Shared Nothing Architecture

How to scale to infinity

https://intertwingly.net/slides/2024/

Sam Ruby, Saturday August 24th, 2024

Just imagine…

What if you could give **every** user of your software the *experience* of having a **dedicated server machine** assigned to them?

Sam Ruby

Brief History

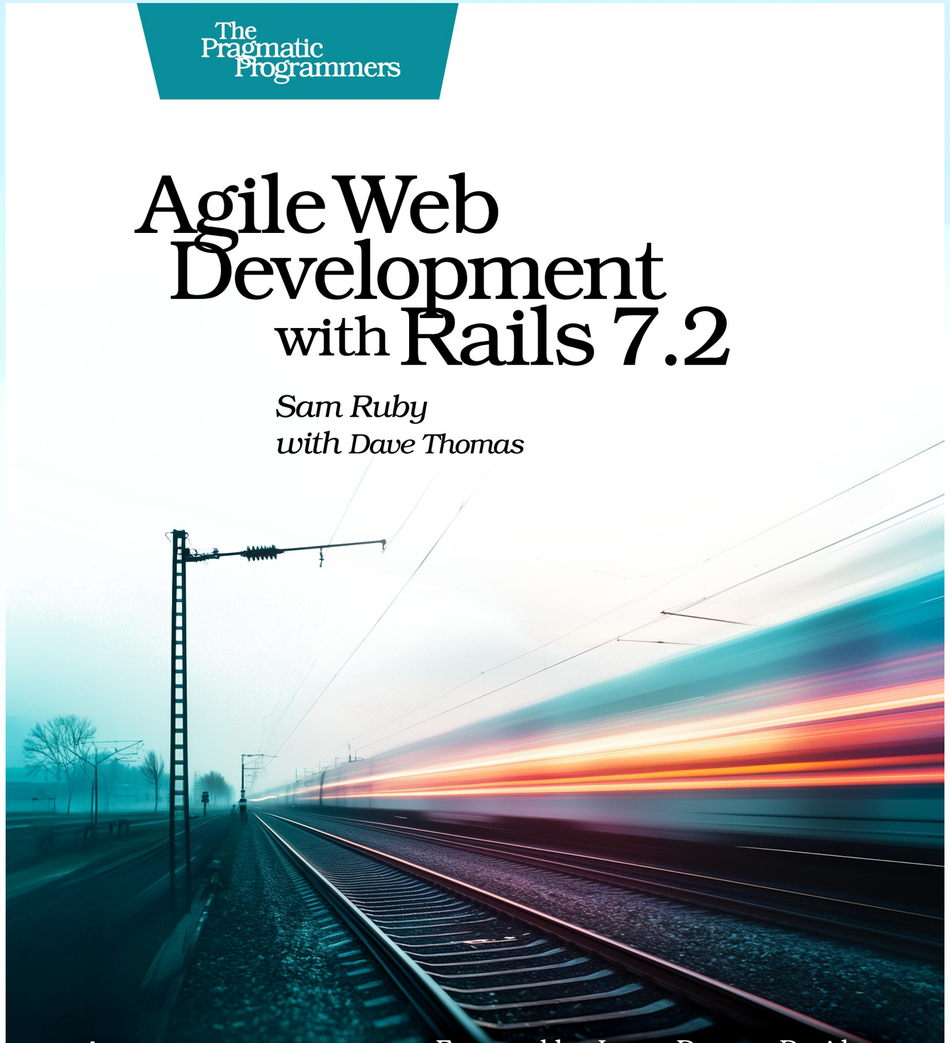
- Retired June 2020
  - 39 years at IBM
  - Wikipedia
- Wrote an app March 2022
  - https://github.com/rubys/showcase
- Unretired August 2022
  - Fly.io

Introduction

Part 0

Introduction

So, I wrote a book…



Introduction

… revised it frequently

… RailsWorld is in September, when Rails 8 is expected to ship.

