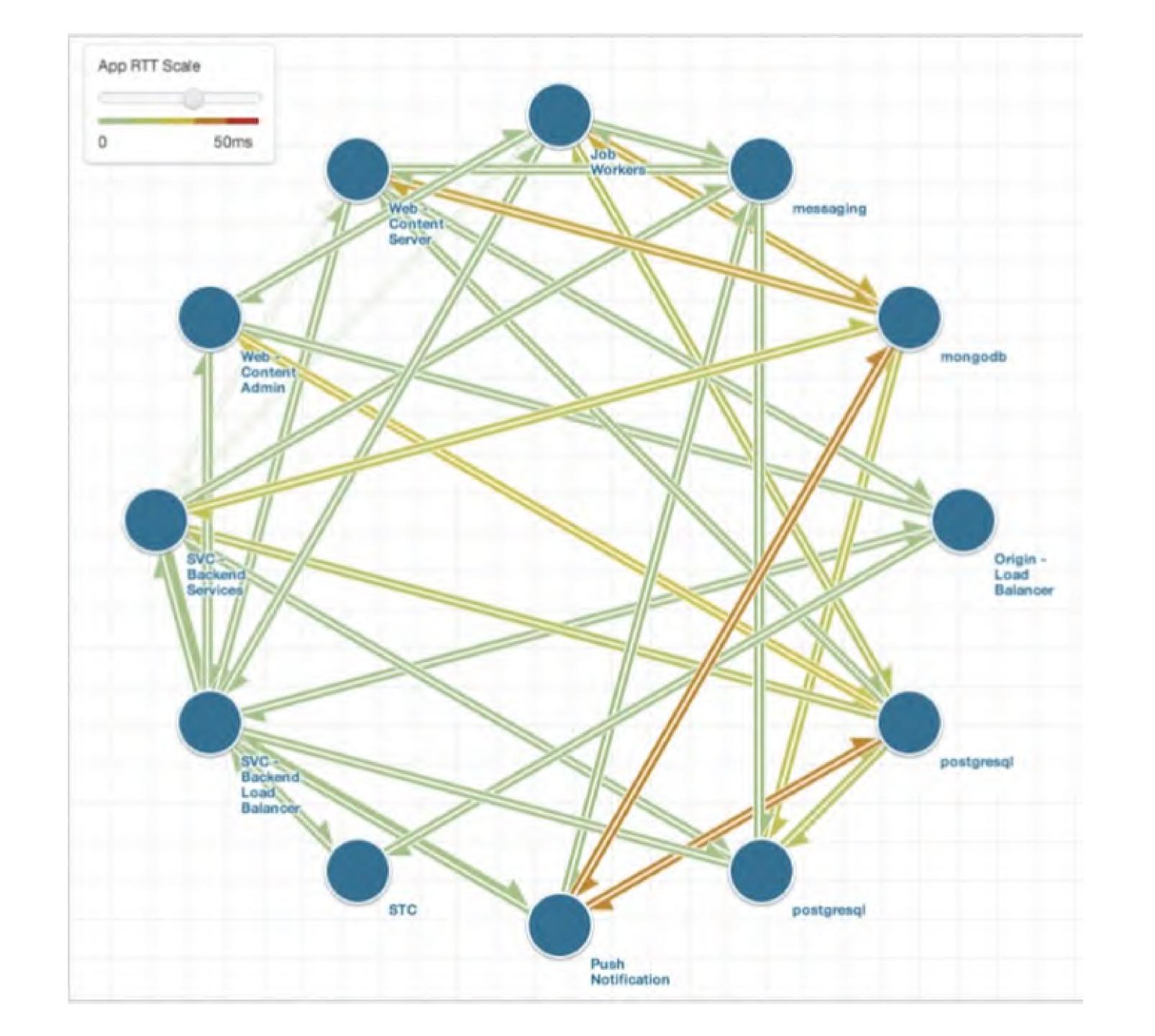
- Ronald H. Coase

"Naturally, a point must be reached where the costs of organising an extra transaction within the firm are equal to the costs involved in carrying out the transaction in the open market, or, to the costs of organising by another entrepreneur."

