CS 4530 & CS 5500 Software Engineering

Lecture 10.3: Deployment Infrastructure

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

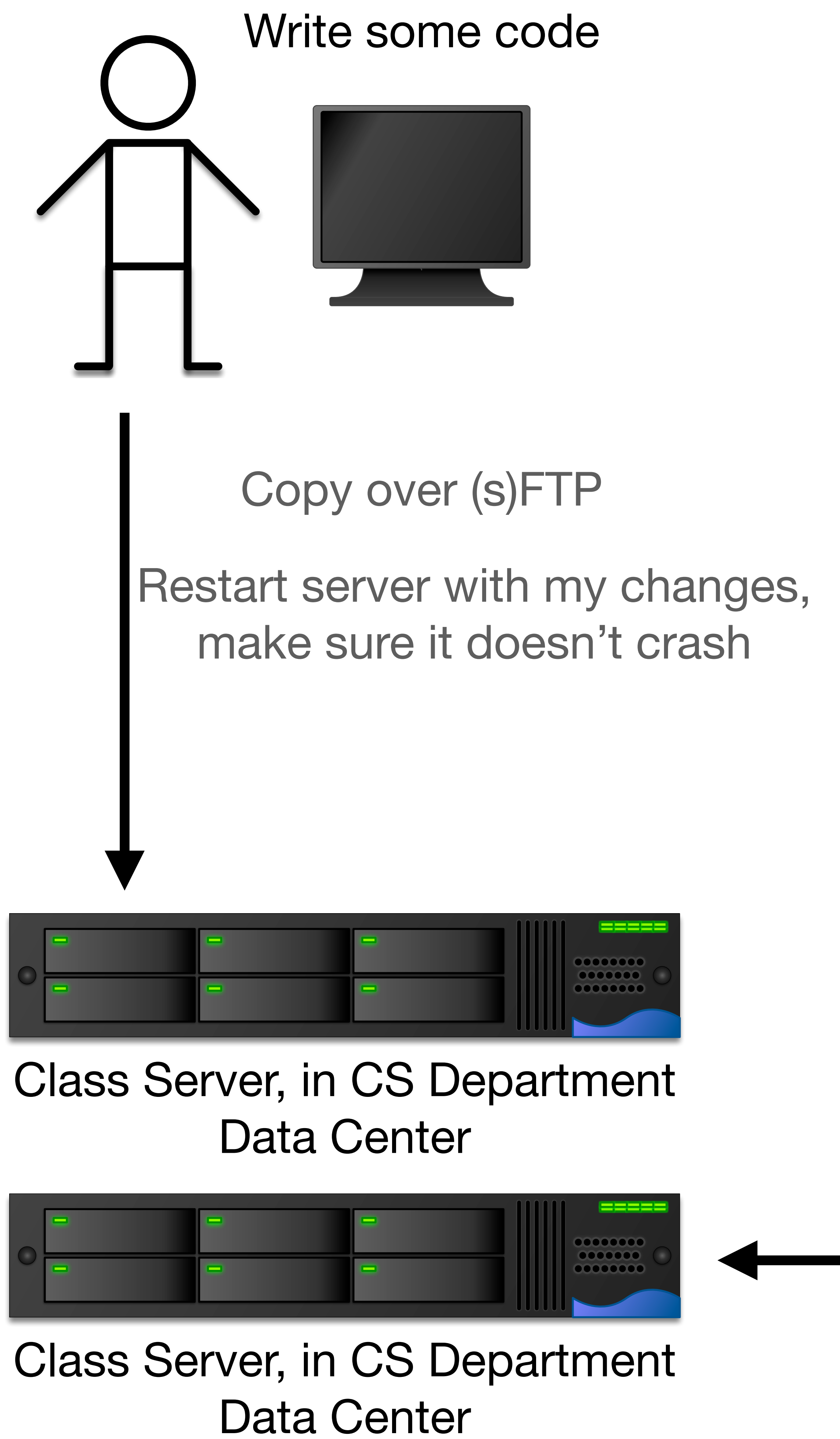
Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