Introduction

I work for…

Fly.io

Introduction

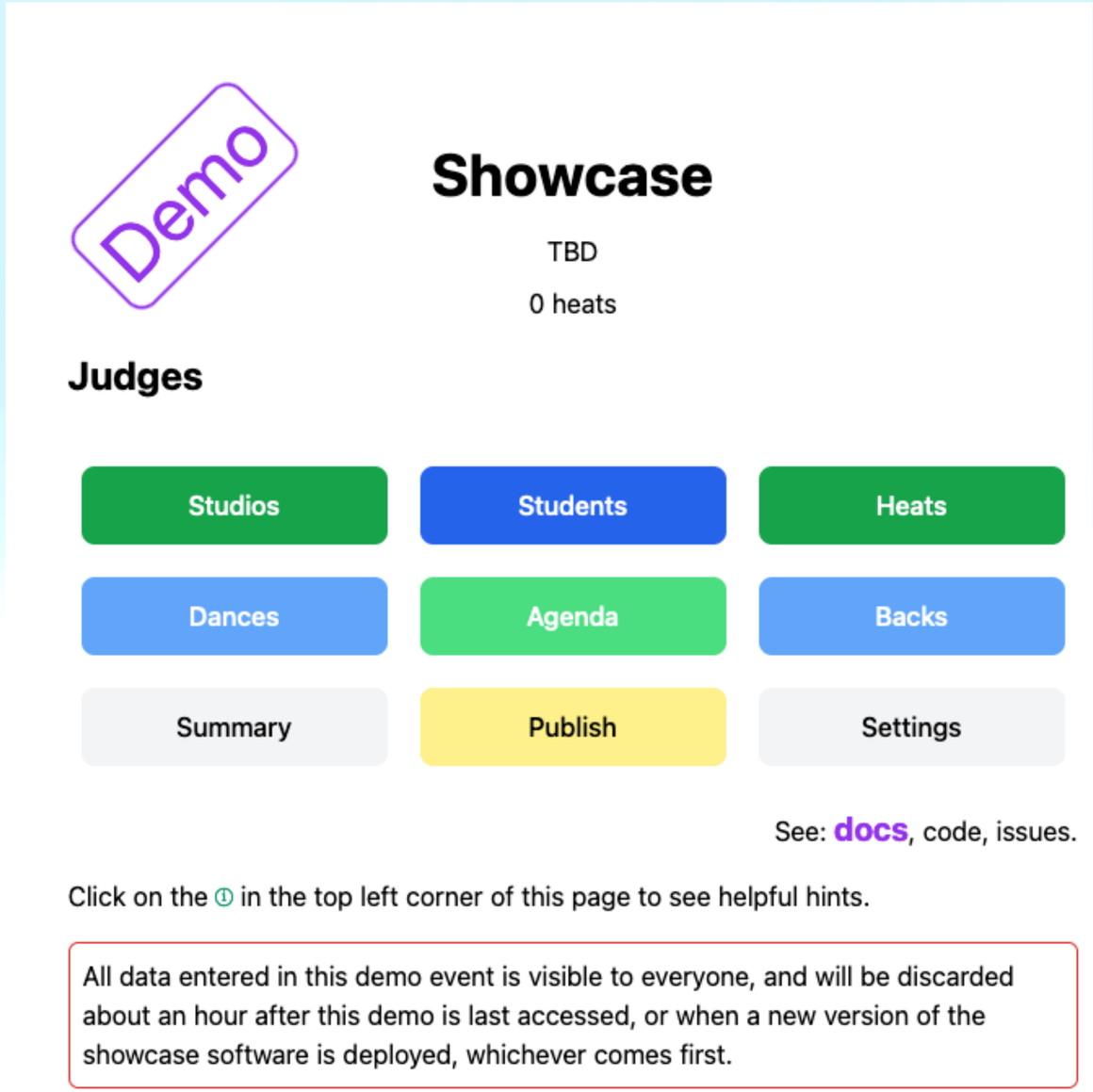
I wrote an app…

smooth.fly.dev

Showcase Application

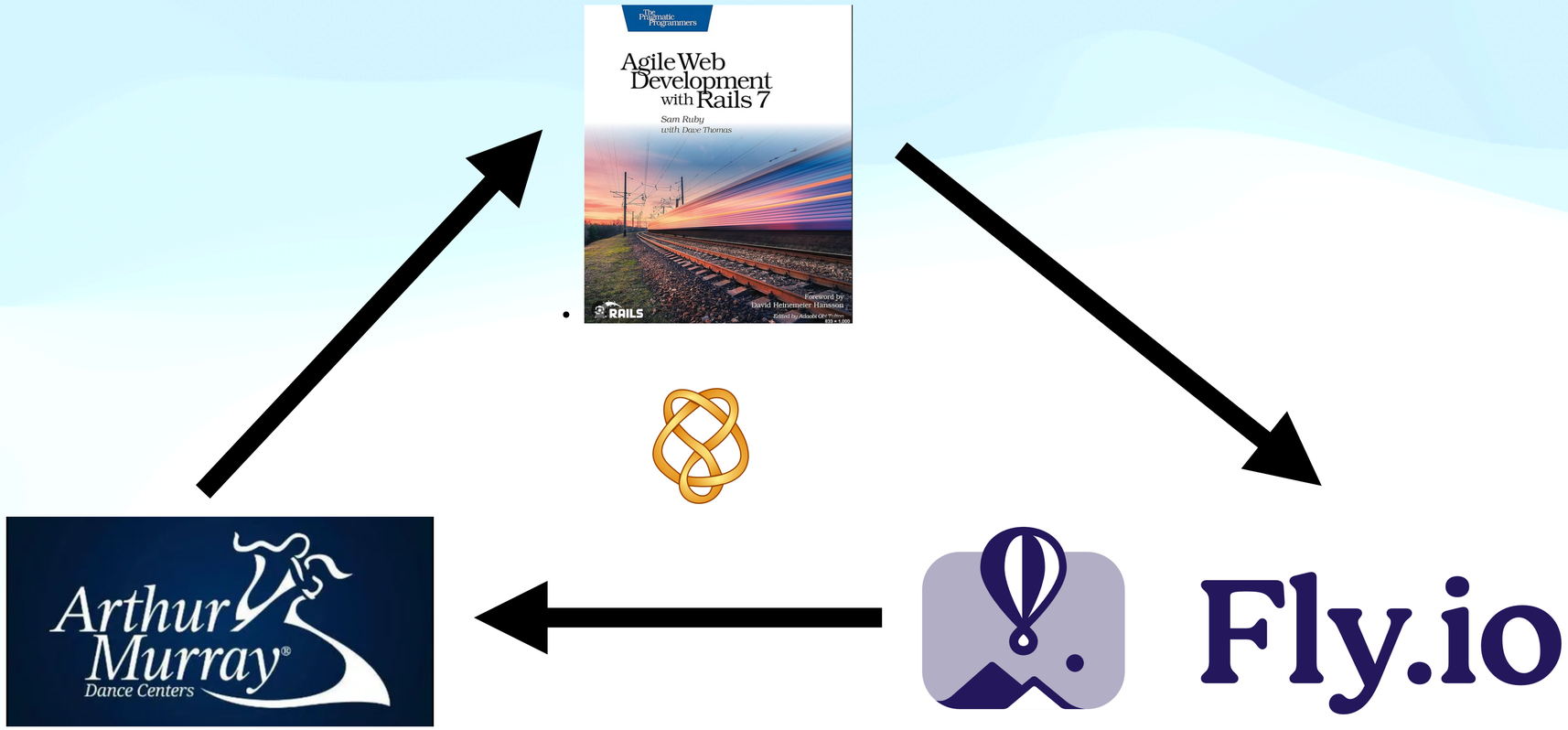
smooth.fly.dev

- Application is live, feel free to explore
- Events understandably require authentication
- Fully functional demo - click on Demo to start
- Documentation and source code are available



Introduction

… all three are all interrelated



Introduction

Shared Nothing Architecture

- My app is written primarily in Ruby on Rails, but yours need not be.
- My app is hosted on fly.io, but yours need not be.
- My app schedules ballroom dancing events, but yours may do something different.

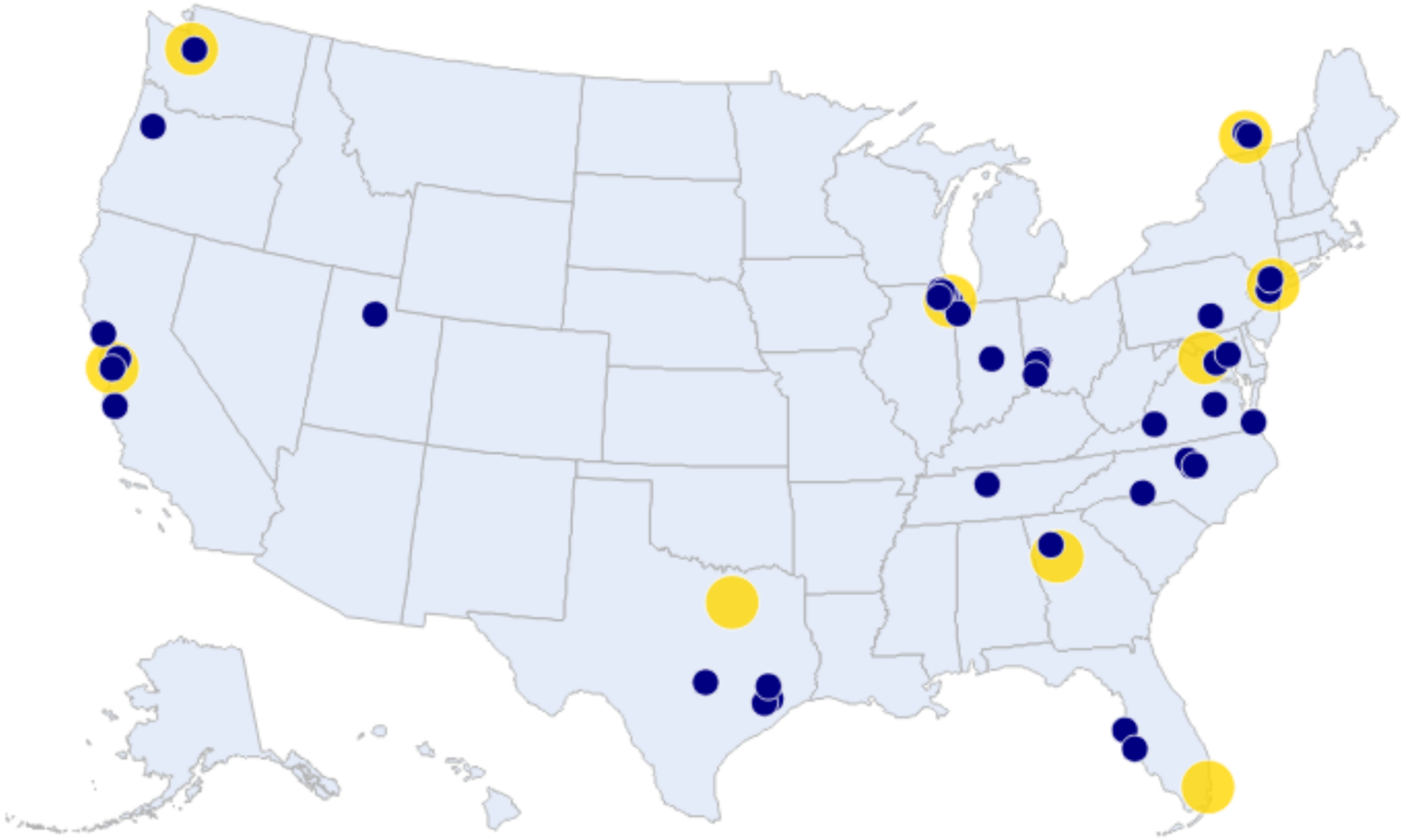
My app is just for illustrative purposes. I don’t get paid on commission.