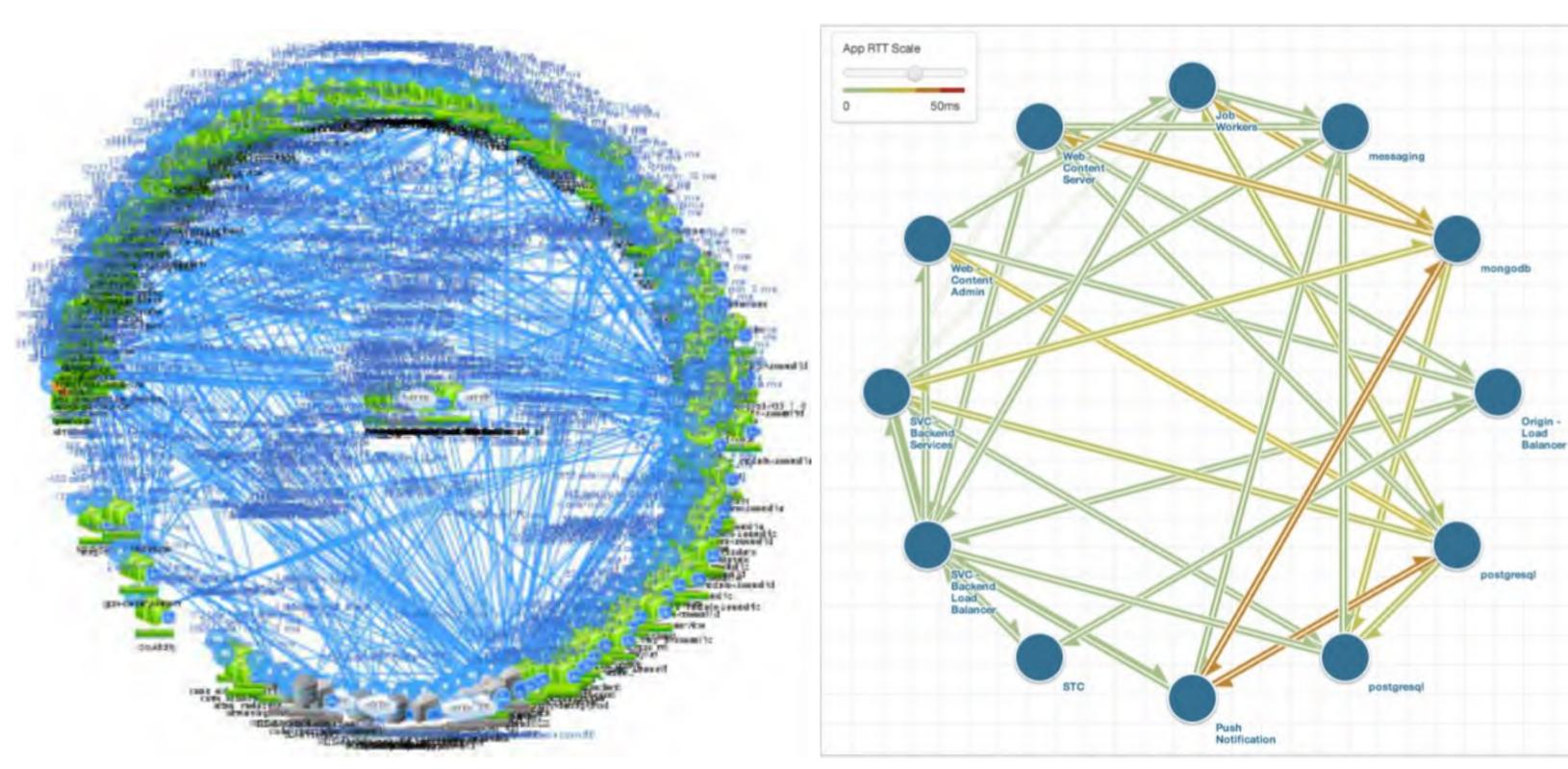




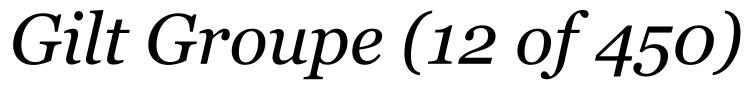


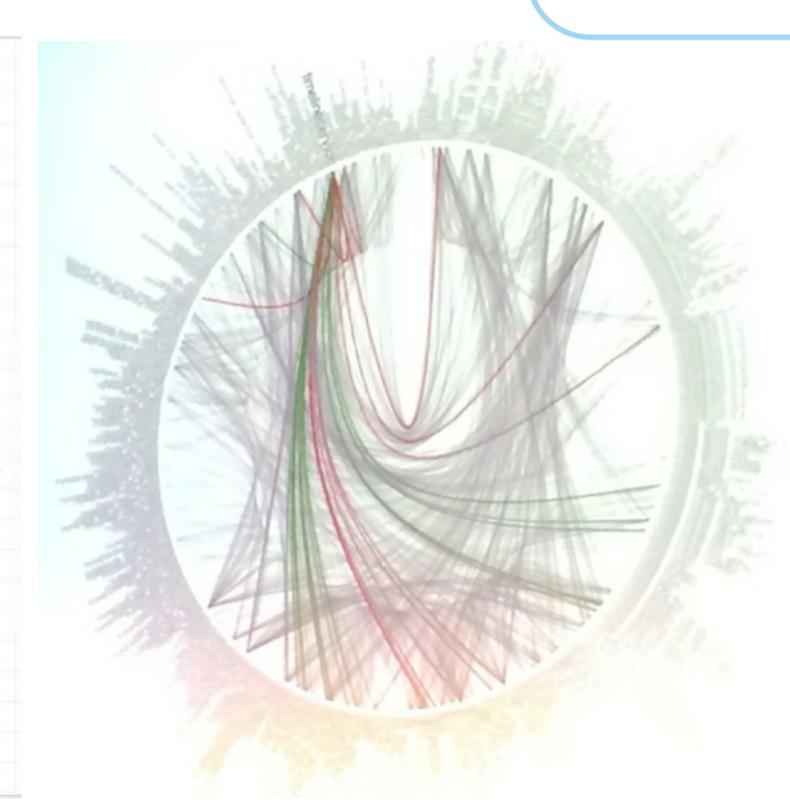


# "Death Star" Architecture Diagrams



Netflix



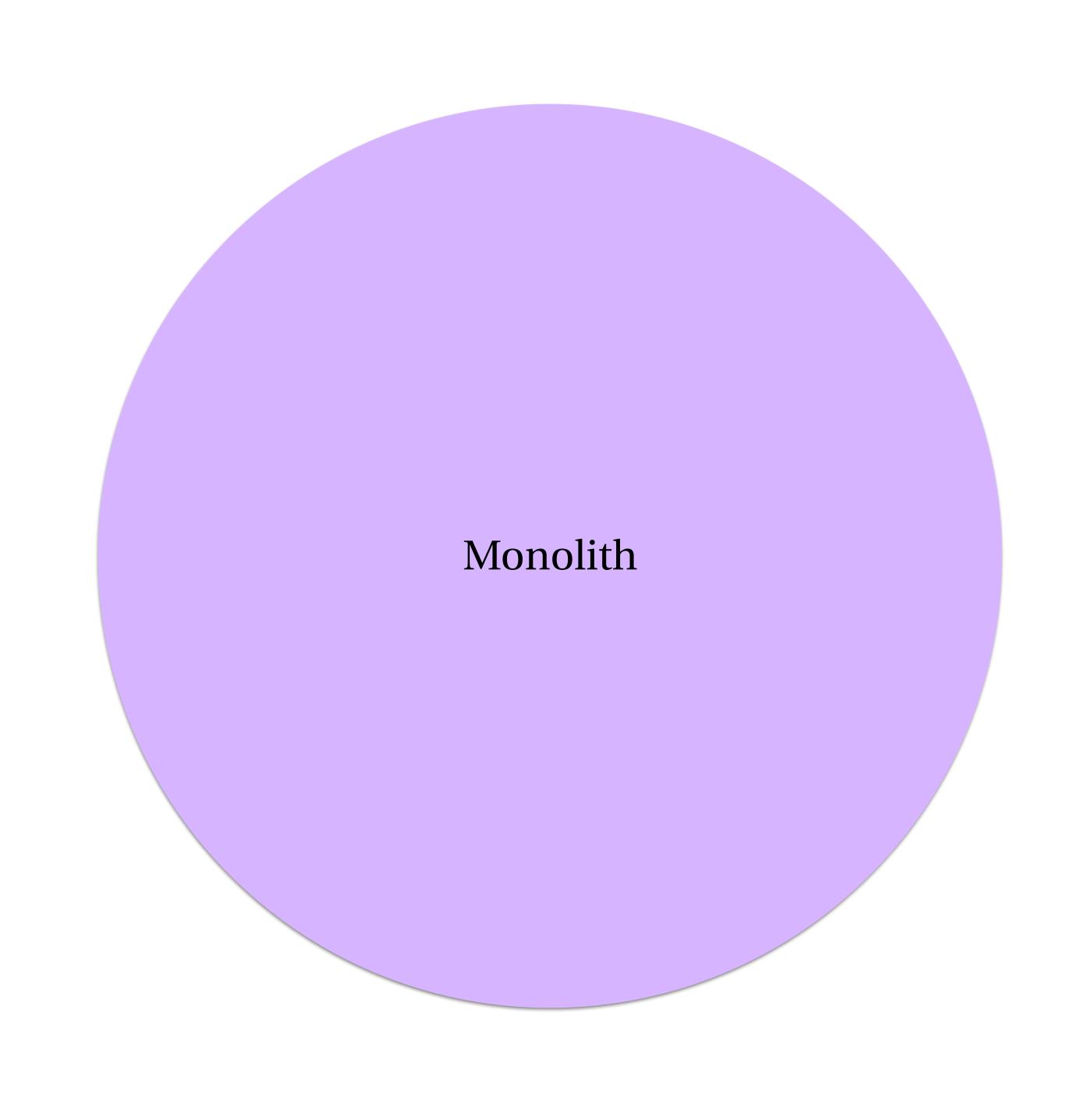


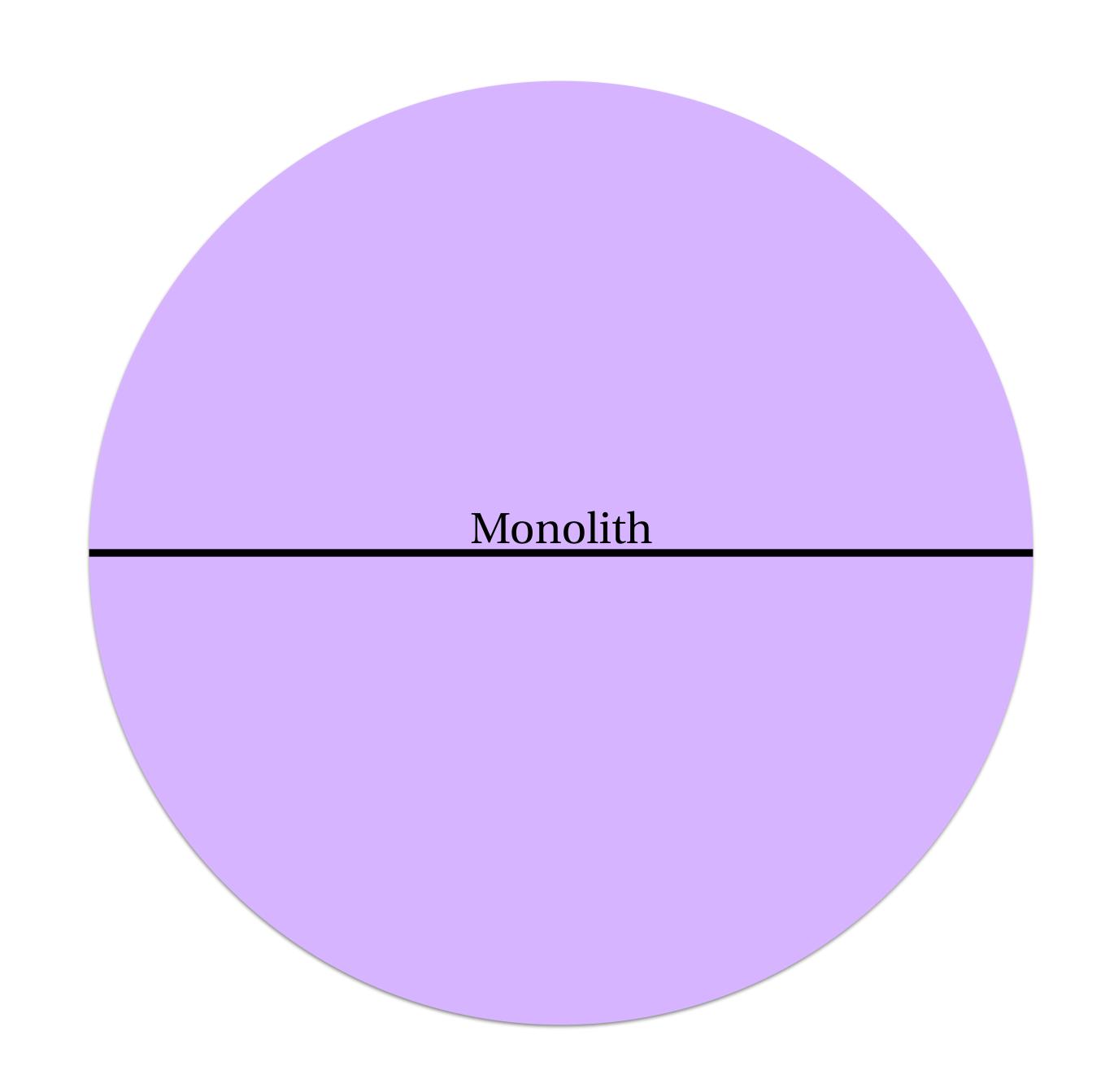
Twitter

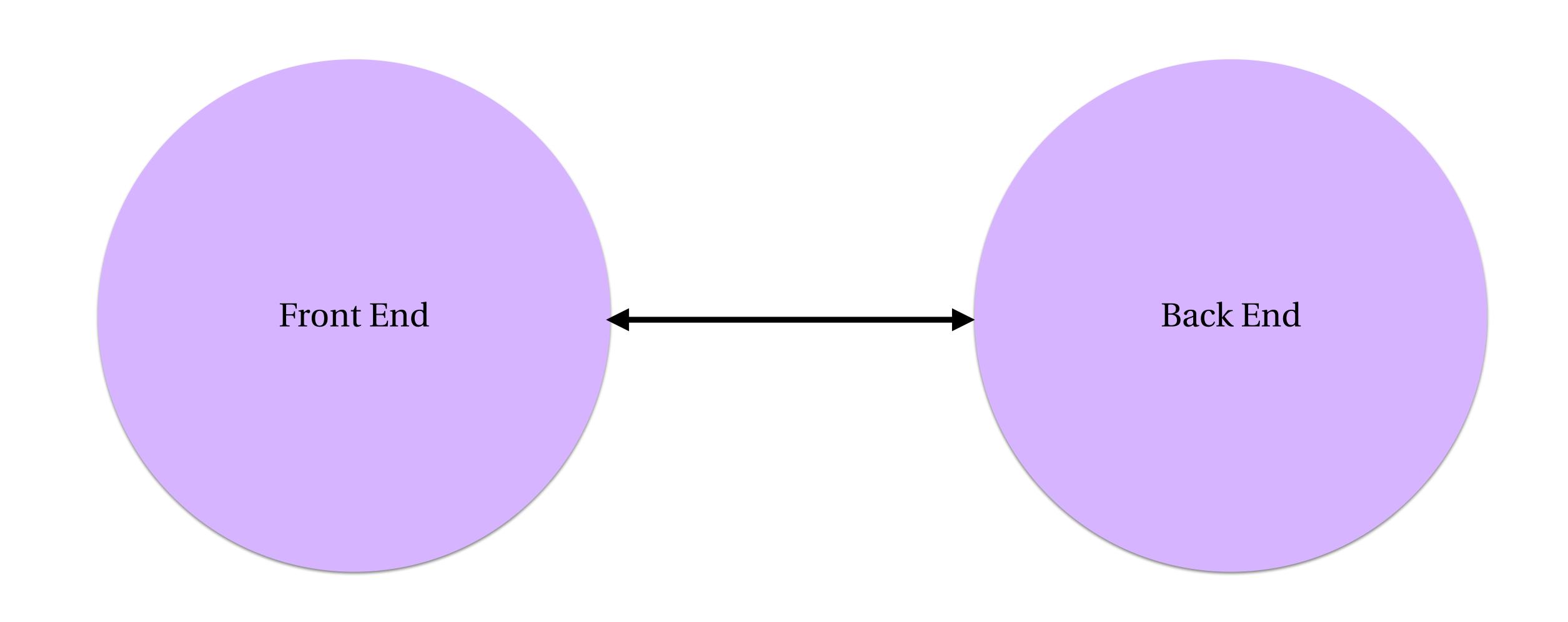


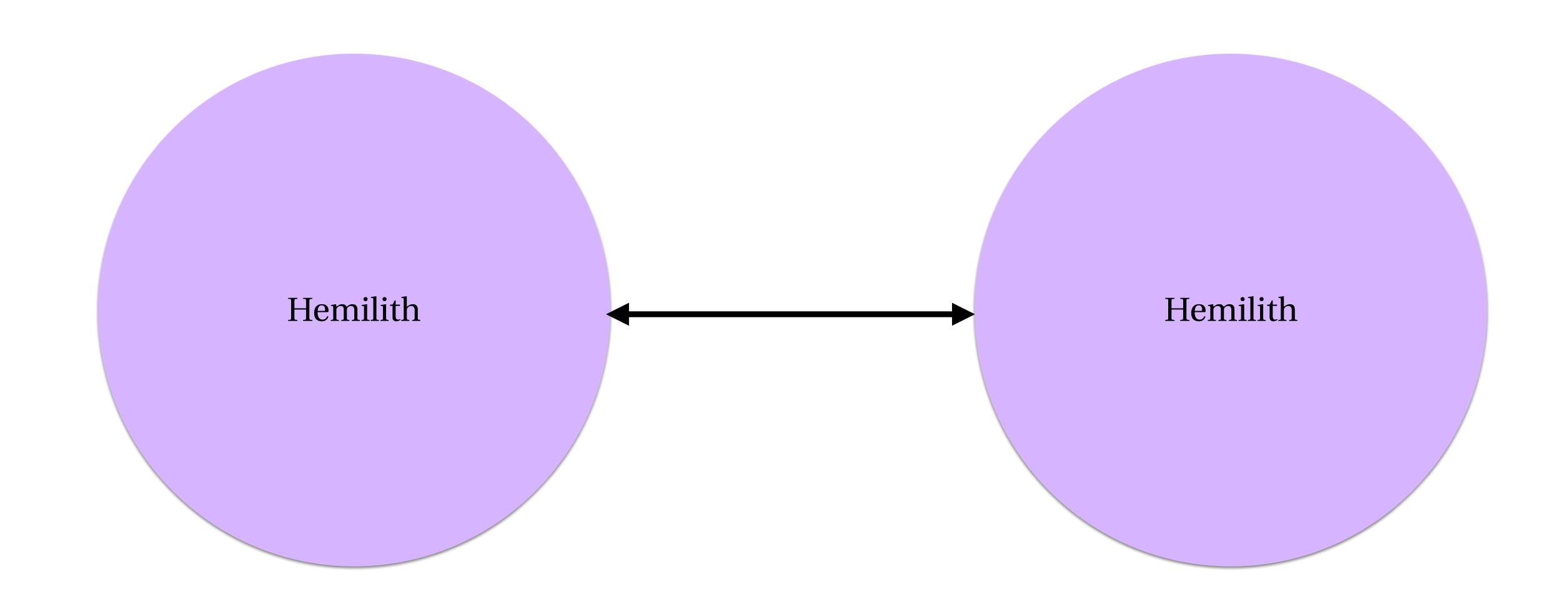
# Only the Giants Survive

- There can be n² interactions among n components
- Reusable components take 3-5 times longer to write than non-reusable ones. (Good ones take 50% to 200% longer to write than mediocre ones)
- Platforms require someone taking responsibility for coherence and stability.
- Big components (over five million lines of code) work. The cost of documentation is too high for smaller components.
- Reusable components often consume 100 times the resources of custom built components. (the reason today's PC is not 10,000 times more featureful than the Xerox Alto)









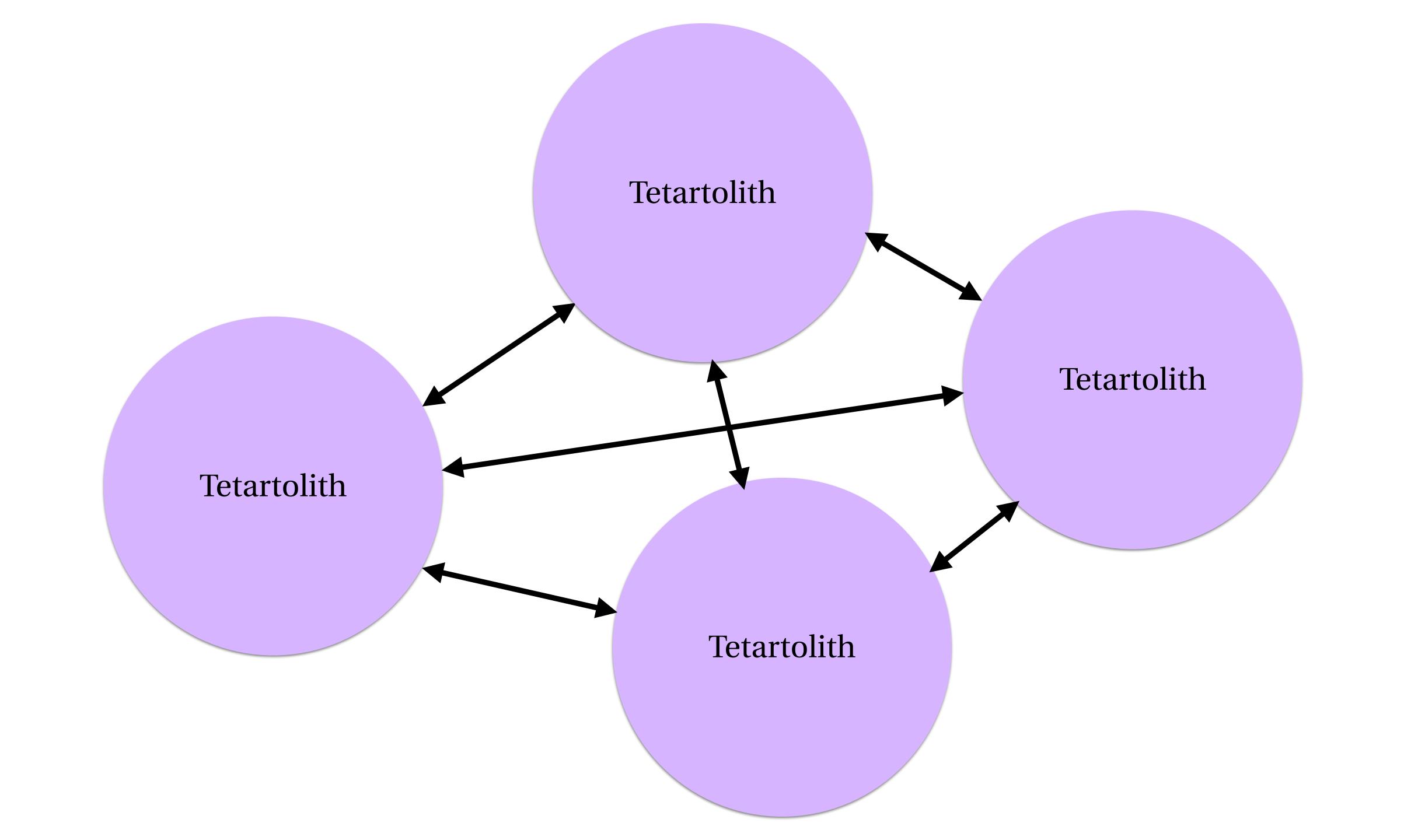




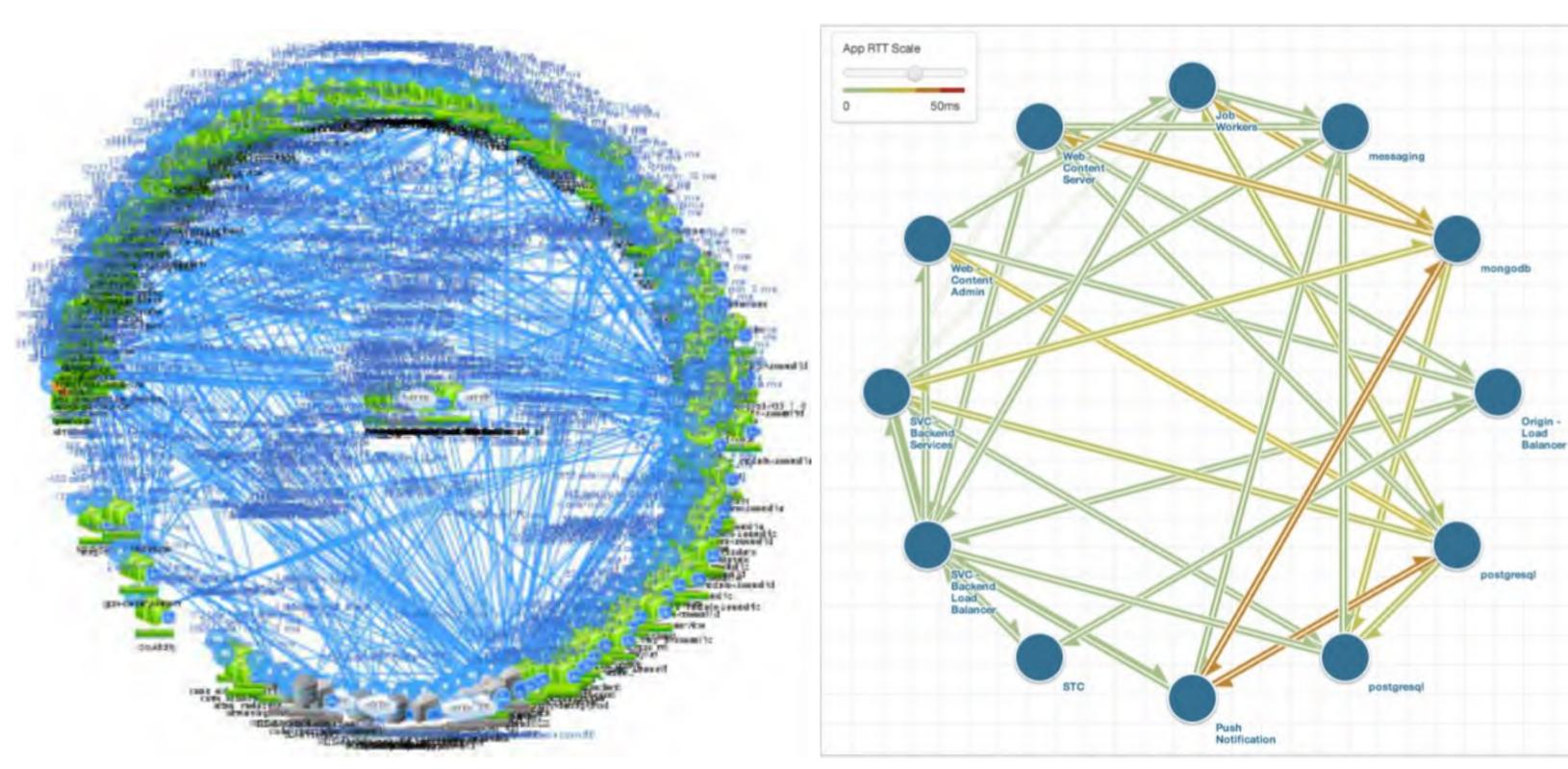
# Conway's Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