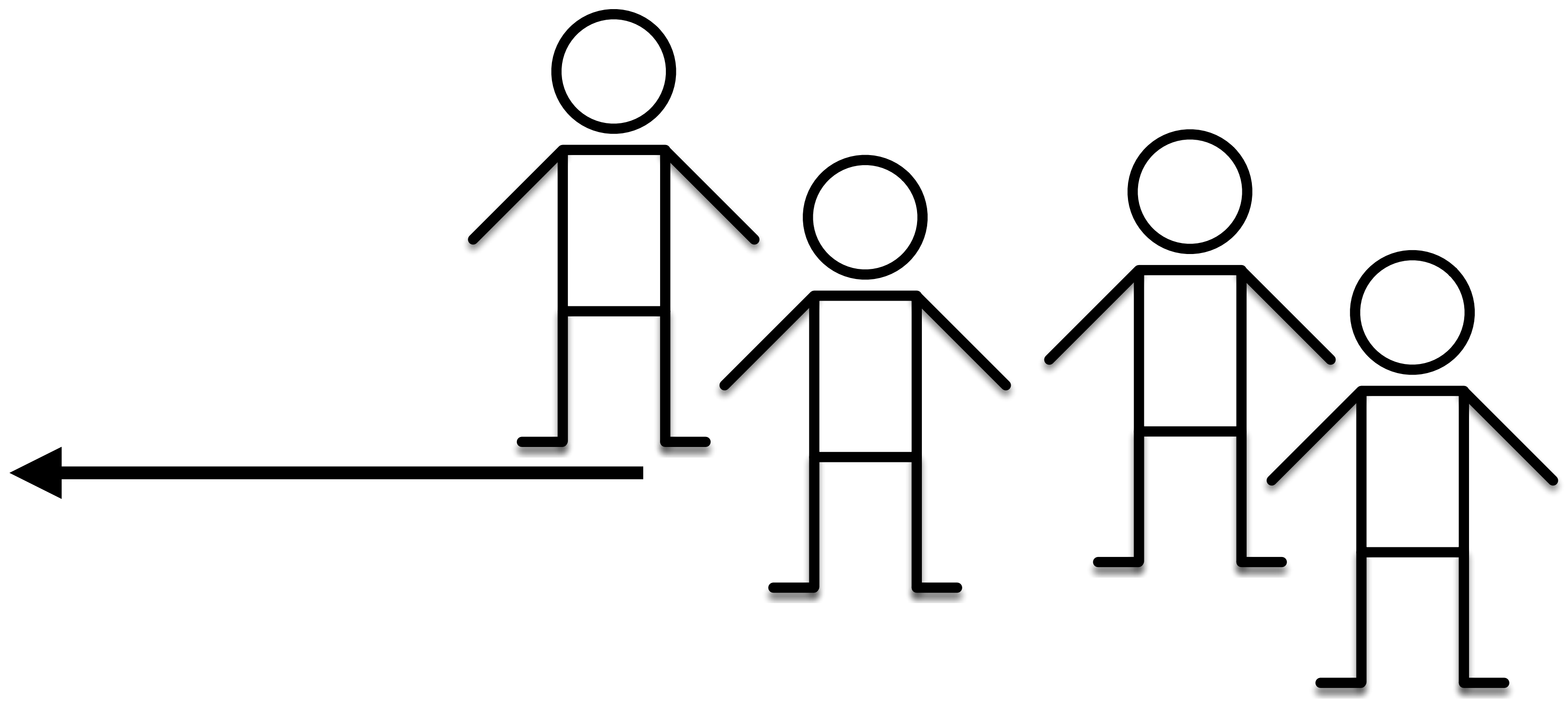
- Describe the difference between key deployment container abstractions and their role in modern software
- Compare the performance and cost of different deployment infrastructures, including platform-as-a-service