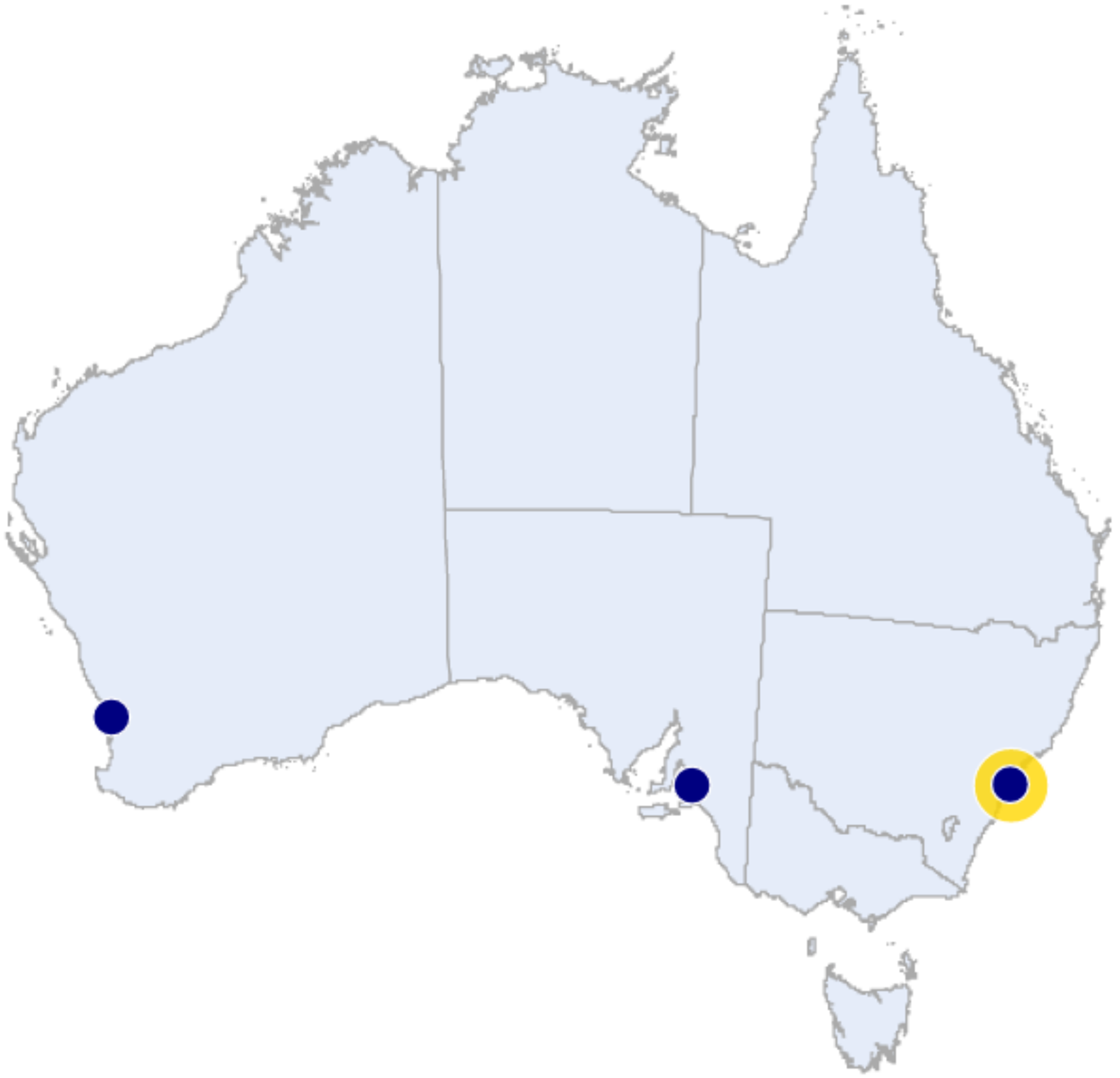
Defining the Problem

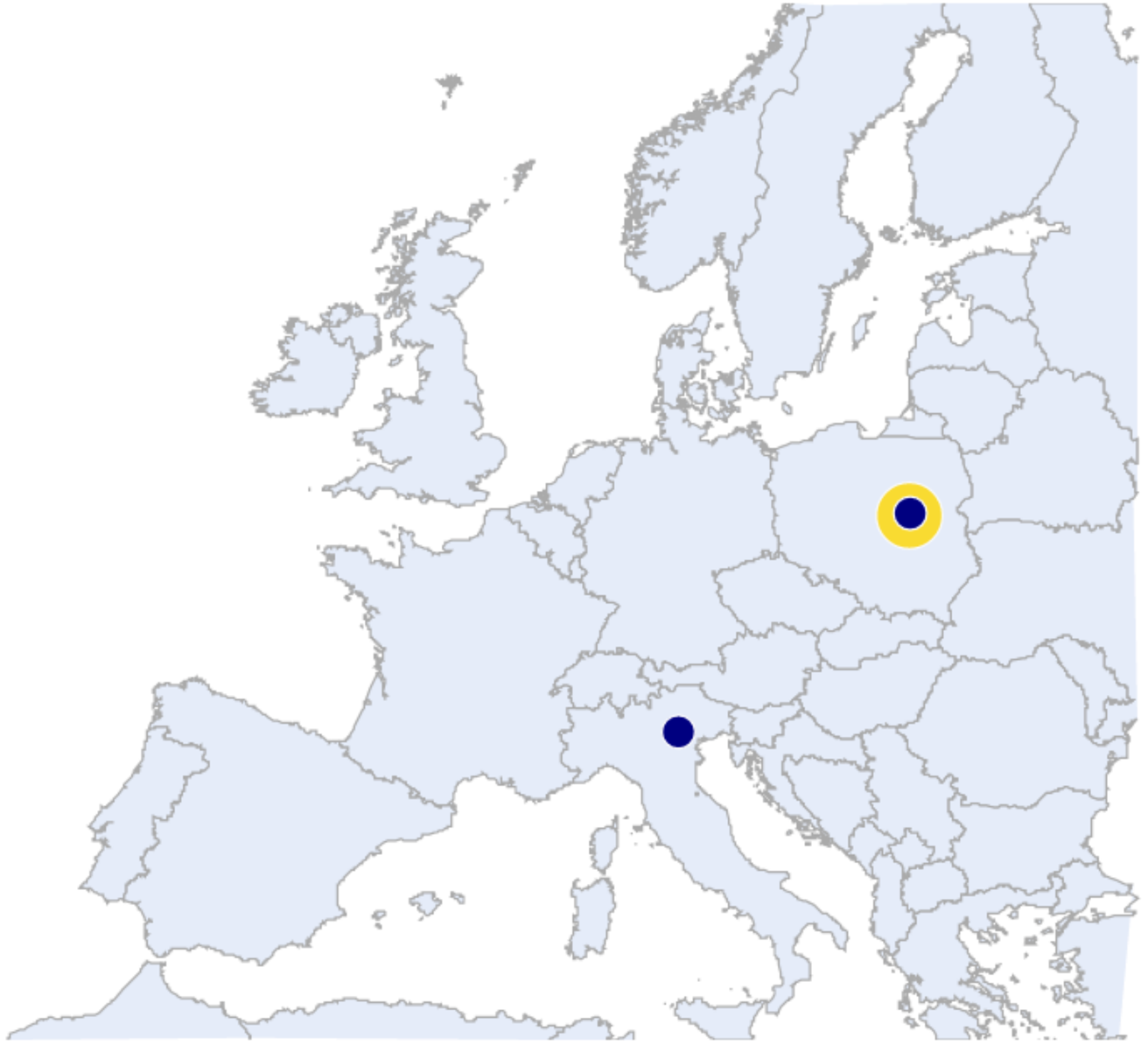
Part 1

Users of my app

	Events	Cities	Countries	Continents
2022	8	6	1	1
2023	30	20	3	2
2024 (YTD)	90	45	5	3







Statement of the problem

… these are good problems to have …

- Exponential growth - tripling each year
- Users geographically distributed
- Apps “want” to be near users
- Apps “want” to be near data

Planning for growth

- What design choices can we make today to prepare for a future where there are a thousand events geographically dispersed around the globe?
  - This conceivably could happen in as little as two to three years
- What engineering tradeoffs would be required to pull this off?
  - You can’t magically make problems go away completely, but you can make them smaller and more manageable