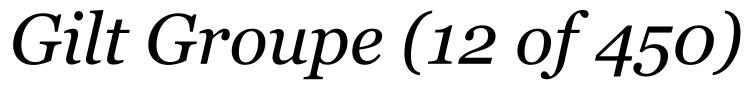
http://www.melconway.com/Home/Committees\_Paper.html

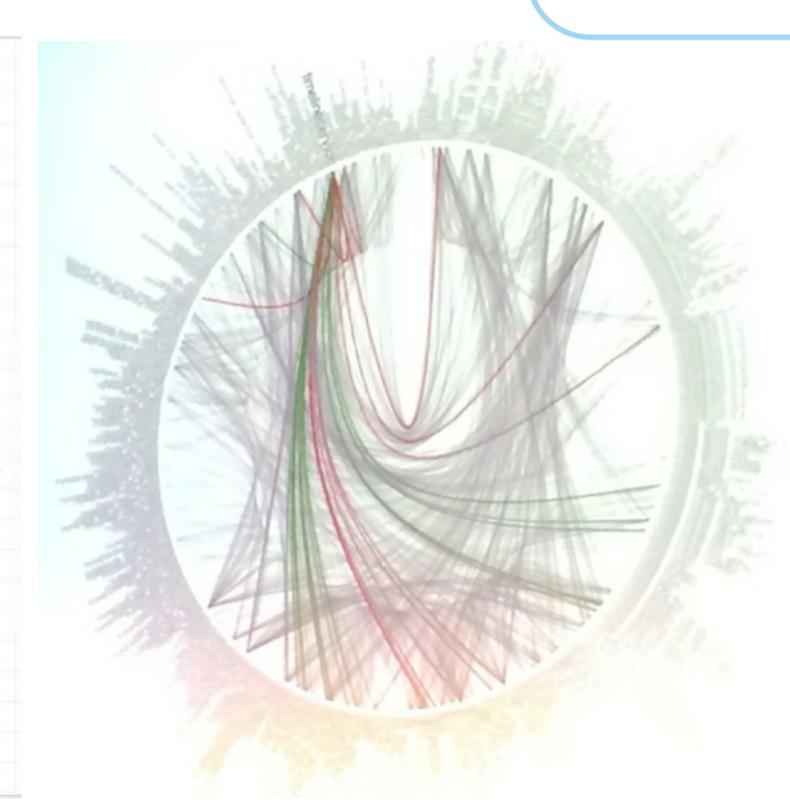


# "Death Star" Architecture Diagrams

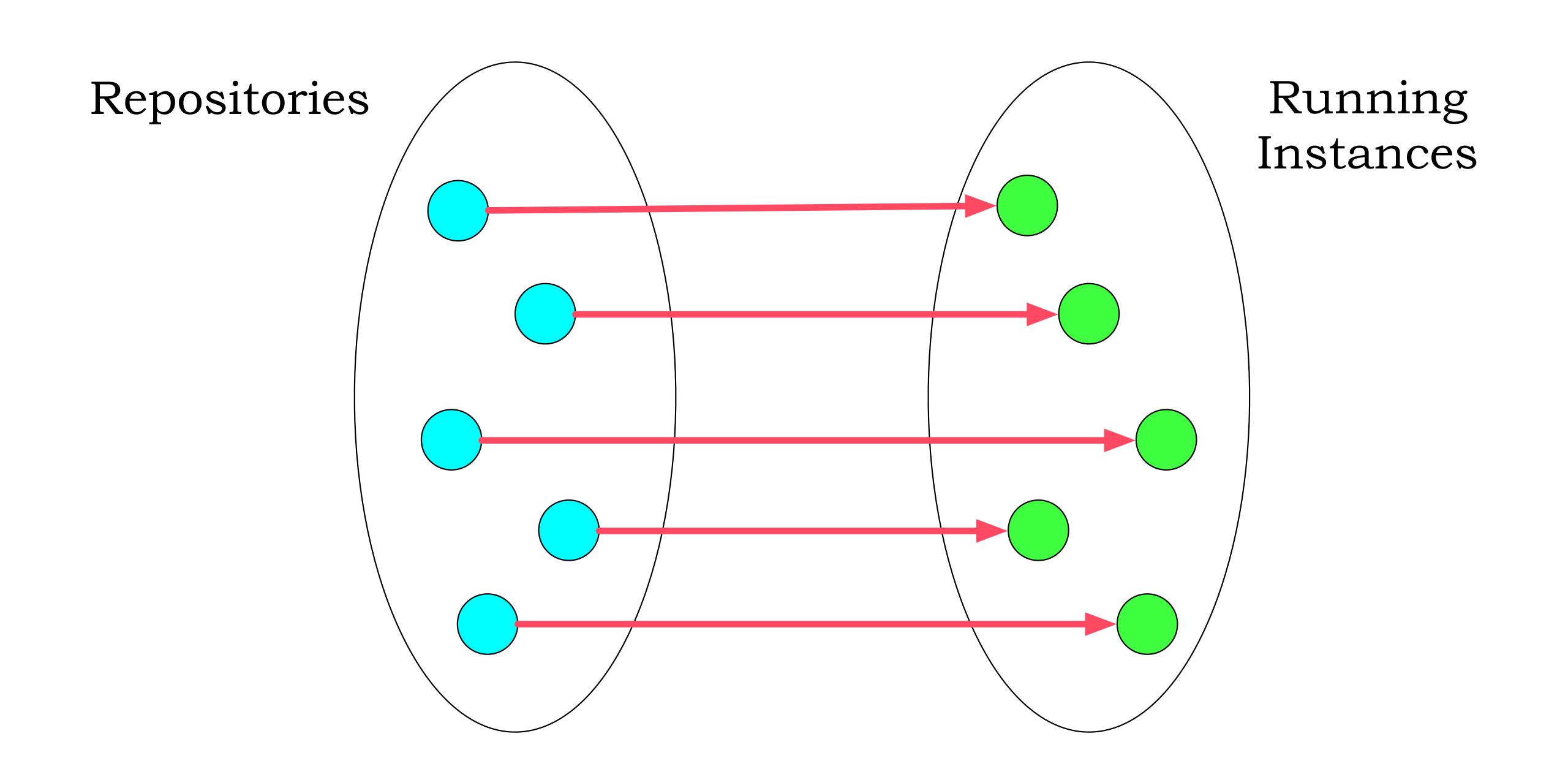


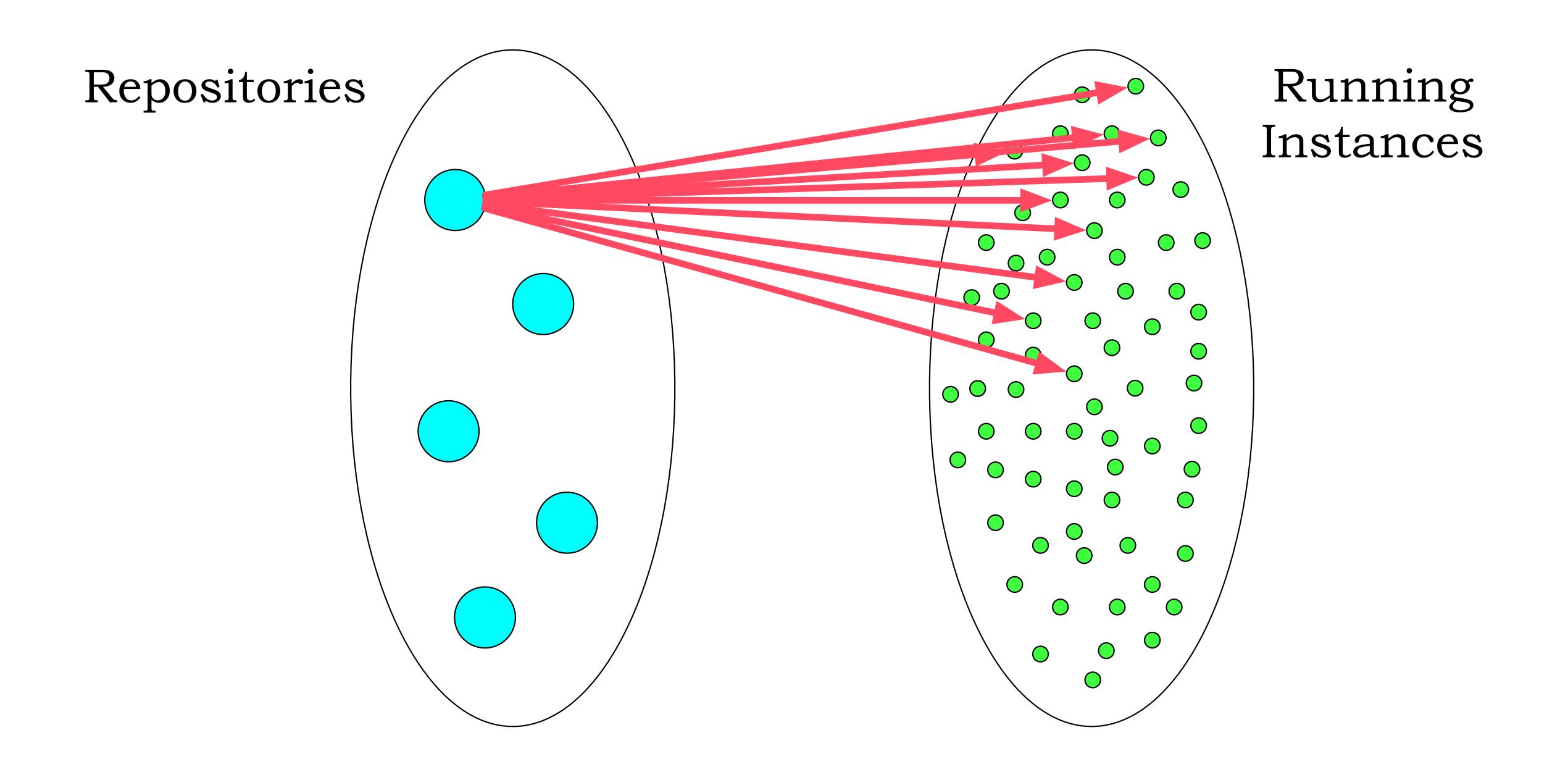
Netflix





Twitter

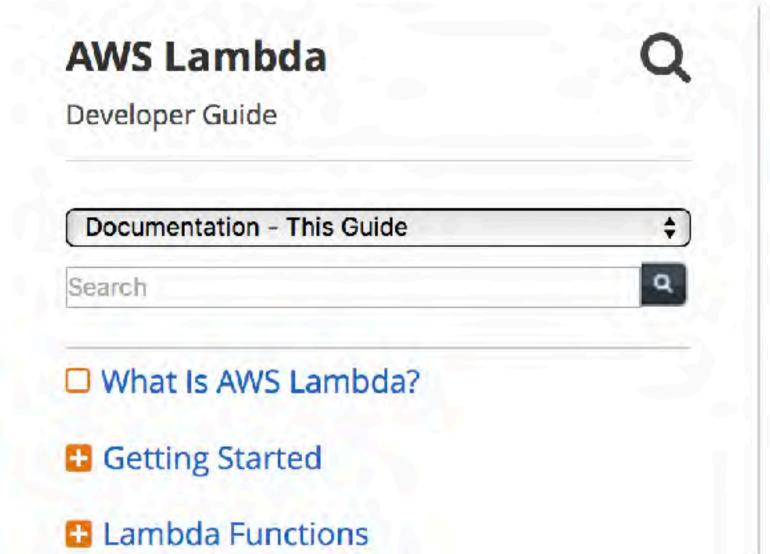




# AWS Lambda

- every update is versioned
- source code can be included
- the versioning mechanism maps to running code.