Deploying a Web App

Circa 2008: Manual deployments to private or shared machines



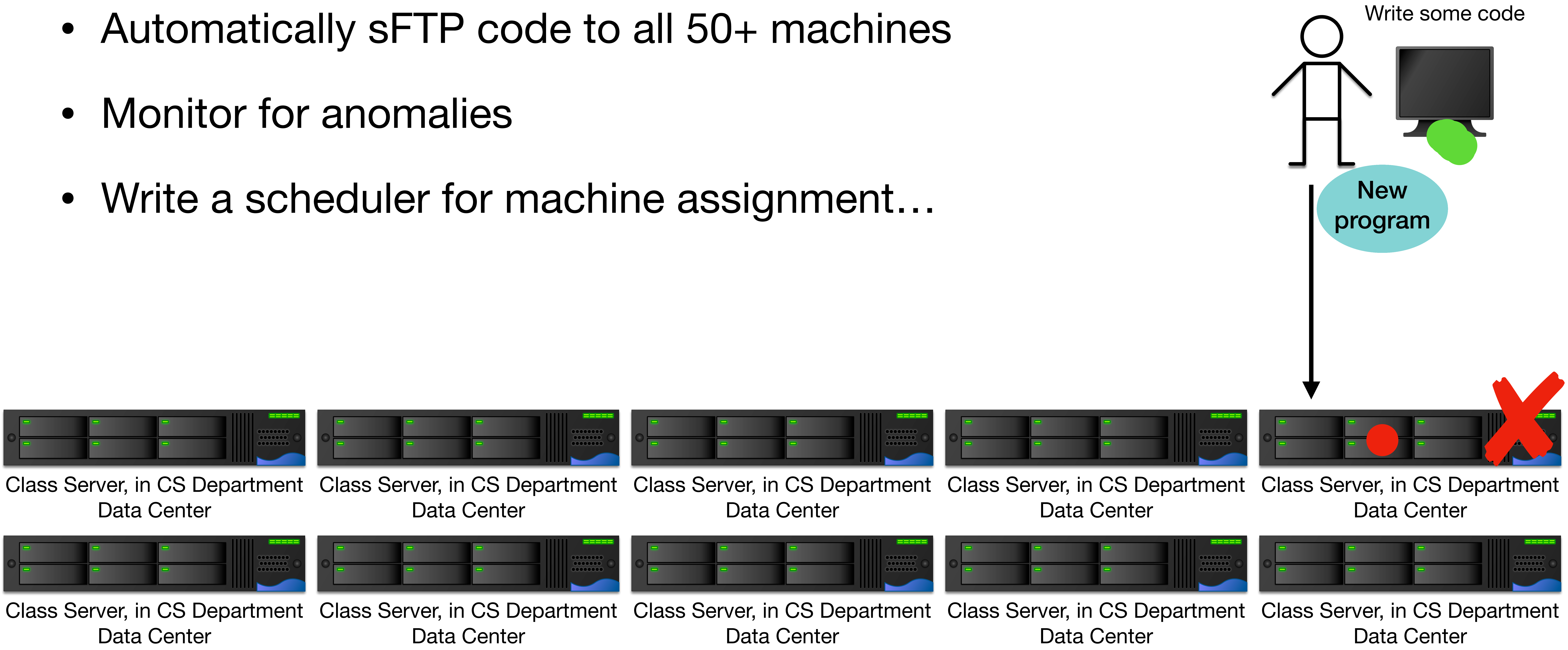
- A simple approach that works, but does not scale in:
  - Number of machines
  - Number of programs
  - Size of programs
  - Frequency of deployments