Establishing the baseline

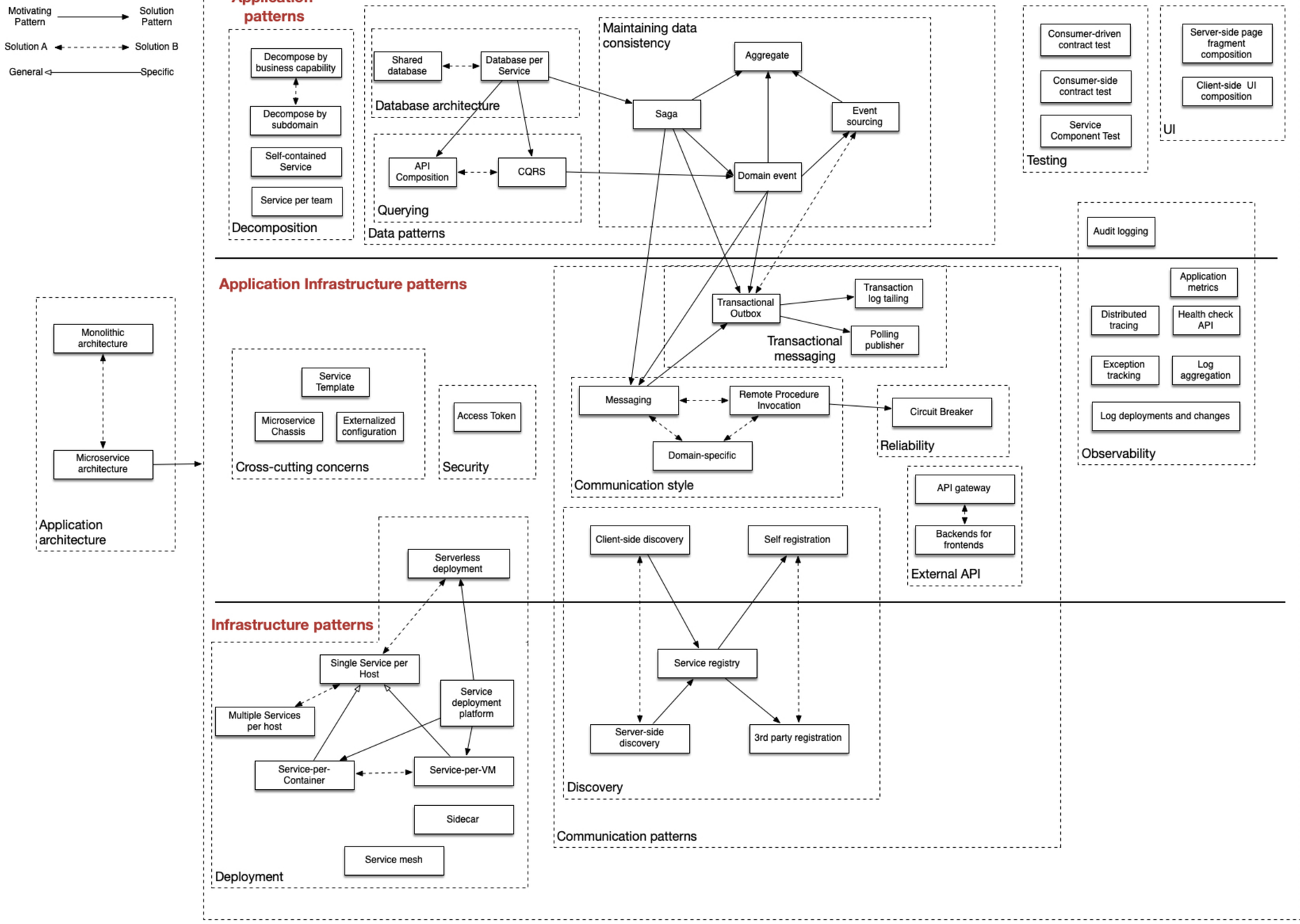
Part 2

Establishing the Baseline

K8s and microservices

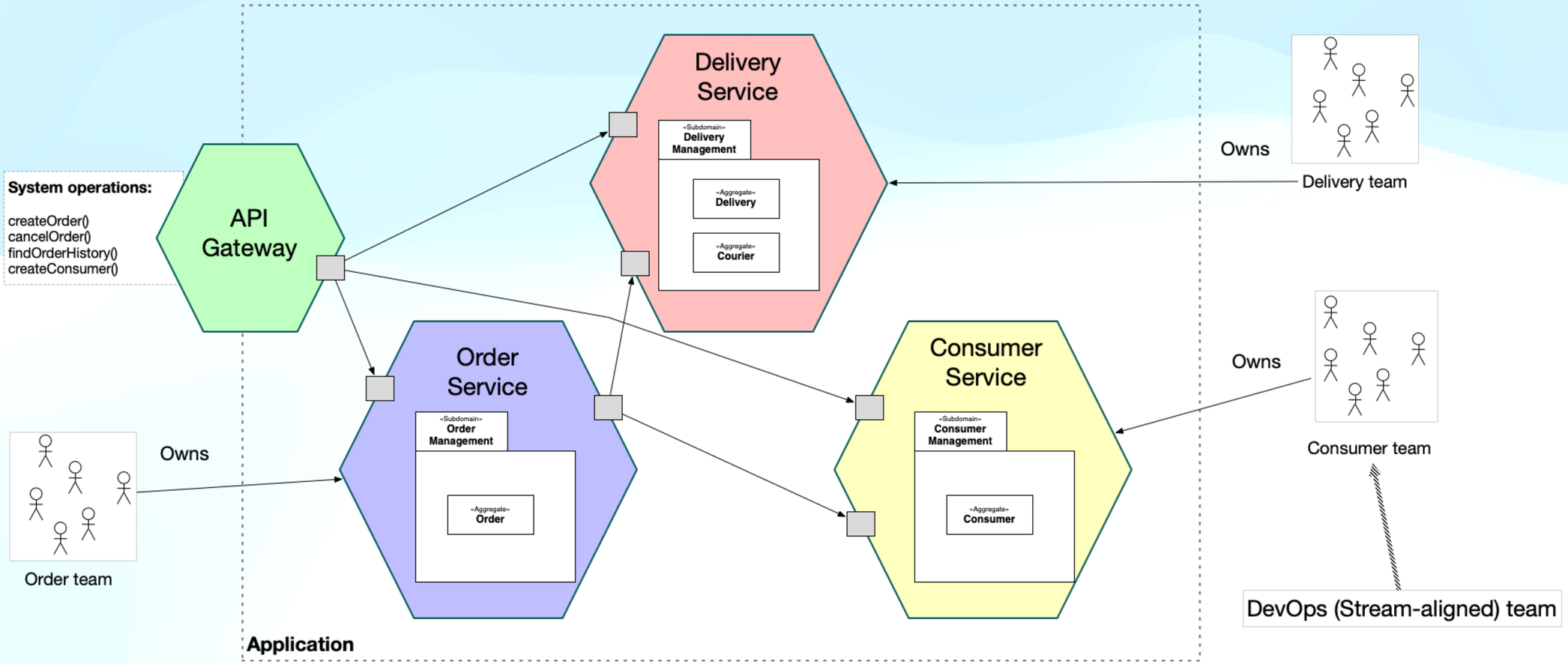
- You can’t make proper engineering tradeoffs without a baseline
- You may have heard of Kubernetes and microservices, they seem all the rage these days.
- Let’s start with a chart from the top (non-sponsored) Google search result for microservices: https://microservices.io/

The Microservice Architecture Pattern Language



Something a bit smaller

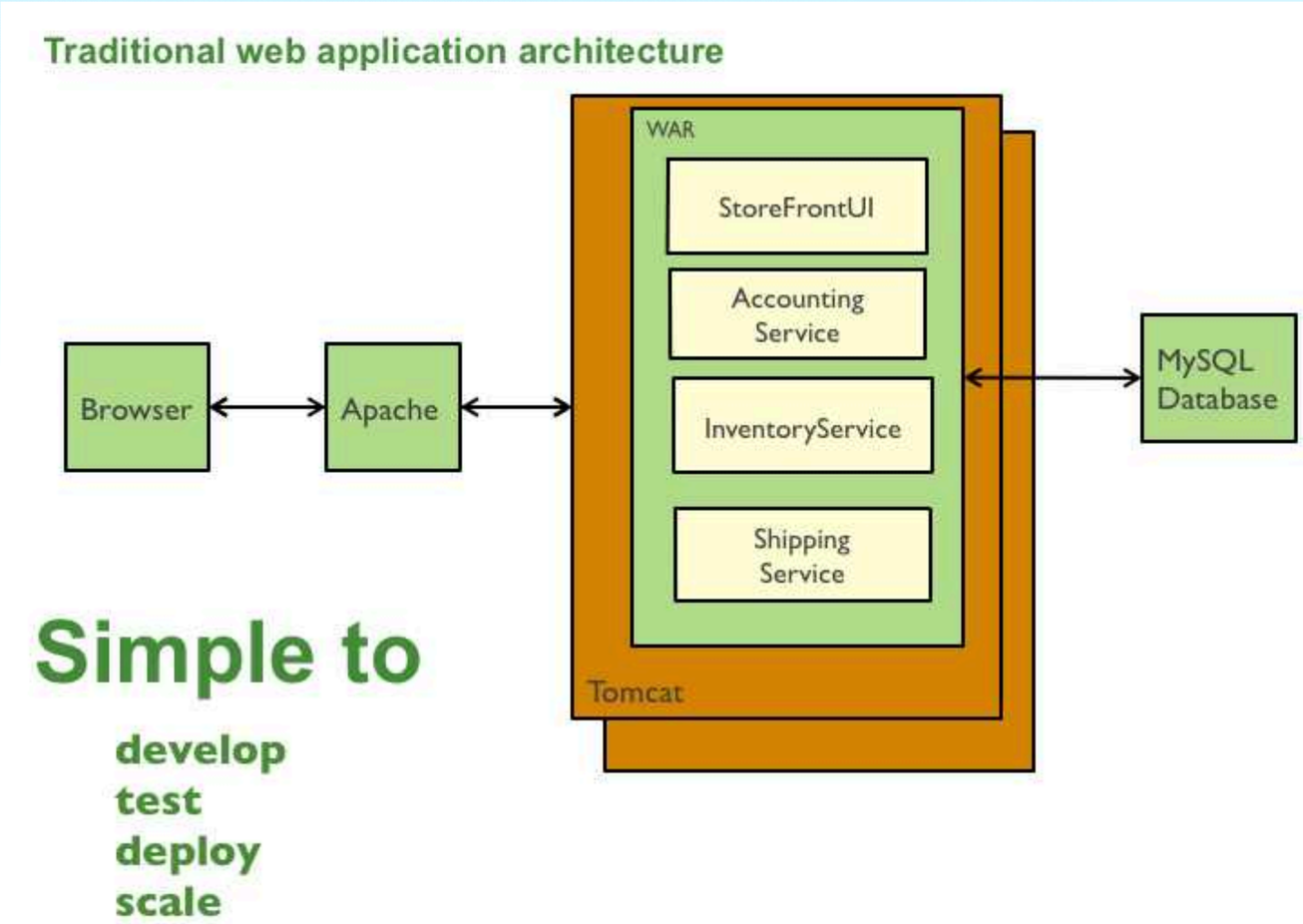
(from the same site)



Alternative

(Also from the same site)

- I kid you not, this is from the same site - complete with the description in the lower left



Observations

Comparison of the two approaches identified so far

- Microservices trades application complexity for orchestration complexity - in some cases that may be a valid trade-off
- Both approaches can be made to scale
- Neither approach directly address the key problems
  - Apps “want” to be near users
  - Apps “want” to be near data

Shared Nothing

Part 3

Shared Nothing

A journey from one to infinity

This part is going to have a number of sub-parts:

- Start with a single user / single machine
- Run multiple users with separate databases
- Run multiple machines ← **this is where we scale**
- Identify macro services
- Backups ← **this part is crucial**
- Logging
- Startup
- Admin UI