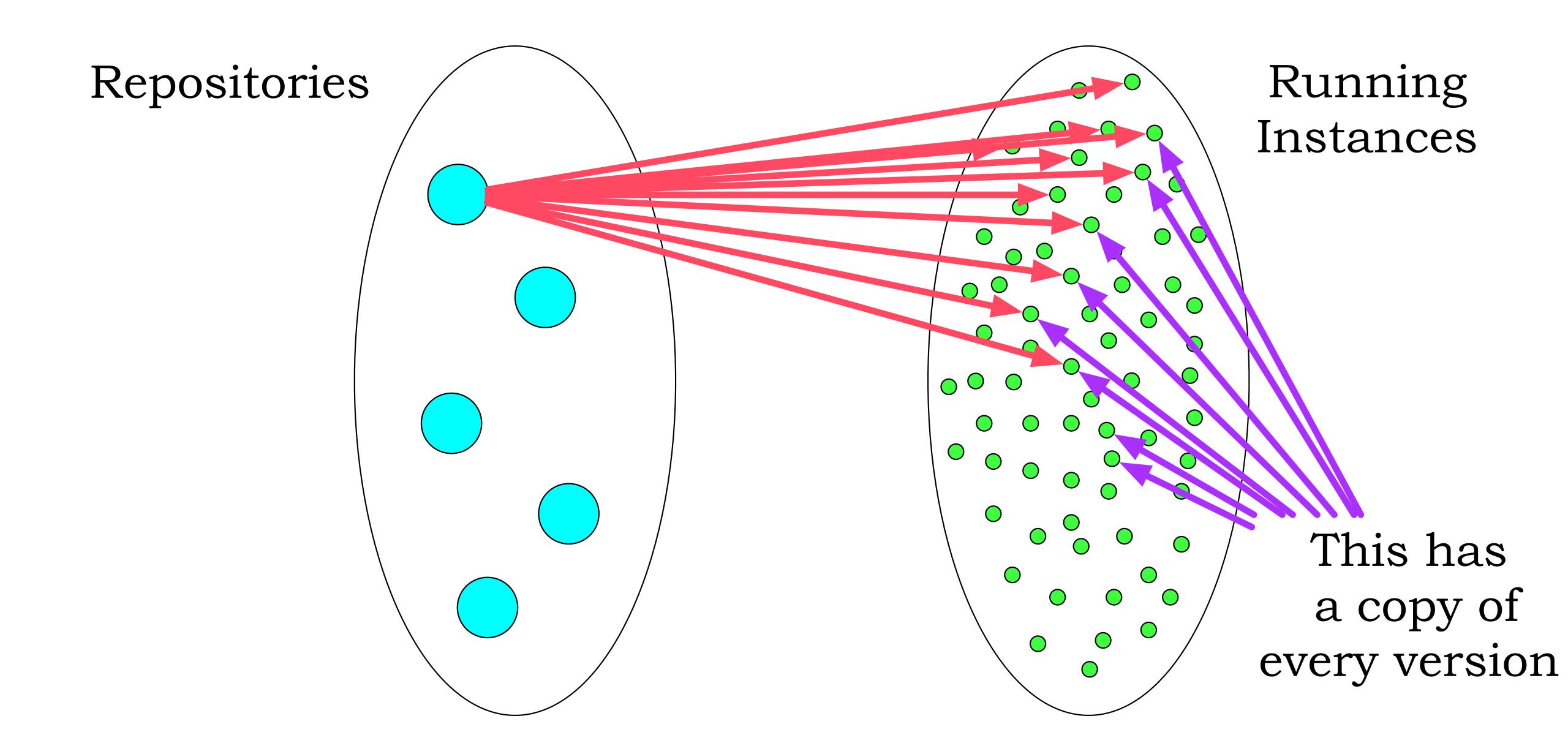




AWS Documentation » AWS Lambda » Developer Guide » Deploying Lambda-based Applications » AWS Lambda Function Versioning and Aliases » Introduction to AWS Lambda Versioning

### Introduction to AWS Lambda Versioning

Following, you can find how to create a Lambda function and publish a version from it. You can also find how to update function code and configuration information when you have one or more published versions. In addition, you can find information on how to delete function versions, either specific versions or an entire Lambda function with all of its versions and associated aliases.

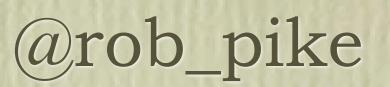


# Your Lambdas Are Your Versioned Code Repository

Independence (2)

Each developer should develop, test, and deploy Lambdas on their own AWS account

# Store the Data in the Lambda





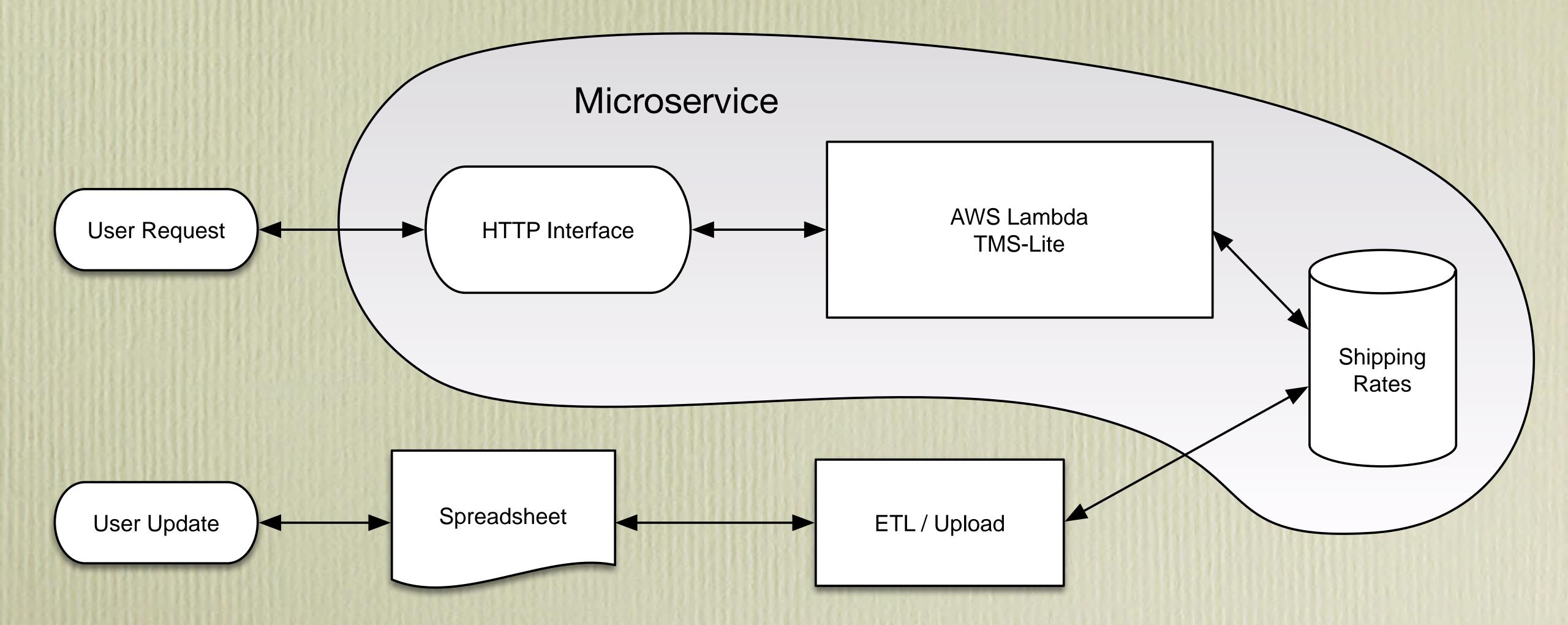
# 5 Rules of Programming

- Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
- Rule 2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
- Rule 3. *Fancy algorithms are slow when n is small, and n is usually small.* Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)
- Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
- Rule 5. *Data dominates*. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

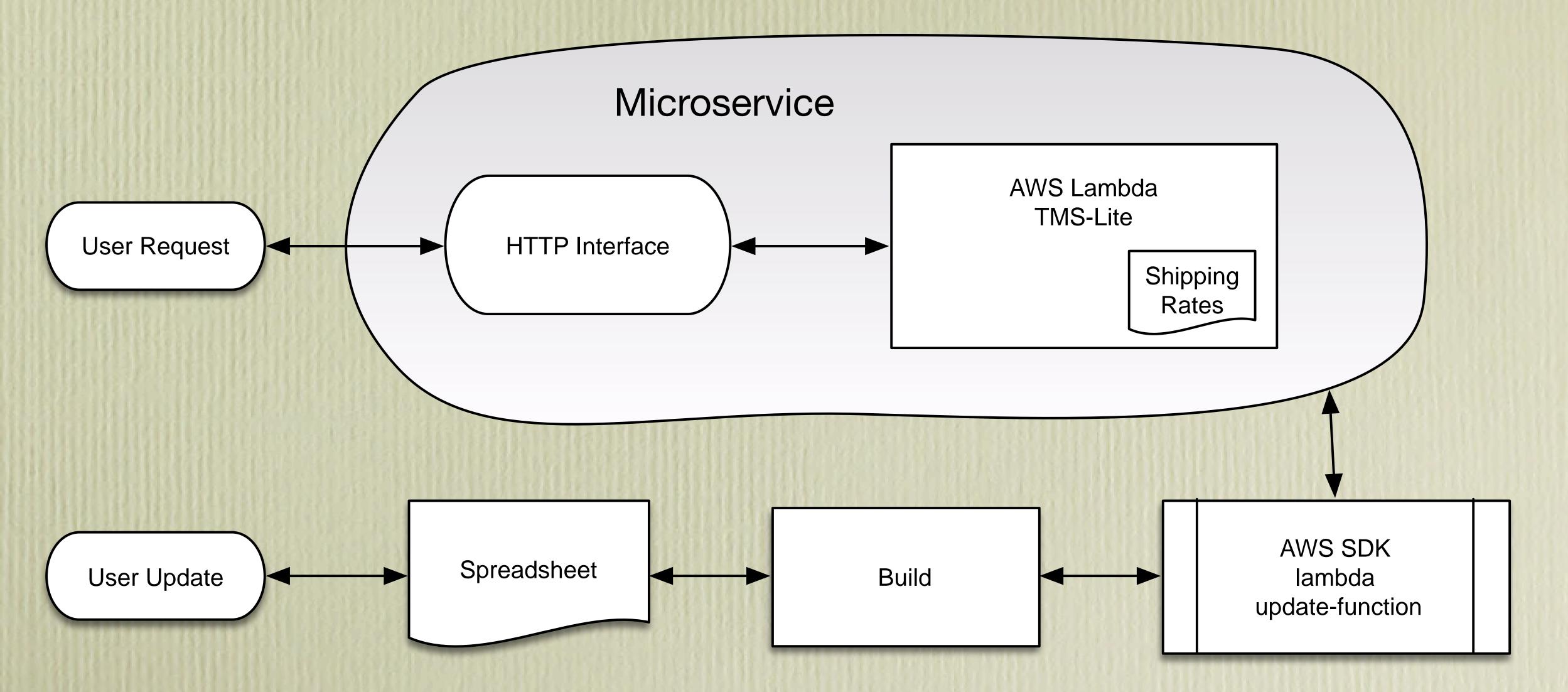
http://users.ece.utexas.edu/~adnan/pike.html

tms-lite

### Harvard



#### von Neumann



# Thinking Through the Implications of Refactoring

"The lurking suspicion that something could be simplified is the world's richest source of rewarding challenges." — Edsger Dijkstra

### @AgileAlliance

### AgileAlliance

## Refactoring

### ★ Definition

- Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.
- ★ Common Pitfalls: Refactoring does not mean:
  - rewriting code
  - fixing bugs
  - improve observable aspects of software such as its interface

https://www.agilealliance.org/glossary/refactoring/

- Adding Types
- Improving Understanding
- Reusable Design Elements
- Variations
- Collective Code Ownership

Adding Types

## Refactoring

Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behaviour preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and **aids to formal reasoning such as type systems**.

https://www.agilealliance.org/glossary/refactoring/

### aids to formal reasoning such as type systems.

- CoffeeScript ⇒ Typescript
- Python ⇒ Python with type annotations ⇒ 未定
- JSON ⇒ Protobuf (API)
- JSON ⇒ SQL Tables
- https://quicktype.io



### adabeaz

### David Beazley

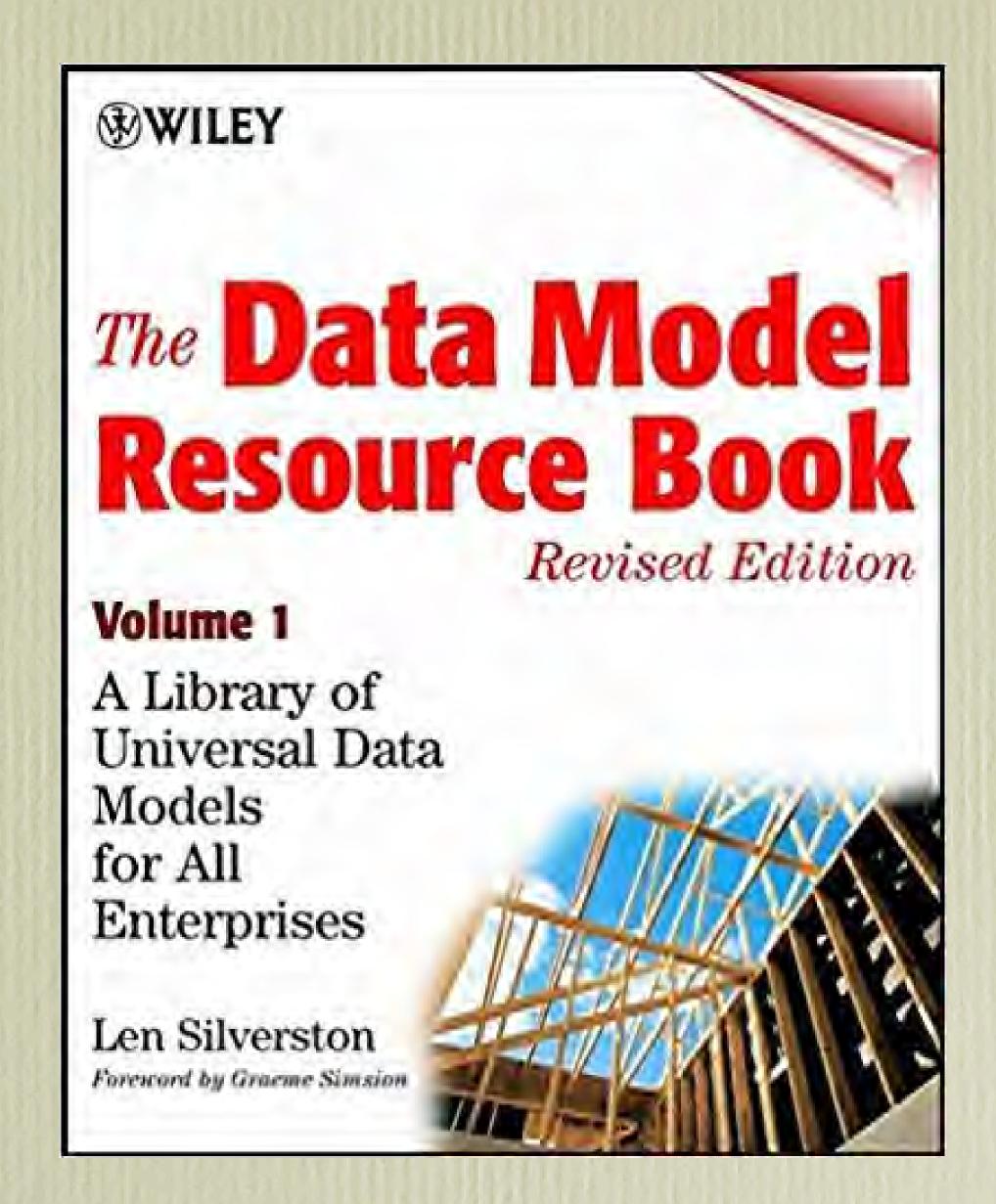
Why is it always "strongly typed" or "weakly typed?" Is there really no room for "averagely typed?"

## @JeffDean

Jeff Dean

# JSON is Tech Debt

- Low-level MapReduce interfaces are in terms of byte arrays
  - Hardly ever use textual formats, though: slow, hard to parse
  - Most input & output is in encoded Protocol Buffer format



# Understanding

## @AgileAlliance

## AgileAlliance

## Refactoring: Expected Benefits

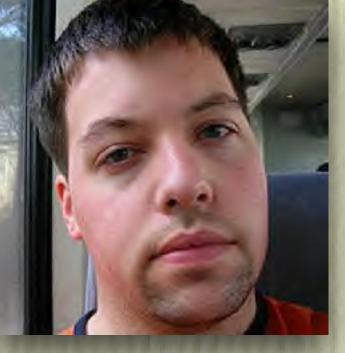
- refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
- refactoring helps code understanding
- refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
- refactoring favors the emergence of reusable design elements (such as design patterns) and code modules

https://www.agilealliance.org/glossary/refactoring/

## @houshuang

#### Stian Håklev

```
#!/usr/bin/env python
# File name: while.twpy
number = 23
running = True
while running:
    guess = int(raw input('Enter an integer : '))
    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False # this causes the while loop to stop
    elif guess < number:
        print 'No, it is higher than that.'
    else:
        print 'No, it is lower than that.'
else:
    print 'The while loop is over'
print 'Done'
```



### @houshuang

#### Stian Håklev

```
#!/usr/bin/env zhpy
# 檔名: while.py
數字 = 23
運行 = 真
當 運行:
   猜測 = 整數(輸入('輸入一個數字: '))
   如果 猜測 == 數字:
      印出'恭喜,你猜對了.'
      運行 = 假 # 這會讓循環語句結束
   假使 猜測 < 數字:
      印出'錯了,數字再大一點.'
   否則:
      印出'錯了,數字再小一點.'
否則:
   印出 '循環語句結束'
```

## Python In Chinese

```
# -*- coding: utf-8 -*-
# this is a sample Python program
定義 範例_1(訊息,次數):
取 數 在 range(次數):
印出 "哈囉, *s!" * 訊息

範例_1("世界", 百)
```

## **Converted Python**

```
# -*- coding: utf-8 -*-

# this is a sample Python program

def p0_1(p1,p2):
    for p3 in range(p2):
        print "哈囉, *s!" * p1

p0_1("世界", 100)
```

http://chinesepython.org

# APL vs. Python

# APL vs. Python

```
# @brief
          Metodo principal de execucao, aplica as regras a um estado
#
            da matriz
def executa_passo_jogo(self):
     lista_mudanca_estado = []
     # Verifica quais celulas devem mudar o estado (nascer/morrer),
     # agendando a mudanca para um loop posterior
     for i in range(0, self._tamanho_total):
         vizinhos = self._conta_vizinhos(i)
         # Celulas vivas so permanecem vivas com um
         # pop_baixa <= n <= pop_alta, n sendo o numero de vizinhos.
         if self._matriz_jogo[i].esta_viva():
            if (vizinhos < self._pop_baixa) or (vizinhos > self._pop_alta):
                lista_mudanca_estado.append(i)
         # Celulas mortas se tornam vivas com exatamente pop_alta vizinhos.
         else:
            if vizinhos == self._pop_alta:
                lista_mudanca_estado.append(i)
     # Aplica mudancas determinadas no estagio anterior
     for i in lista_mudanca_estado:
         self._matriz_jogo[i].muda_estado()
```

## 1986

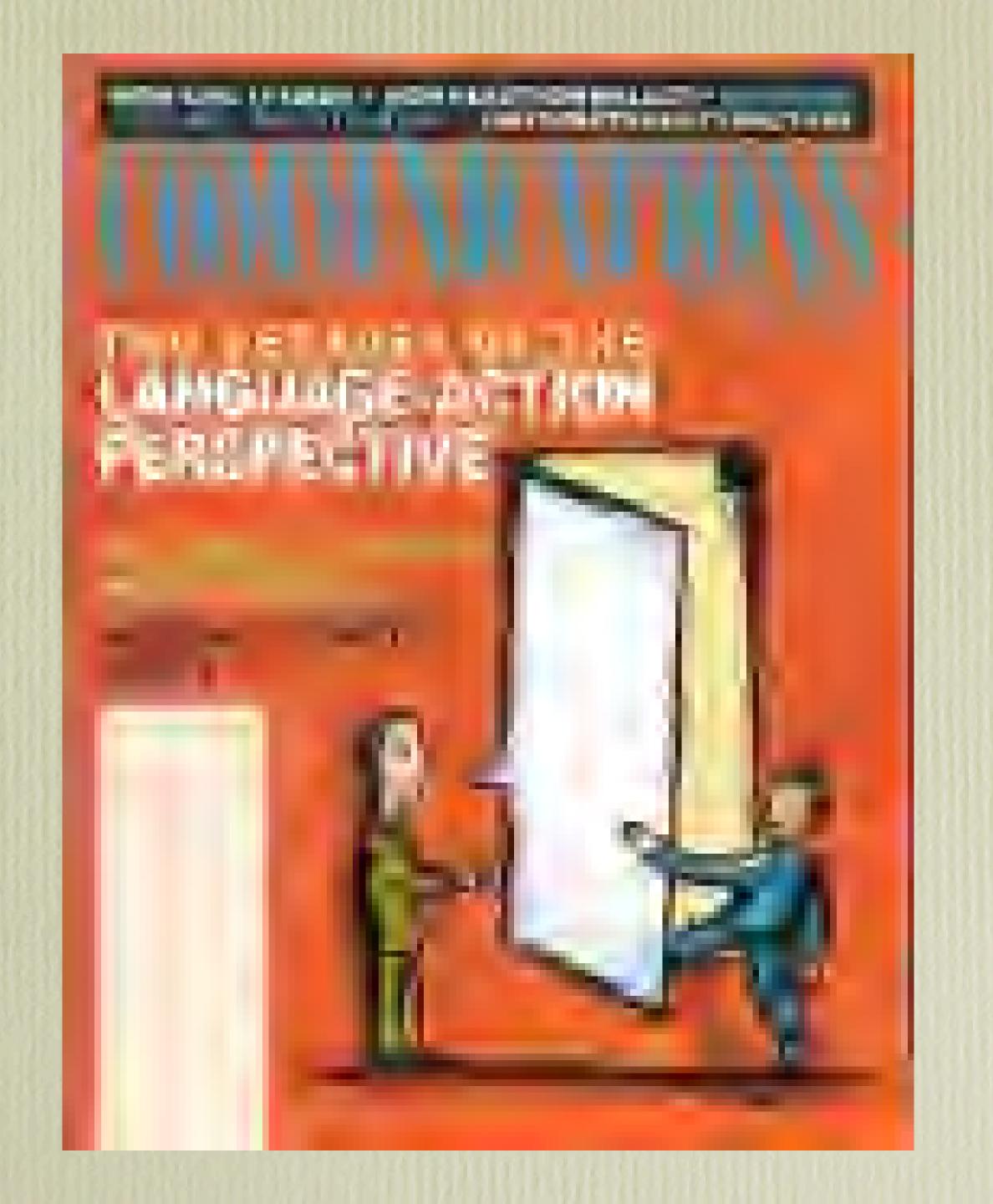
# Understanding Computers and Cognition

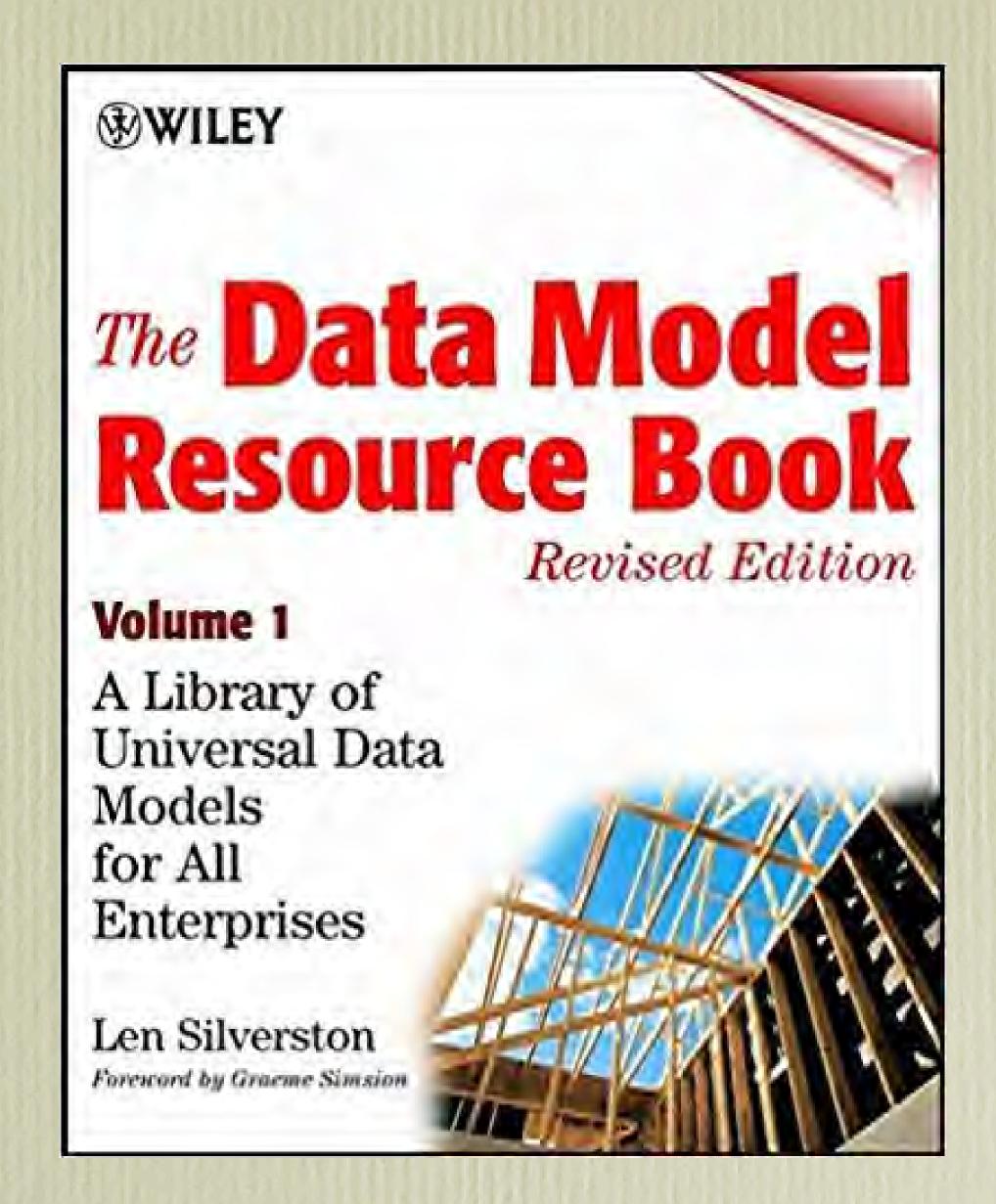
A
New
Foundation
for Design

#### TERRY WINOGRAD FERNANDO FLORES

"This is a ground breaking book.
It is about computers, artificial intelligence, language . . . it is a deeply thought out work."

— Joseph Weizenbaum, M.I.T.







Writing specs is like flossing: everybody agrees that it's a good thing, but nobody does it.

# You'll need a Technical Writer Or a Teacher

# Reusable Design Elements

Before software can be reusable, it first has to be usable.

— Ralph Johnson





# Conway's Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

http://www.melconway.com/Home/Committees\_Paper.html

- Reusable design elements will be produced only by the team whose purpose is to produce reusable design elements.
- That team may be called
  - shared services
  - infrastructure
  - brand management
  - graphic design



# Only the Giants Survive

- There can be  $n^2$  interactions among n components
- Reusable components take 3-5 times longer to write than good ones. (Good ones take 50% to 200% longer to write than mediocre ones)
- Platforms require someone taking responsibility for coherence and stability.
- Big components (over five million lines of code) work. The cost of documentation is too high for smaller components.
- Reusable components often consume 100 times the resources of custom built components. (the reason today's PC is not 10,000 times more featureful than the Xerox Alto)