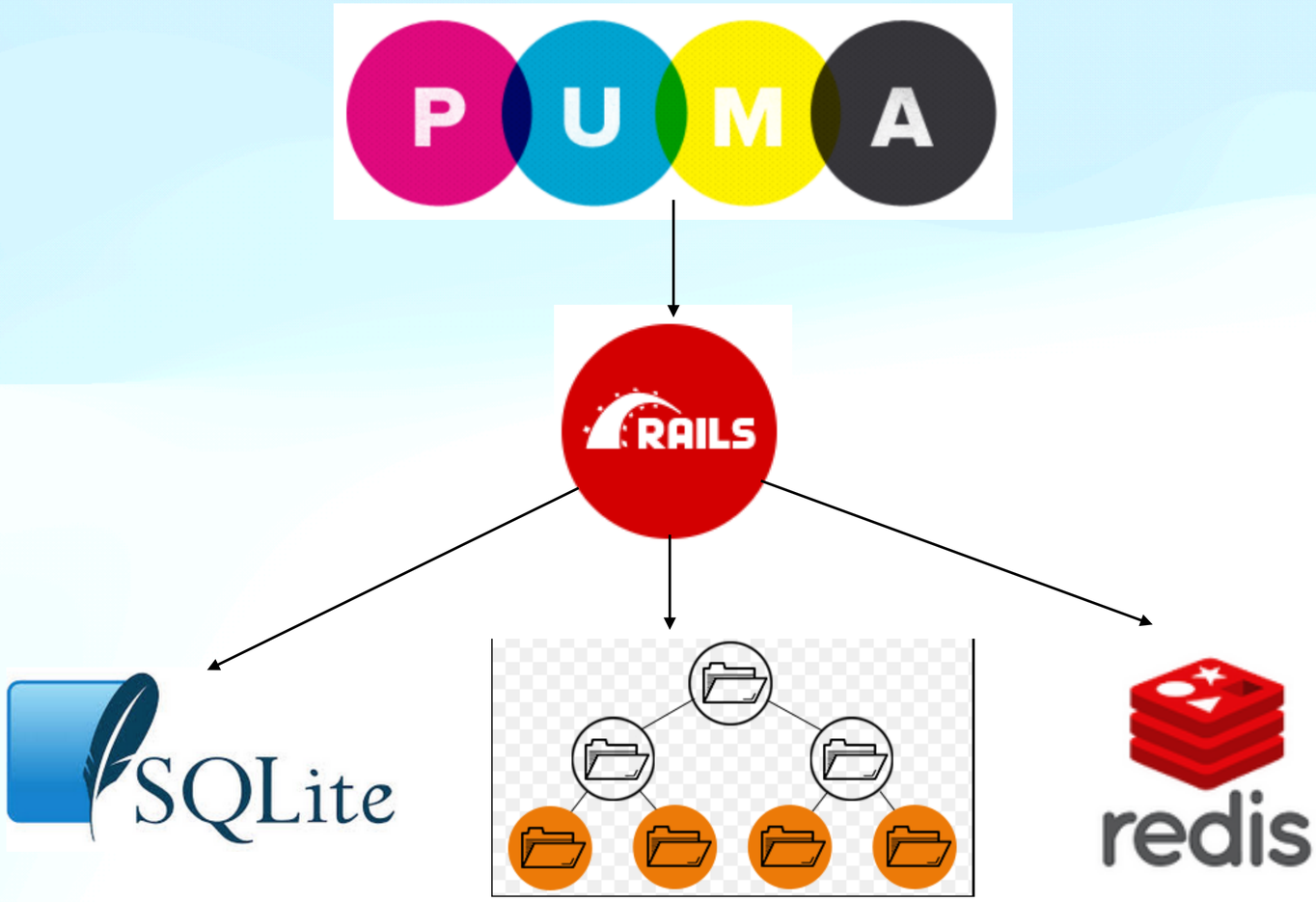
Step 1: Start with a single user / single VM

You are going to start this way anyway, so go with it…

- On your development machine you run all services on one machine, don’t you? (Perhaps some services in a container, but still on the same machine)
- Put all of the services needed to support one customer in one Docker image and deploy it

Step 1: Start with a single user / single VM

Standard Ruby on Rails



Step 1: Start with a single user / single VM

2022 - 3Q 2024

- My original deployment target: a M1 Mac mini in my attic
- Ample capacity
- Latency to Perth would be an issue
- Unaccessible at times due to power outages, storms

Step 1: Start with a single user / single VM

Advantages

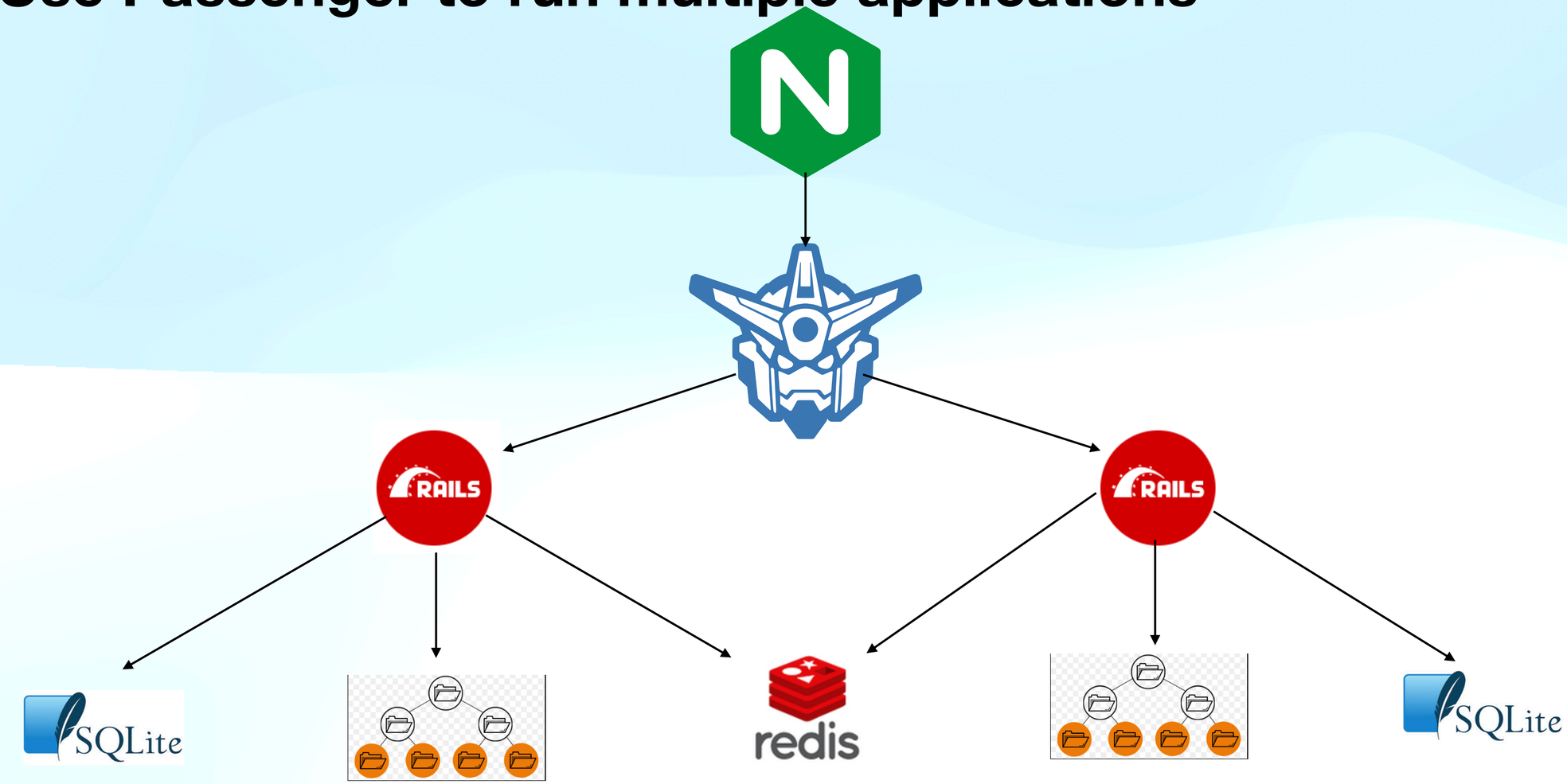
- No need to modify your application
- Eliminating the need for a network in producing a response to a request both increases throughput and reliability.
- Simultaneously deploy interdependent services as an atomic operation as opposed to independent updates of microservices running in production.
  - First rule of distributed computing: don’t.

Step 2: Run multiple users with separate databases

- This step is technically optional, but it helps when you break a larger task into discrete subtasks
- Making databases multi-tenant is hard, and requires application changes and/or database support, so lets not do that.
- All we need is an application server that can serve multiple applications, and Phusion Passenger can do that.