Application Application Application

Your Shared Library

Platform Library

Operating System

Hardware

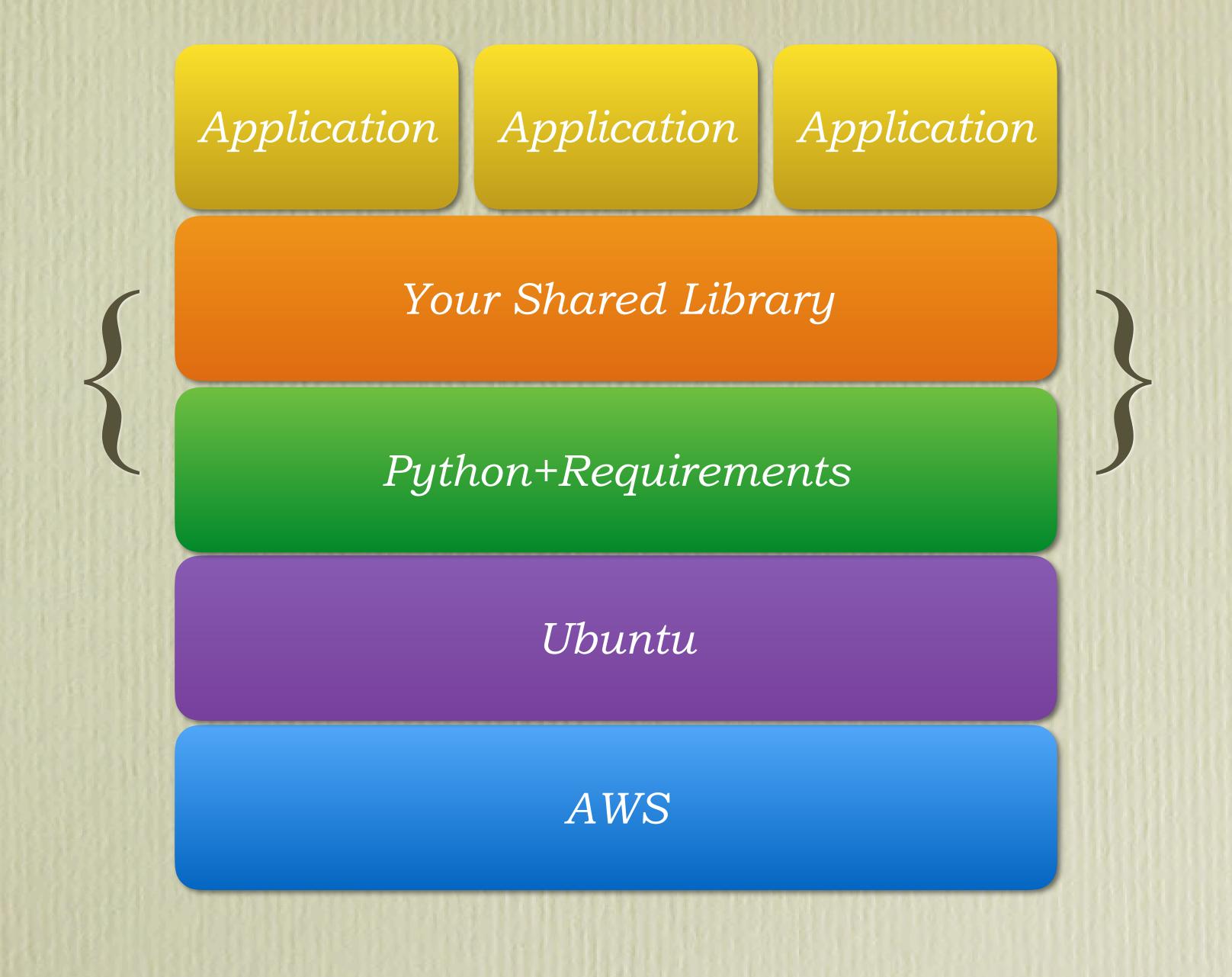
Application Application Application

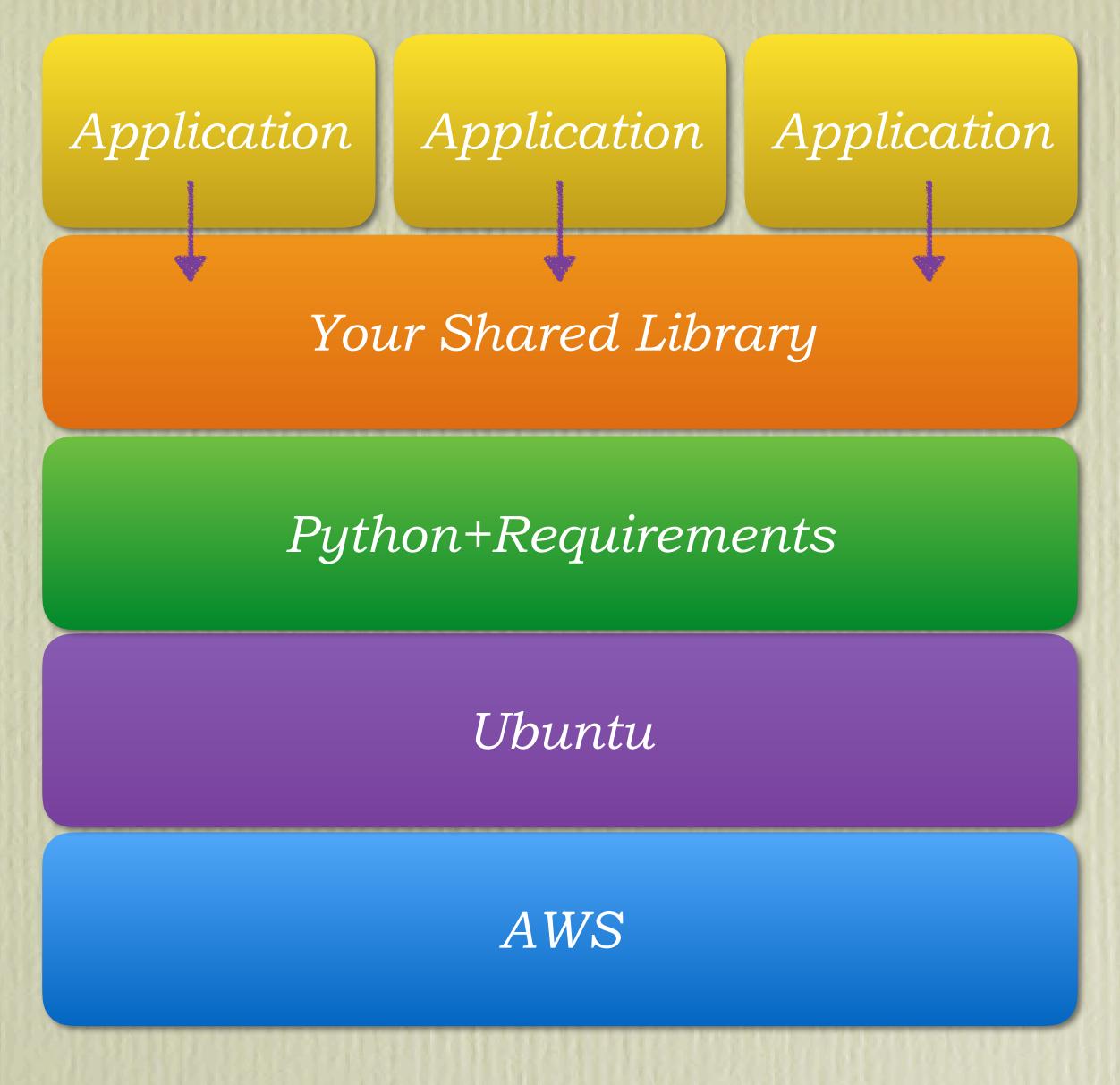
Your Shared Library

Python+Requirements

Ubuntu

AWS





Sedimentary Software

Applic Applic Applic

Your Shared Library

Python+Requirements

Ubuntu

AWS

Applic Applic Applic

Your Shared Library

UIKit

iOS

iPhone

Applic Applic Applic

Your Shared Library

React

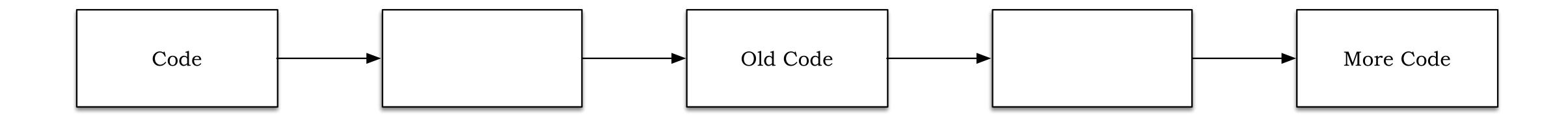
Web Browser

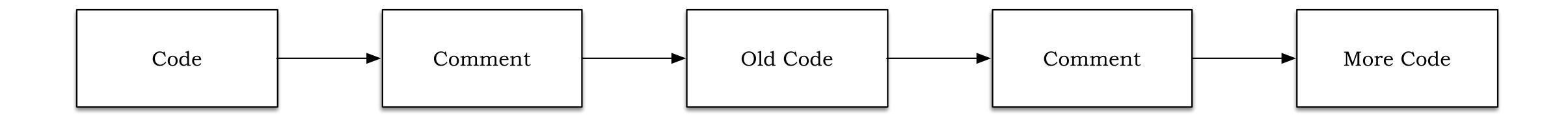
Desktop

# Variations

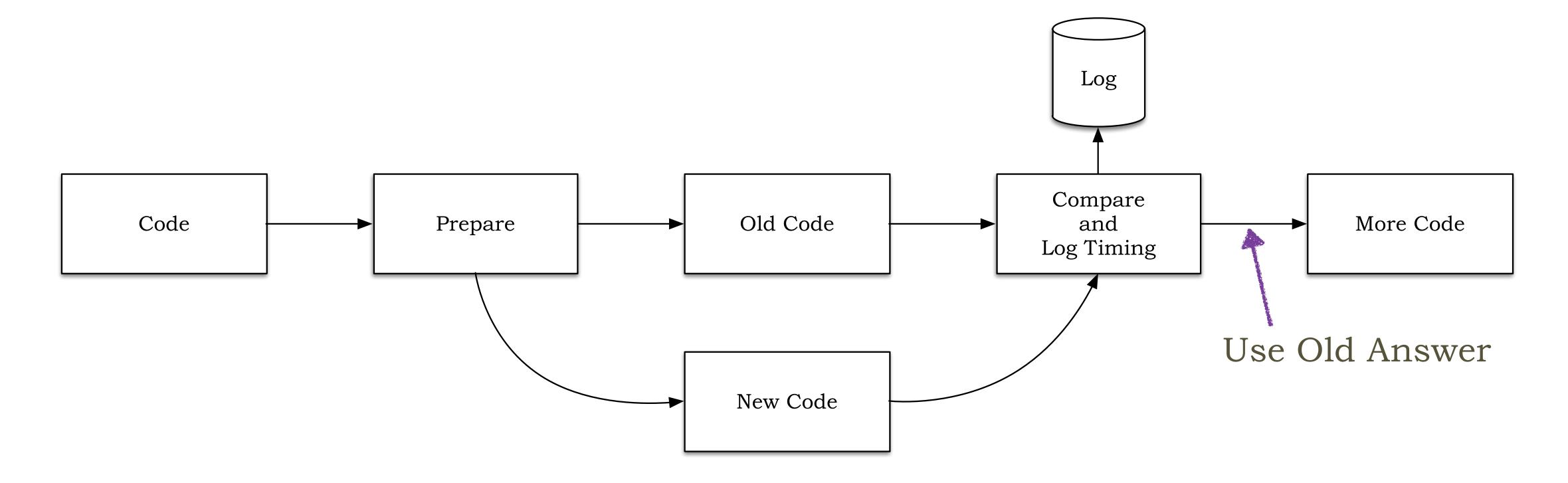


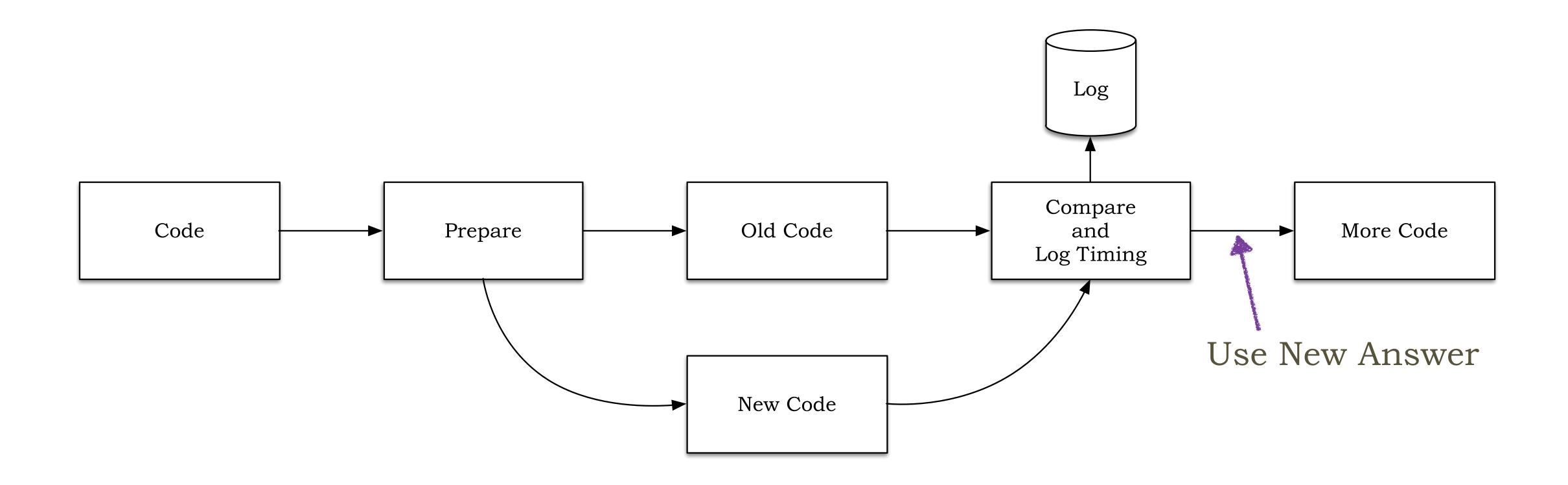
"One accurate measurement is worth a thousand expert opinions." — Grace Hopper

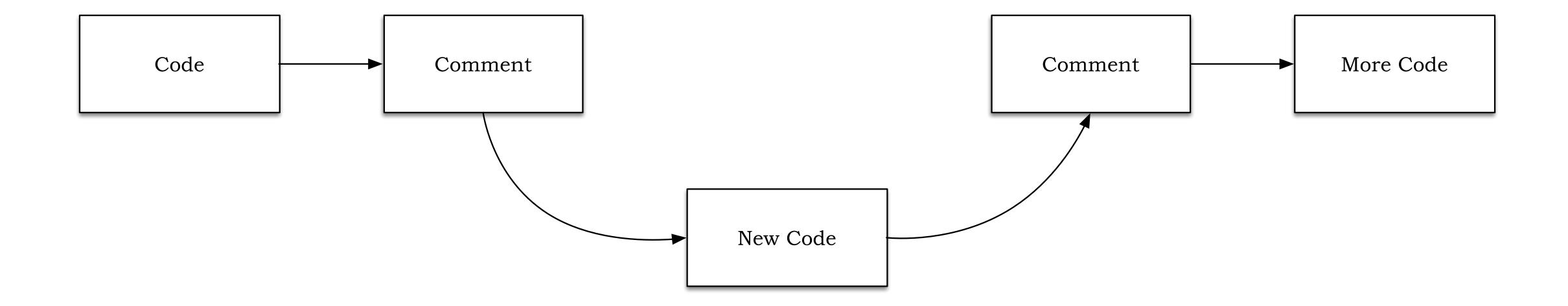




#### The Variation Pattern







## @AgileAlliance

### AgileAlliance

## Refactoring: Expected Benefits

- refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
- refactoring helps code understanding
- refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
- refactoring favors the emergence of reusable design elements (such as design patterns) and code modules

https://www.agilealliance.org/glossary/refactoring/

# Collective Code Ownership

- "collective code ownership" is the organizational manifestation of "monolith"
- if everybody must be prepared to maintain any part of the collective code base, as the code base scales, the cognitive load for each developer increases
- if the code base is partitioned so that only some people must be prepared to maintain certain sections, then the ownership is no longer collective, but partitioned

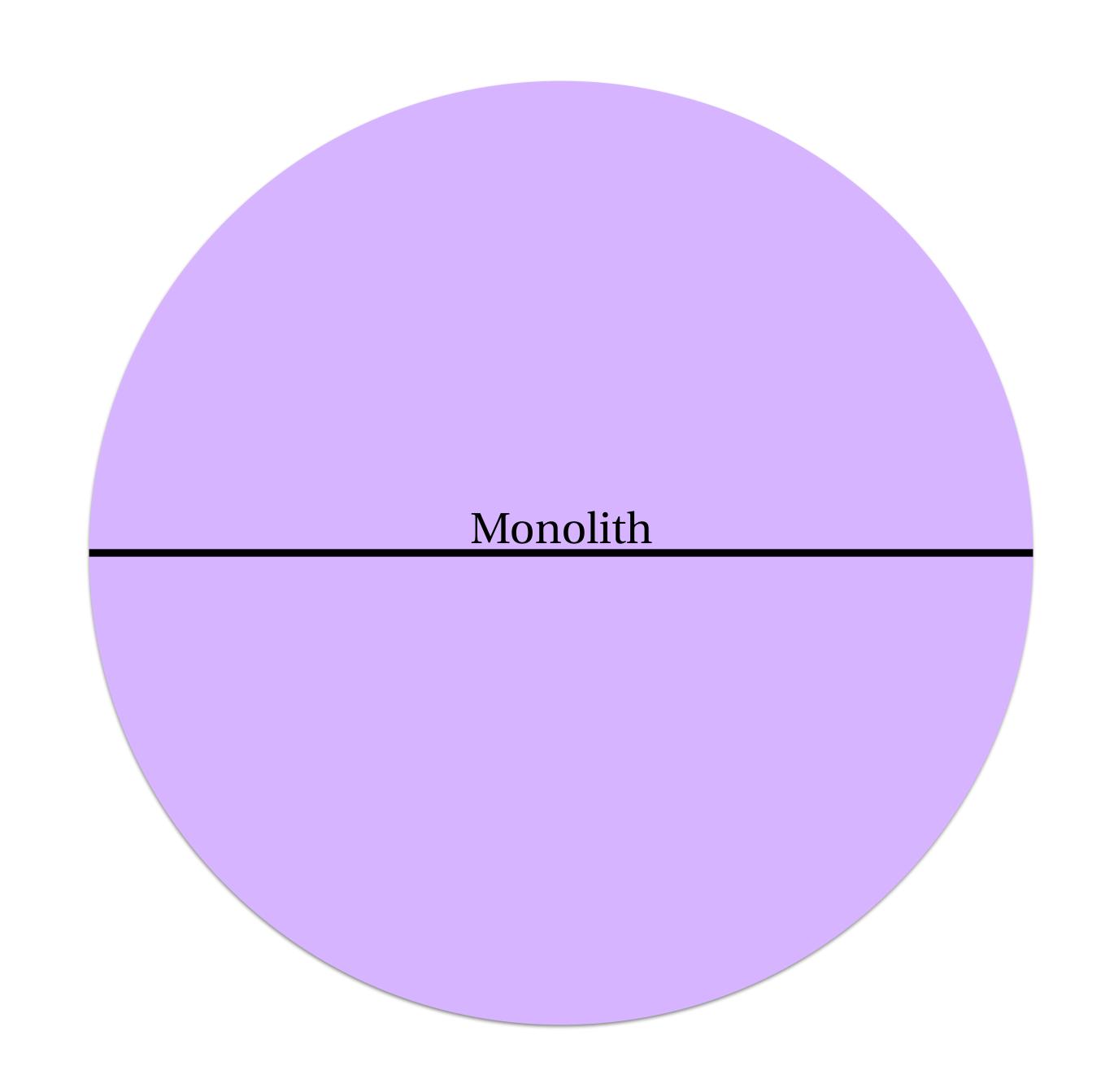


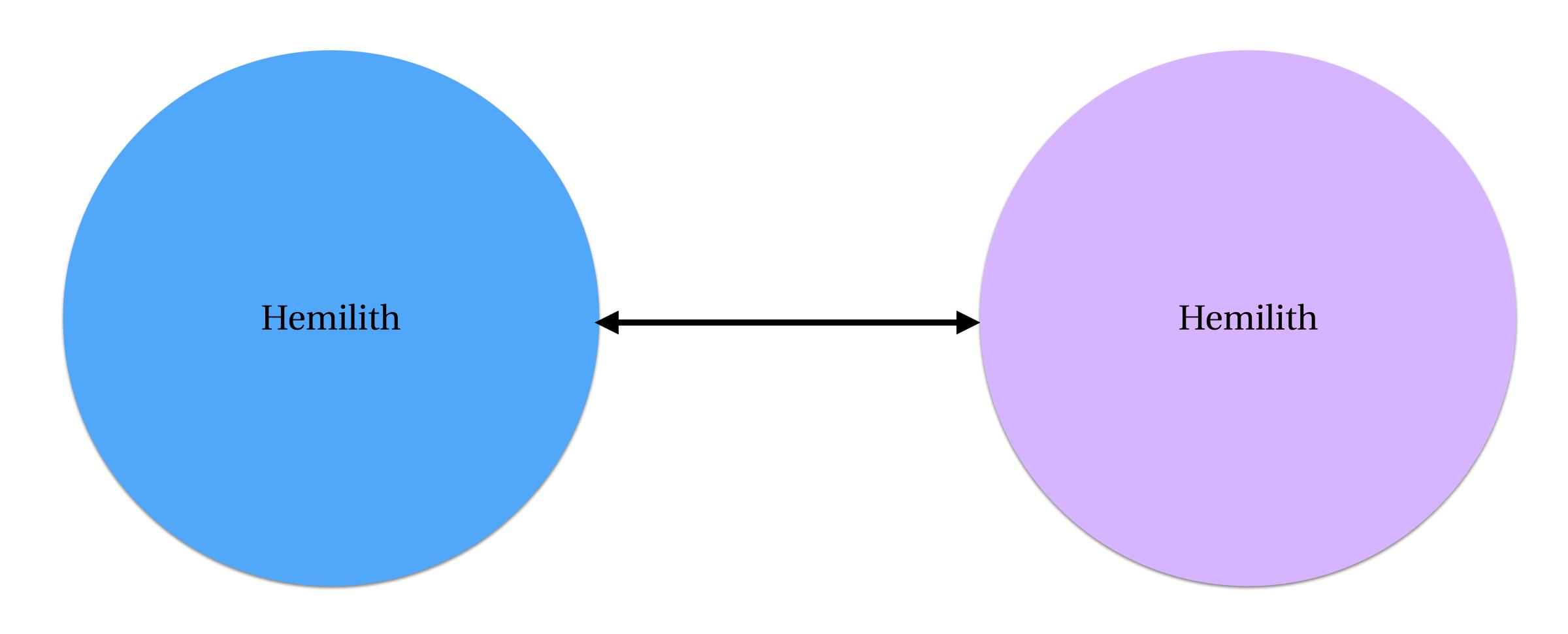


# Conway's Law

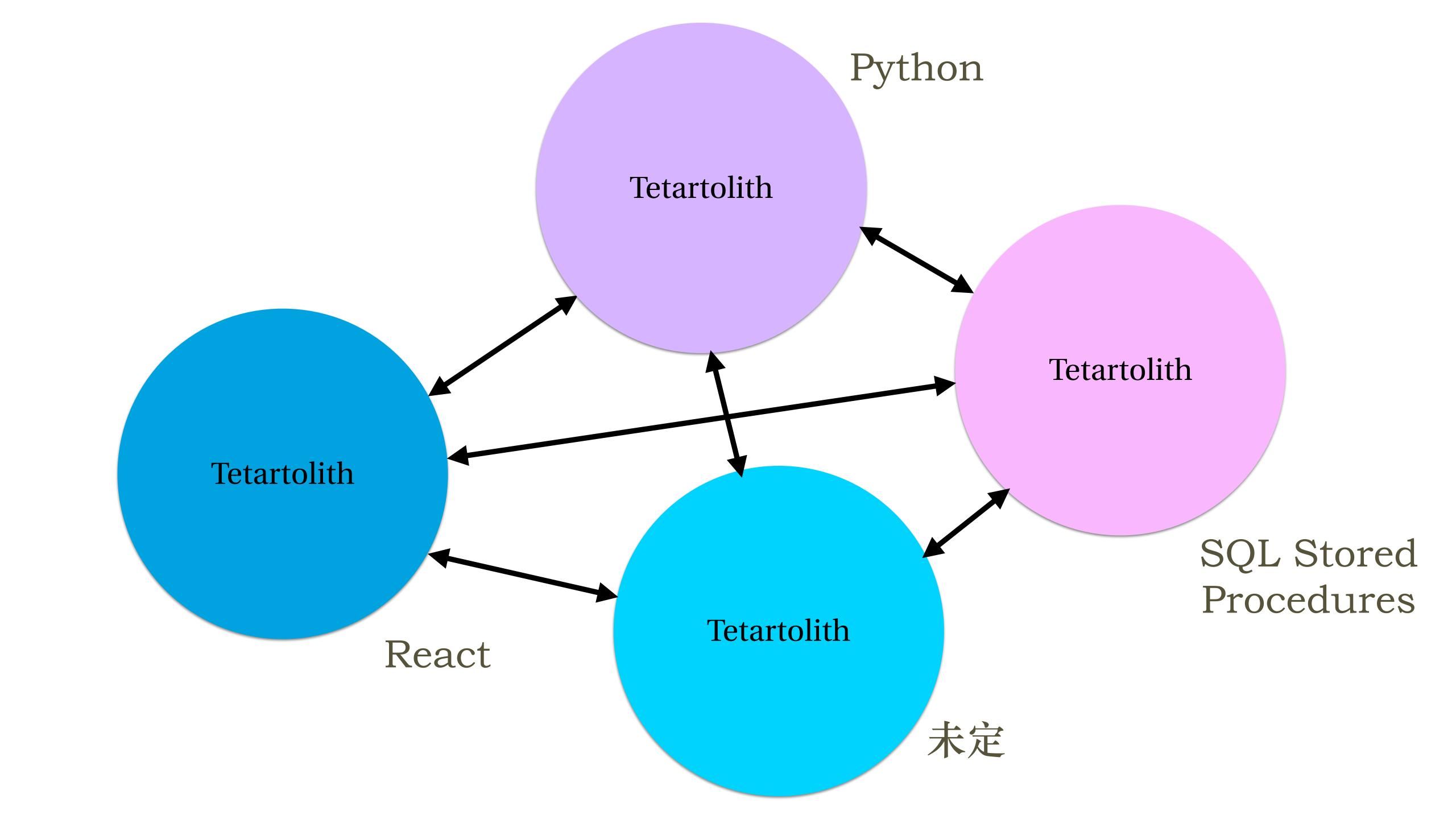
Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

http://www.melconway.com/Home/Committees\_Paper.html





Front End CoffeeScript Back End Python



## Collective Code Ownership: Configuration



# The Case Against Configuration

Robert M. Lefkowitz



## adanluu

#### Dan Luu

## Lessons Learned from Reading Postmortems

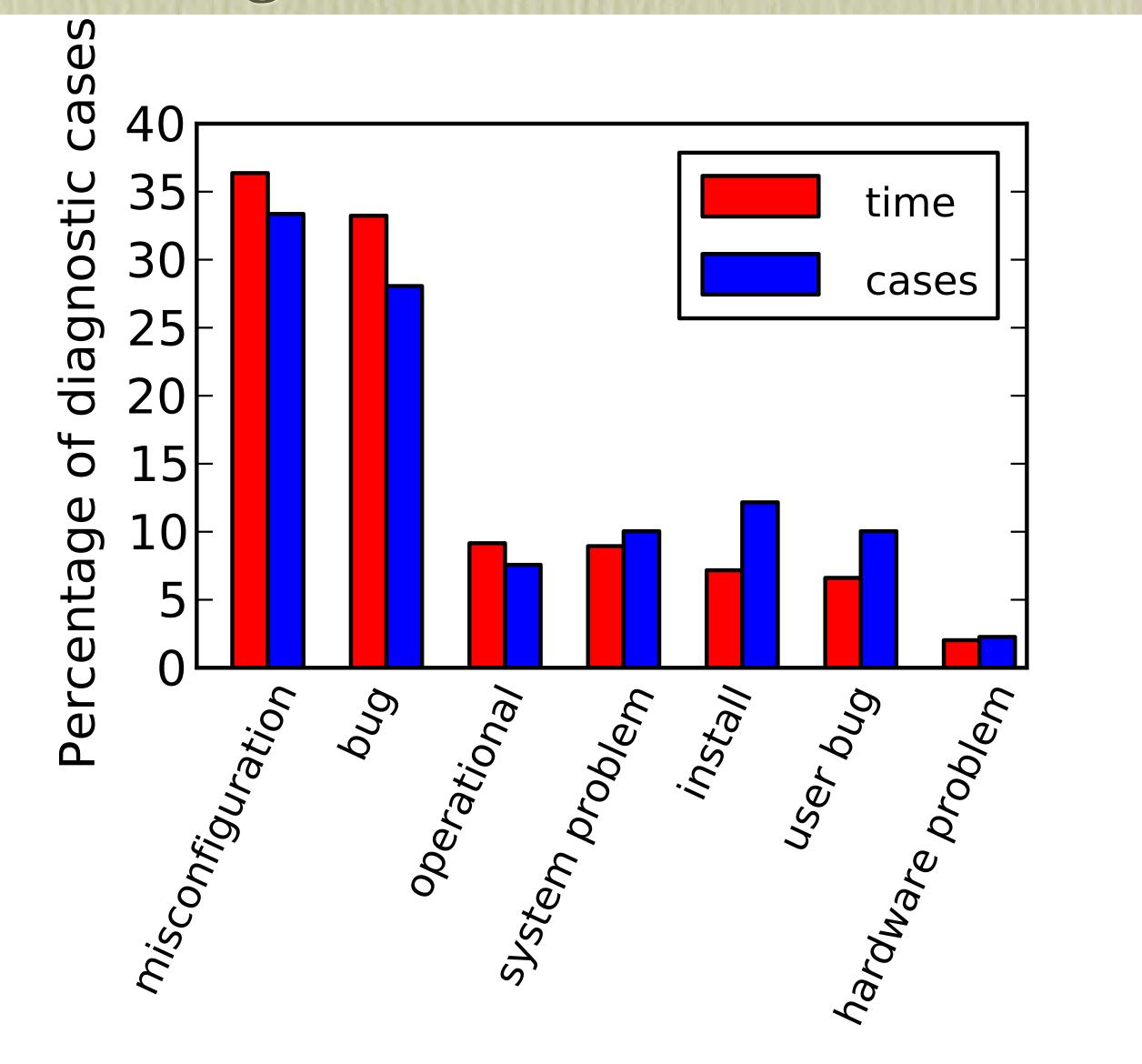
Configuration bugs, not code bugs, are the most common cause I've seen of really bad outages. When I looked at publicly available postmortems, searching for "global outage postmortem" returned about 50% outages caused by configuration changes.