Step 2: Run multiple users with separate databases

Use Passenger to run multiple applications

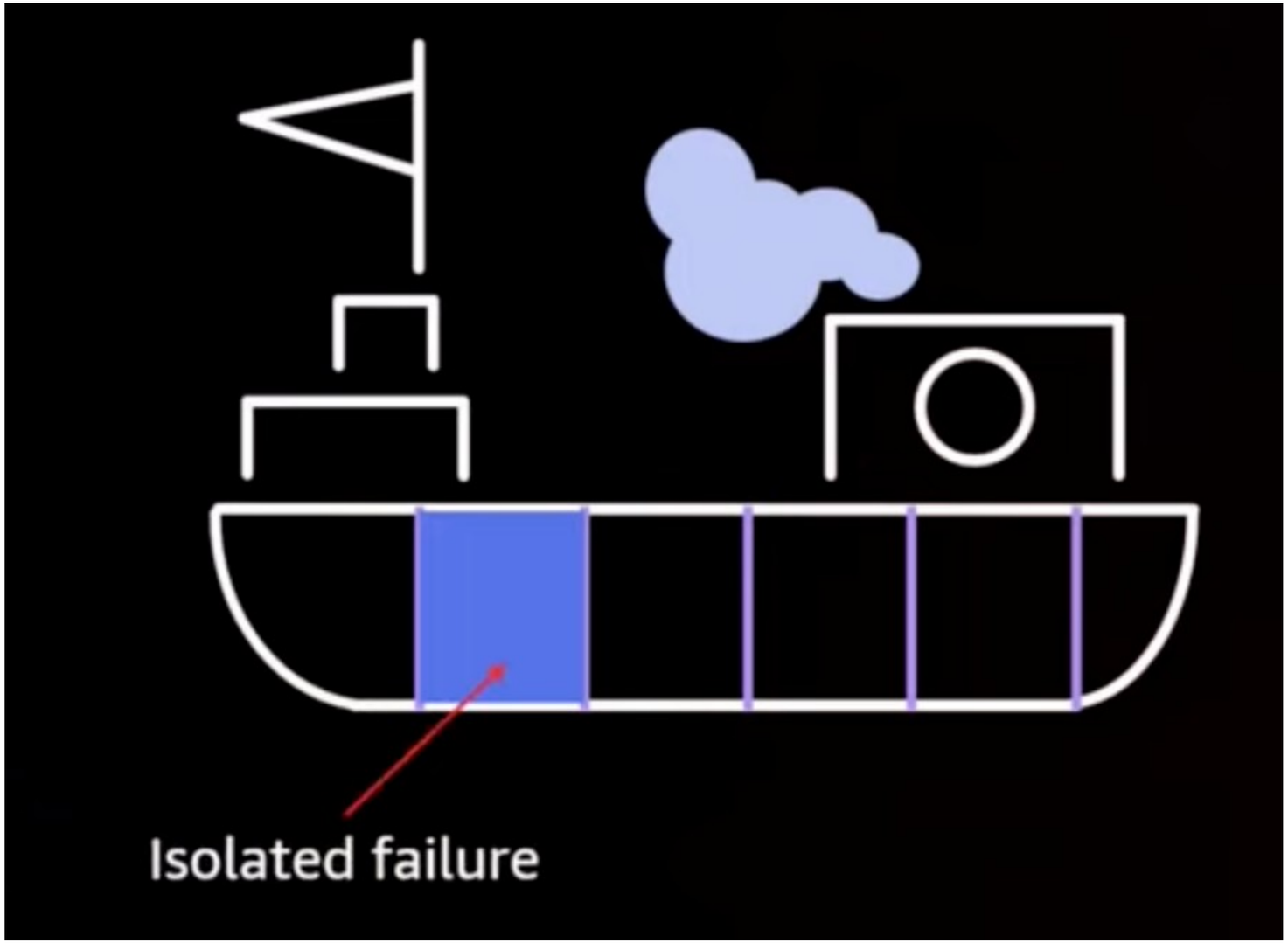


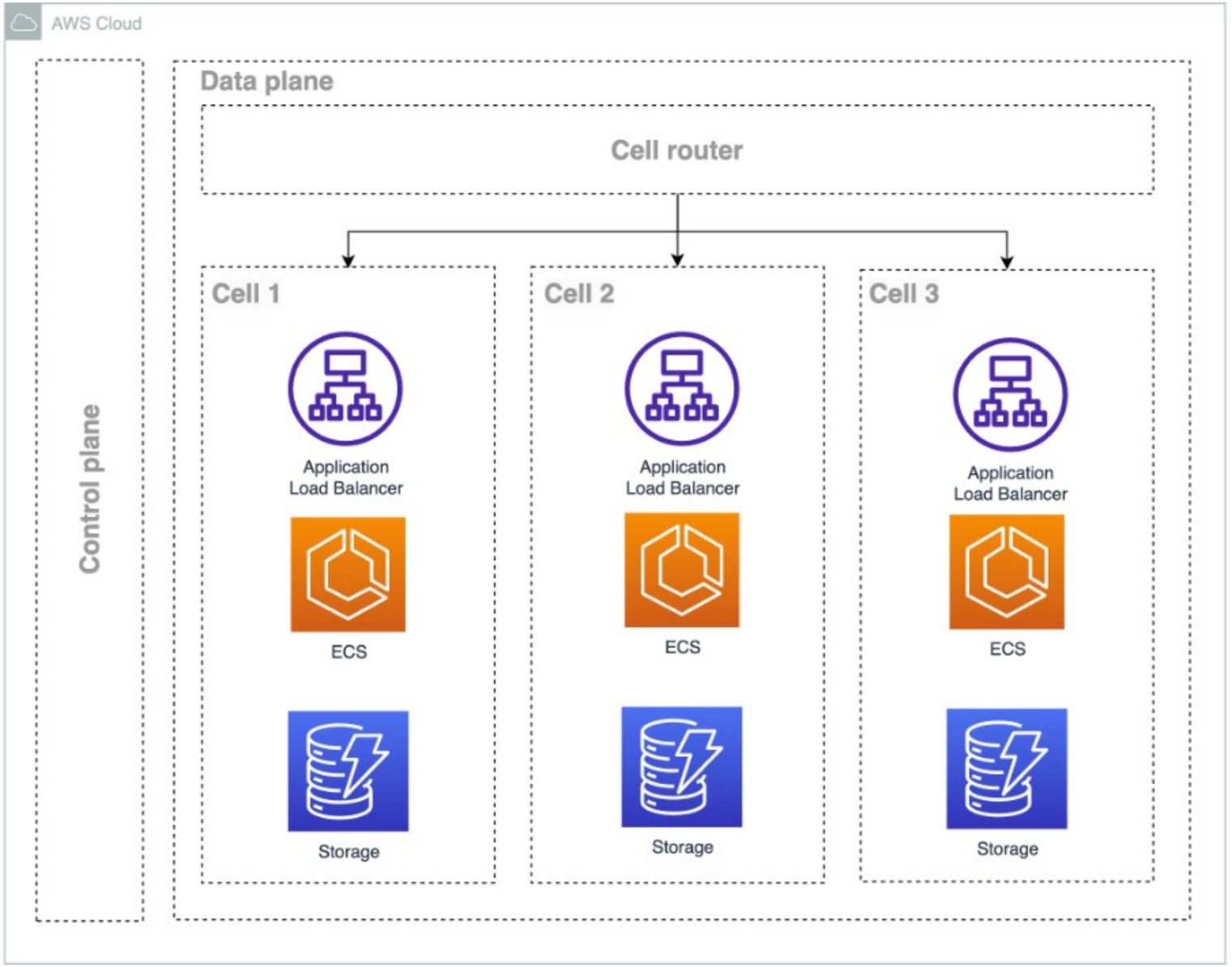
Step 2: Run multiple users with separate databases

Shared-Nothing Architecture

- Grey beards know this as a shared-nothing architecture
- Amazon rediscovered this pattern and called it cell-based architecture
  - I like this name too!

- Astute members of the audience will note that redis is shared, but only technically; prefixes are used to ensure that no data overlaps


Isolated failure

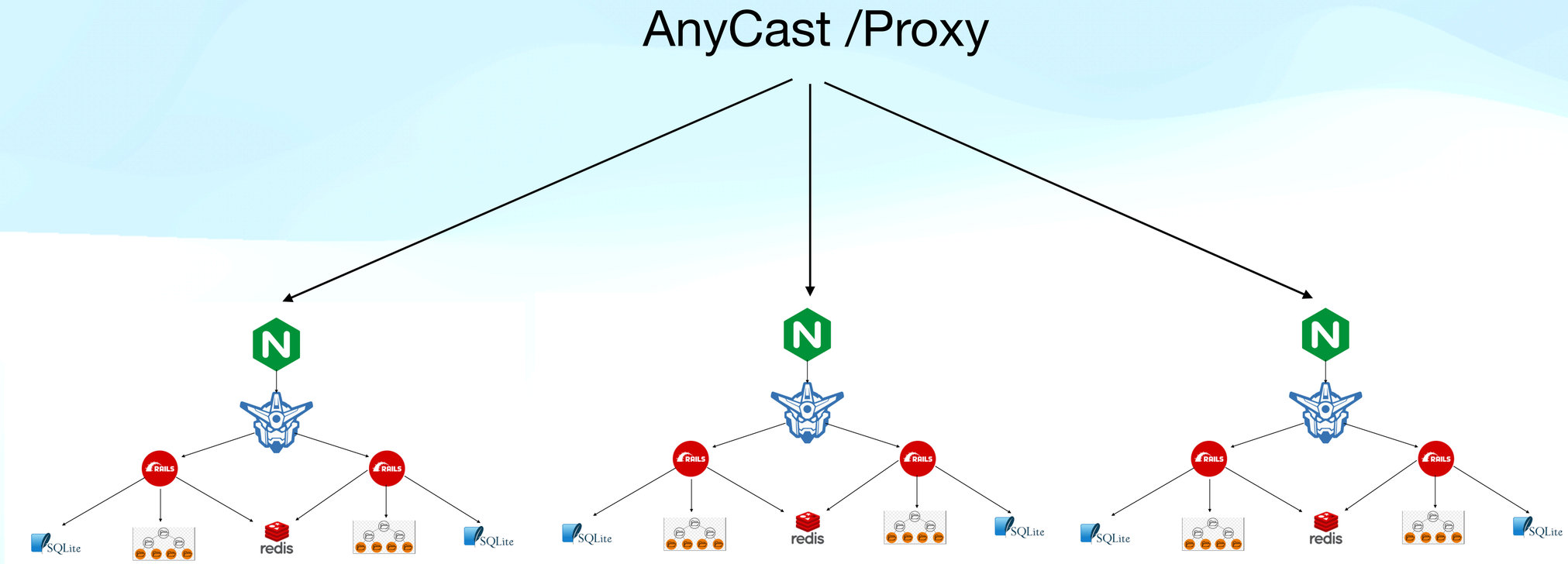

AWS Cloud
Control plane
Data plane
Cell router
Cell 1
Application
Load Balancer
ECS
Storage
Cell 2
Application
Load Balancer
ECS
Storage
Cell 3
Application
Load Balancer
ECS
Storage

Step 3: Multiple machines

Scale!