http://danluu.com/postmortem-lessons/



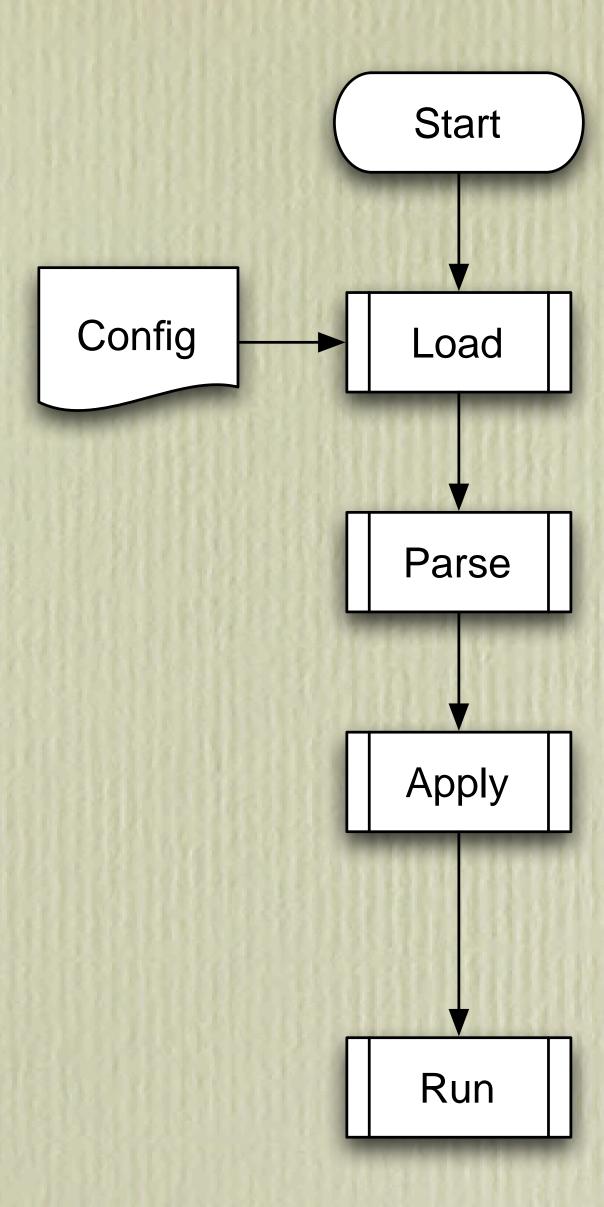
#### @AriTalking

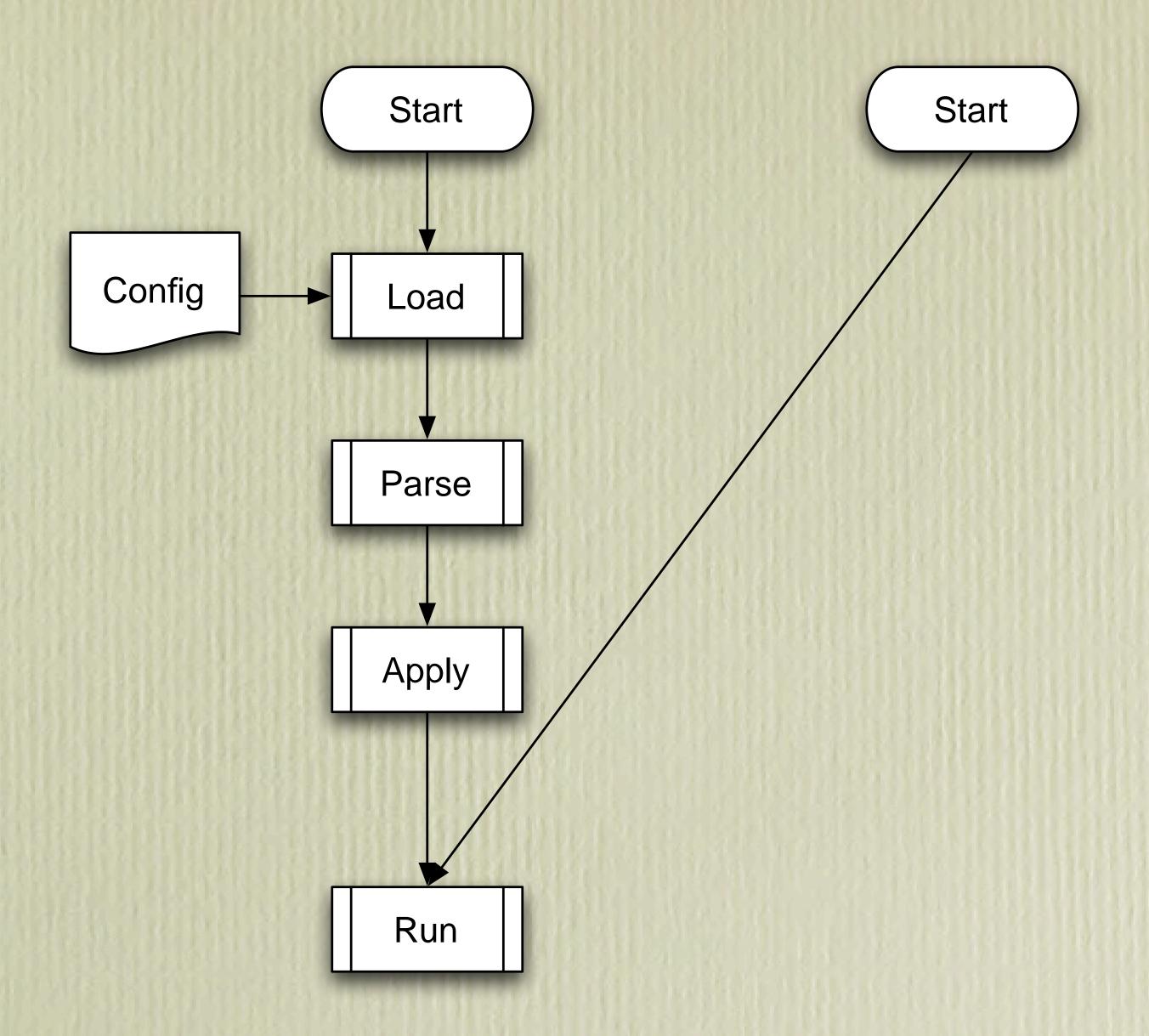
#### Ariel Rabkin



• In a world where software is proprietary, how can you modify its behavior for different computing environments?

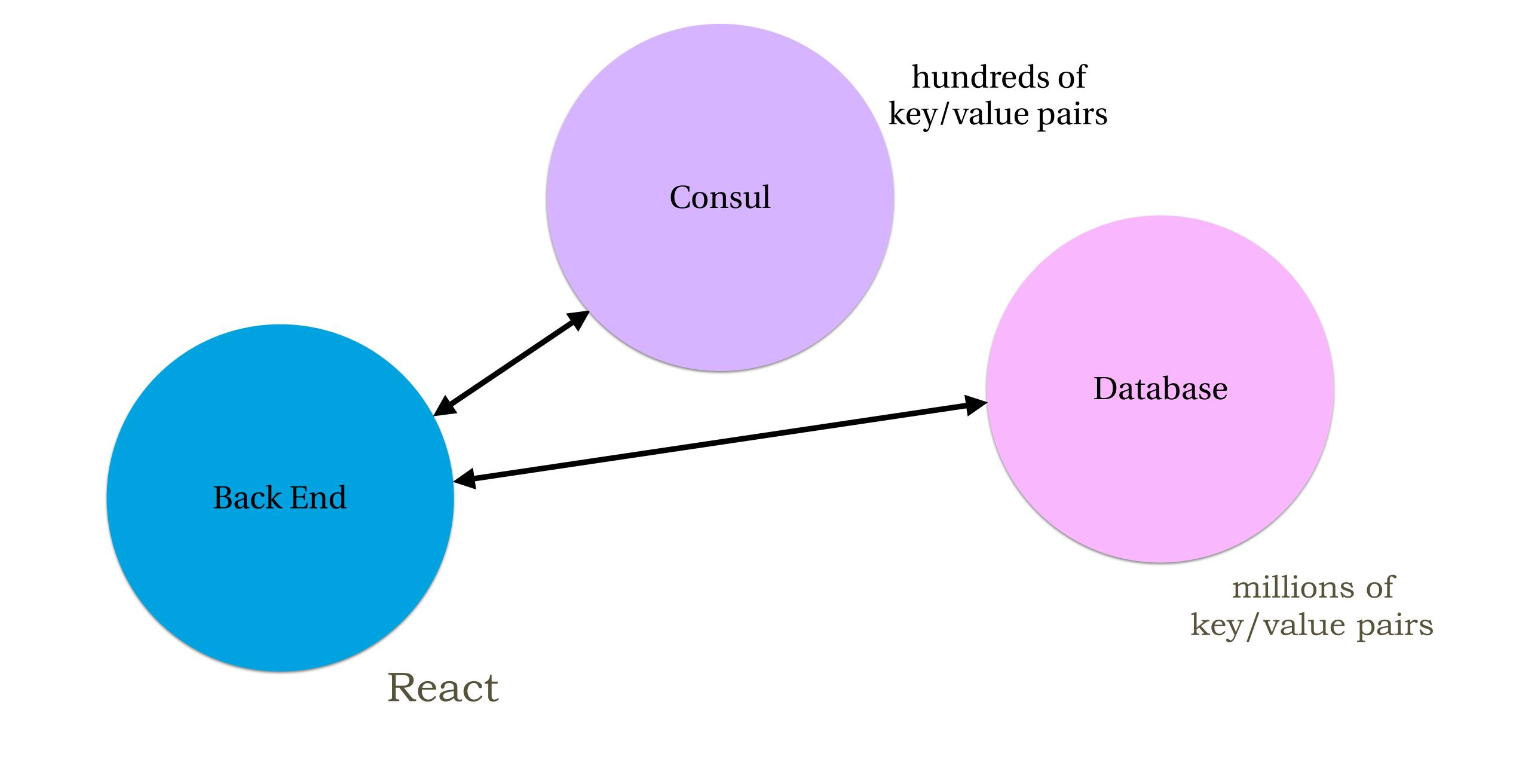
• Data files containing configuration information.

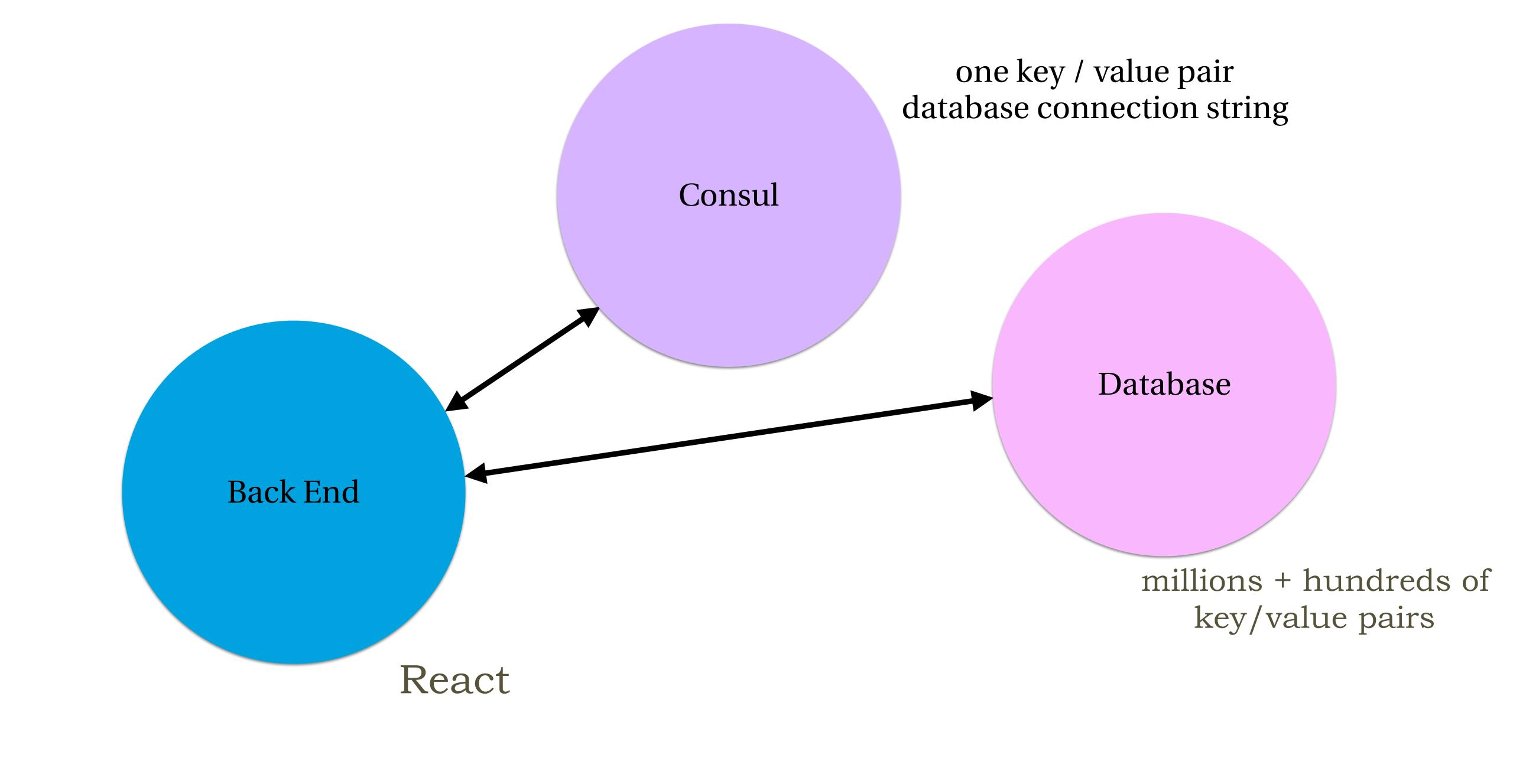


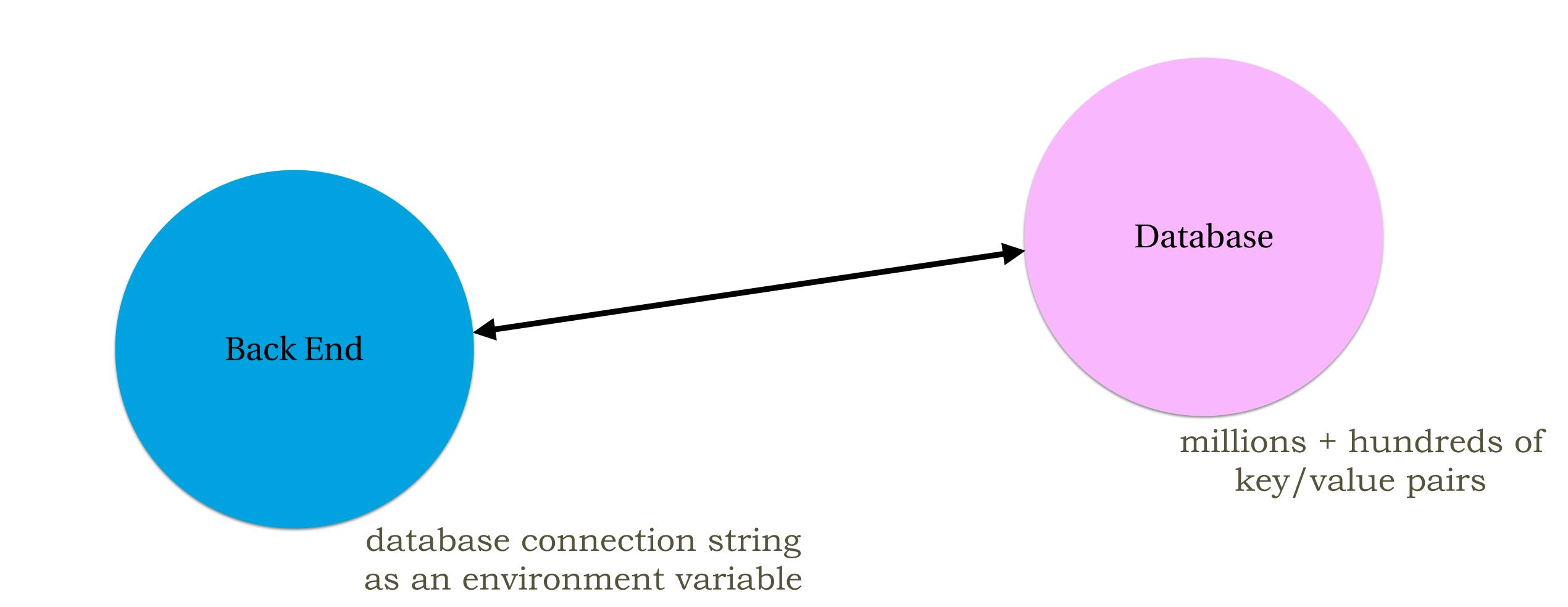


## Configuration files are inferior to code...

- Linguistically
- Editability
- Auditability
- Functionality
- Deployability







## Collective Code Ownership: Dependencies

## @AgileAlliance

#### AgileAlliance

## Refactoring: Expected Benefits

- refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
- refactoring helps code understanding
- refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
- refactoring favors the emergence of reusable design elements (such as design patterns) and code modules

https://www.agilealliance.org/glossary/refactoring/

## Eliminate Dependencies

```
$ wc requirements.txt
160 178 2777 requirements.txt
```

\$ wc package.json
145 311 4661 package.json



## Only the Giants Survive

- There can be  $n^2$  interactions among n components
- Reusable components take 3-5 times longer to write than non-reusable ones. (Good ones take 50% to 200% longer to write than mediocre ones)
- Platforms require someone taking responsibility for coherence and stability.
- Big components (over five million lines of code) work. The cost of documentation is too high for smaller components.
- Reusable components often *consume 100 times the resources of custom built components*. (the reason today's PC is not 10,000 times more featureful than the Xerox Alto)

## Questions?

@r0ml