- AnyCast is a methodology where a single IP address is shared between multiple servers
- Load balancing proxies (such as Fly Proxy) will route requests to the nearest available server.
- Dynamic Request Routing features (these are platform specific) can be used to ensure that requests are routed to the right place.

Step 3: Run multiple machines



Anycast

The preceding picture is misleading; it looks like AnyCast is a single machine, and therefore a bottleneck and single point of failure. What actually happens:

- AnyCast is part of TCP/IP; and selects the nearest edge server
- Copy of the proxy running on the edge selects the machine:
  - Based on HTTP headers, if provided
  - Nearest available if not
- Target machine can ask that the request be rerouted (replayed)

Anycast

fly.io regions

Events per region

Why not one event per machine?

atl Region

2024

- Charlotte - June 22, 2024
- Kennesaw -
- Nashville-Green Hills - April 27, 2024
- Richmond
  - Medal Ball - April 27, 2024
  - Team Match -
  - Nouveau Duo -

2023

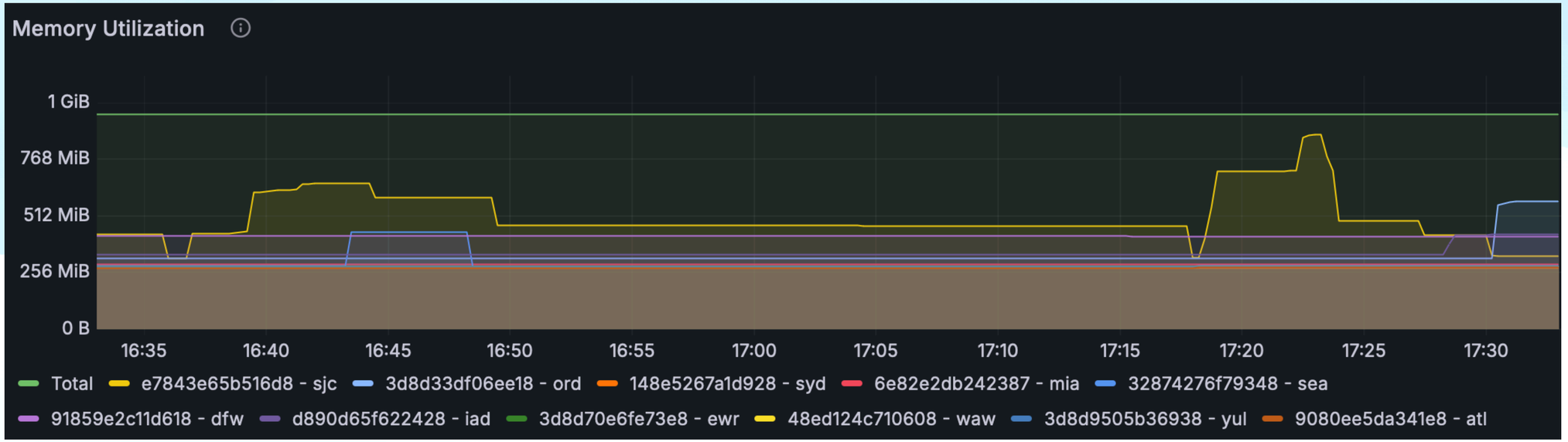
- Charlotte - June 23 & 24, 2023
- Kennesaw - Saturday November 11th, 2023
- Nashville-Green Hills - 11-11-23
- Richmond
  - Medal Ball - April 29th, 2023
  - Team Match - 8/12/2023
  - Nouveau Duo - December 16th, 2023

2022

- Charlotte - July 29 & 30, 2022
- Kennesaw - August 26, 2023

Capacity planning - memory

Typical load



Note: this app reserves 2Gb of swap space on each machine.

Capacity Planning - CPU

Typical load

Capacity Planning

Typical Load

This application certainly isn't CPU constrained:

- Most requests take less than 100ms to process; many under 40ms.
- An average person generates a single digit number of requests per minute.
- Peak load is, perhaps, four users?

Response times

Do we scale?

Response times

- Requests from users local to an event find a machine quickly
  - Typically a request never leaves a geographic region
  - Remote requests are still possible
- Once a machine receives a request, everything needed to produce a response is available on that machine
- Both vertical (more events per machine and bigger machines) and horizontal (more machines) scaling are enabled by this architecture.

Scalability

Mission accomplished?

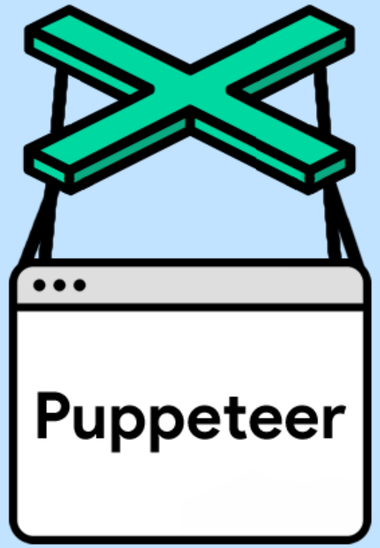
- Revisiting the key goals:
  - Apps “want” to be near users - ✅
  - Apps “want” to be near data - ✅

Step 4: Macro services

Revisiting microservices

I don’t see the value is splitting invoicing from scheduling from data entry, but…

- Users want printed reports
- Printing from the browser is a lousy end-user-interface
- Generating PDFs using Puppeteer is much better



Macroservices

Issues to be resolved

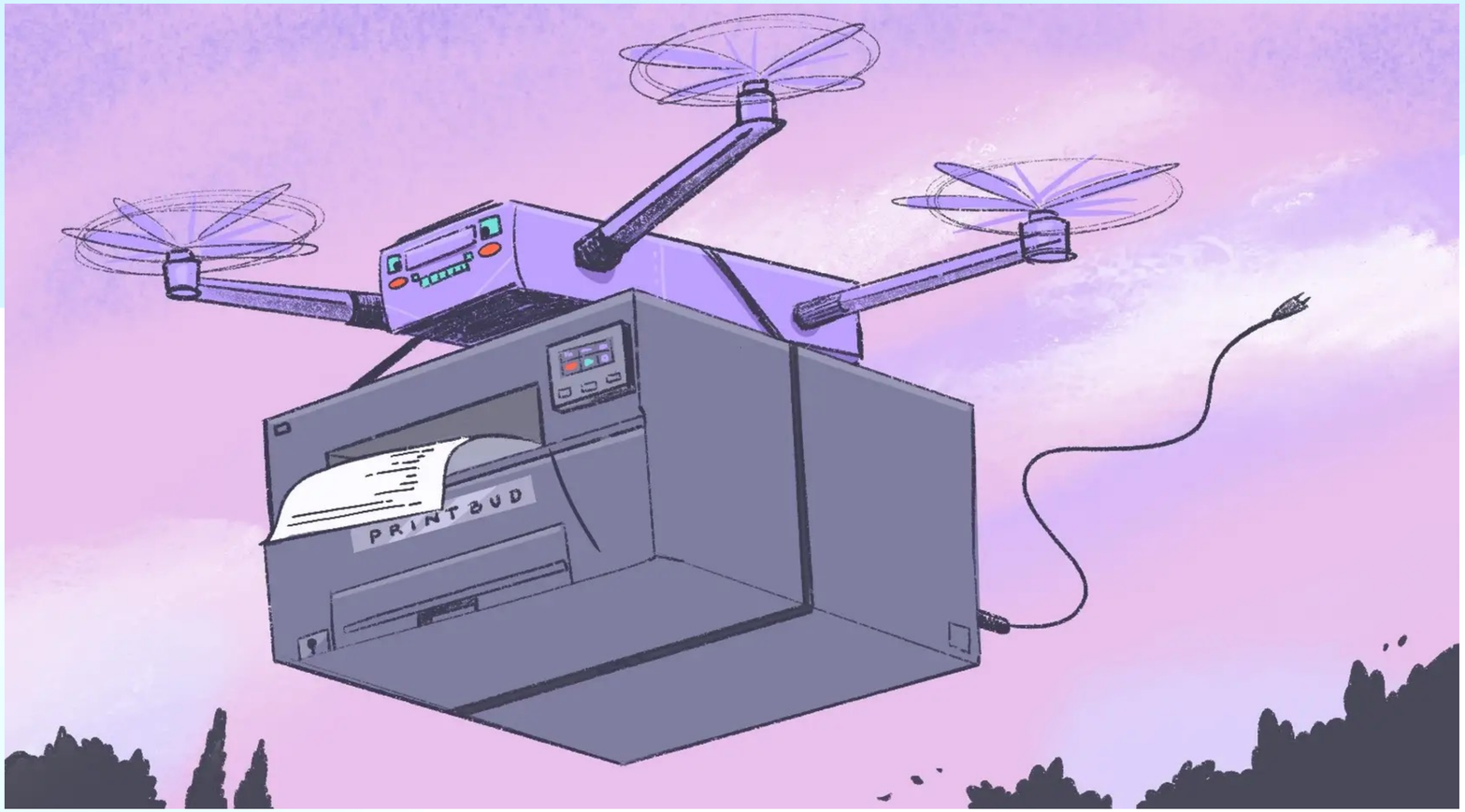
Reality:

- Chrome is bigger than my app
- Chrome is a memory hog

Result: running both events and puppeteer on the same machine results in crashes.

Macroservices

The solution: print on demand



Print on Demand

Machine configuration

- Four geographically dispersed machines are defined
- 2Gb of RAM + 2Gb of swap each
- Each machine is shuts down when not in use; restarts on demand

Step 5: Backups

Engineering tradeoffs

- Backing up 11 regions is a bigger problem than backing up 1 region.
- Data from events don’t overlap

Backup solution

Current implementation

- Whenever an event goes idle, its database and media files are copied to all other regions using rsync.
- All data is also rsync'ed to my home server as a fallback of last resort. This also makes it easy to debug problems with a copy of live data.
- My home server takes daily snapshots of everything (using hard links when data is unchanged to minimize storage).

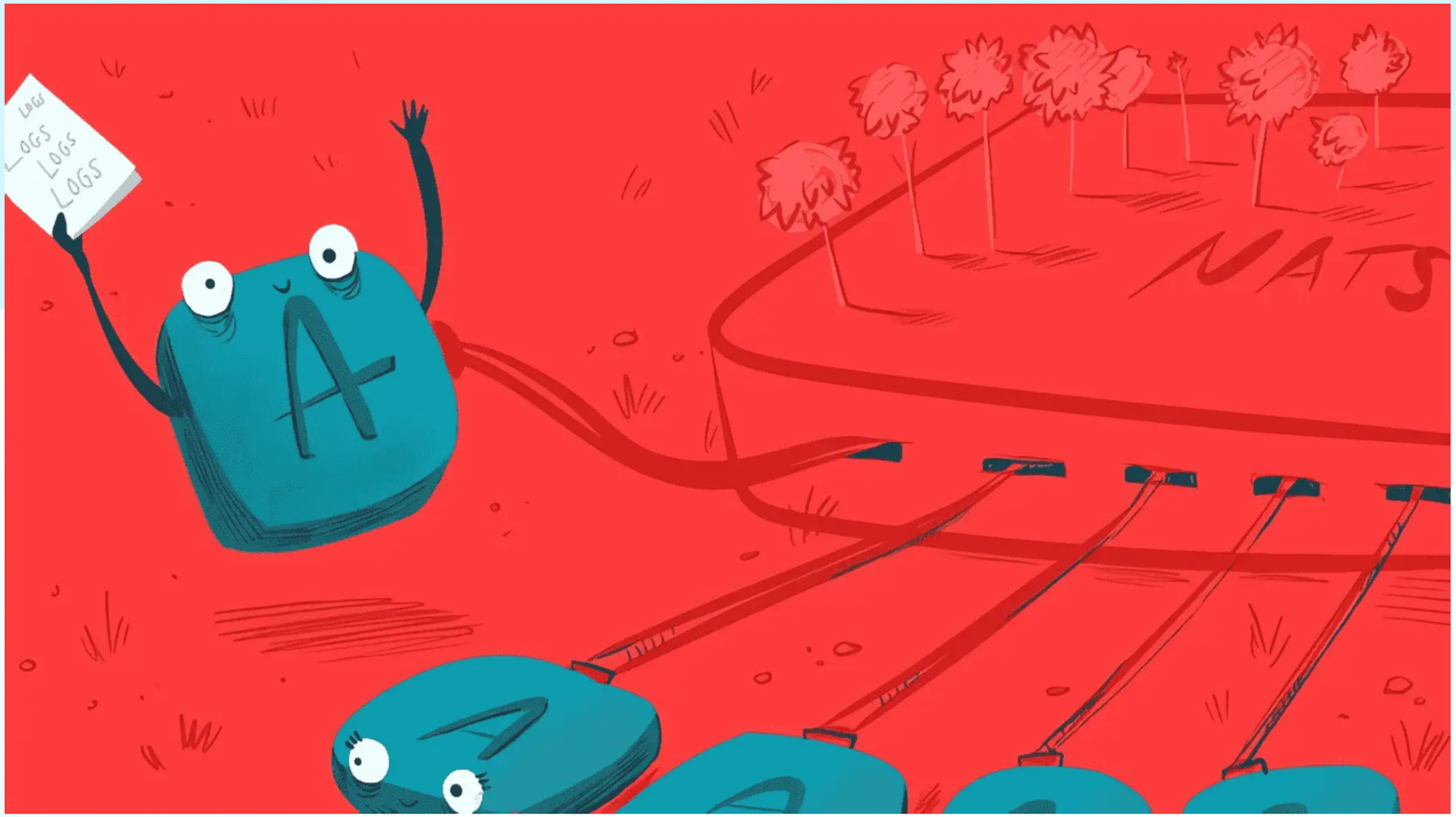
Step 6: Logging

More engineering tradeoffs

- 11 region problem applies here too

Logging

Solution: log shipper!



Shipping Logs

More information

- Log shipper
- Multiple Logs for Resiliency

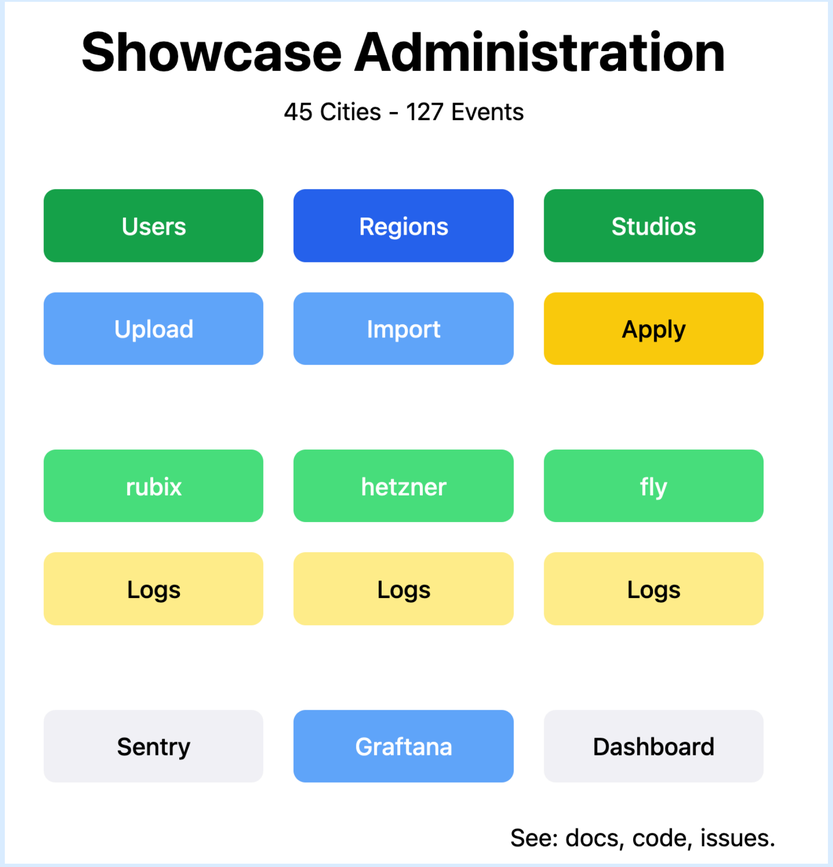
Step 7: Deploying

Rubber meets the road

- Deploying a update is a matter of replacing a machine and starting the application.
  - No ordering problems like you see with microservices
  - Machines start in milliseconds. A (small number) of hundreds of milliseconds to be sure, but still fast. And proxies will buffer.
  - If DB migration or rsync is needed, app startup may be delayed, but displaying a “please wait” helps.

Step 8: Admin UI

Automating administration



Shared Nothing

A journey from one to infinity

This part is going to have a number of sub-parts:

- Start with a single user / single machine
- Run multiple users with separate databases
- Run multiple machines ← **this is where we scale**
- Identify macro services
- Backups ← **this part is crucial**
- Logging
- Startup
- Admin UI

Just imagine…

What if you could give **every** user of your software the *experience* of having a **dedicated server machine** assigned to them?

Shared Nothing

Links

- Agile Web Development with Rails 7.2
- Fly.io
- smooth.fly.dev
- https://github.com/rubys/showcase/blob/main/ARCHITECTURE.md