Deploying a Web App

Making it better: automation

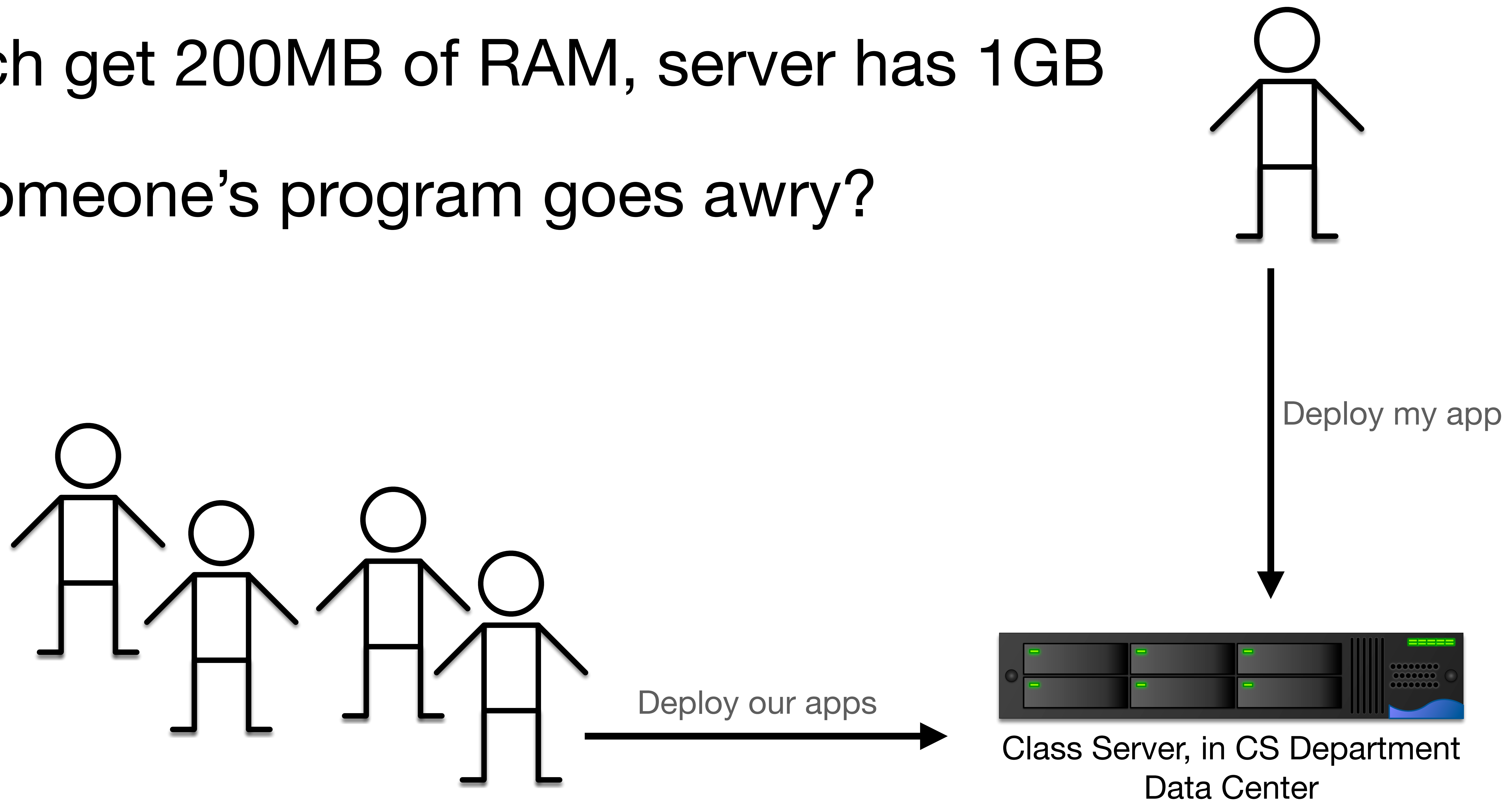
- Automatically sFTP code to all 50+ machines
- Monitor for anomalies
- Write a scheduler for machine assignment…



Deploying a Web App

Making it better: Multitenancy

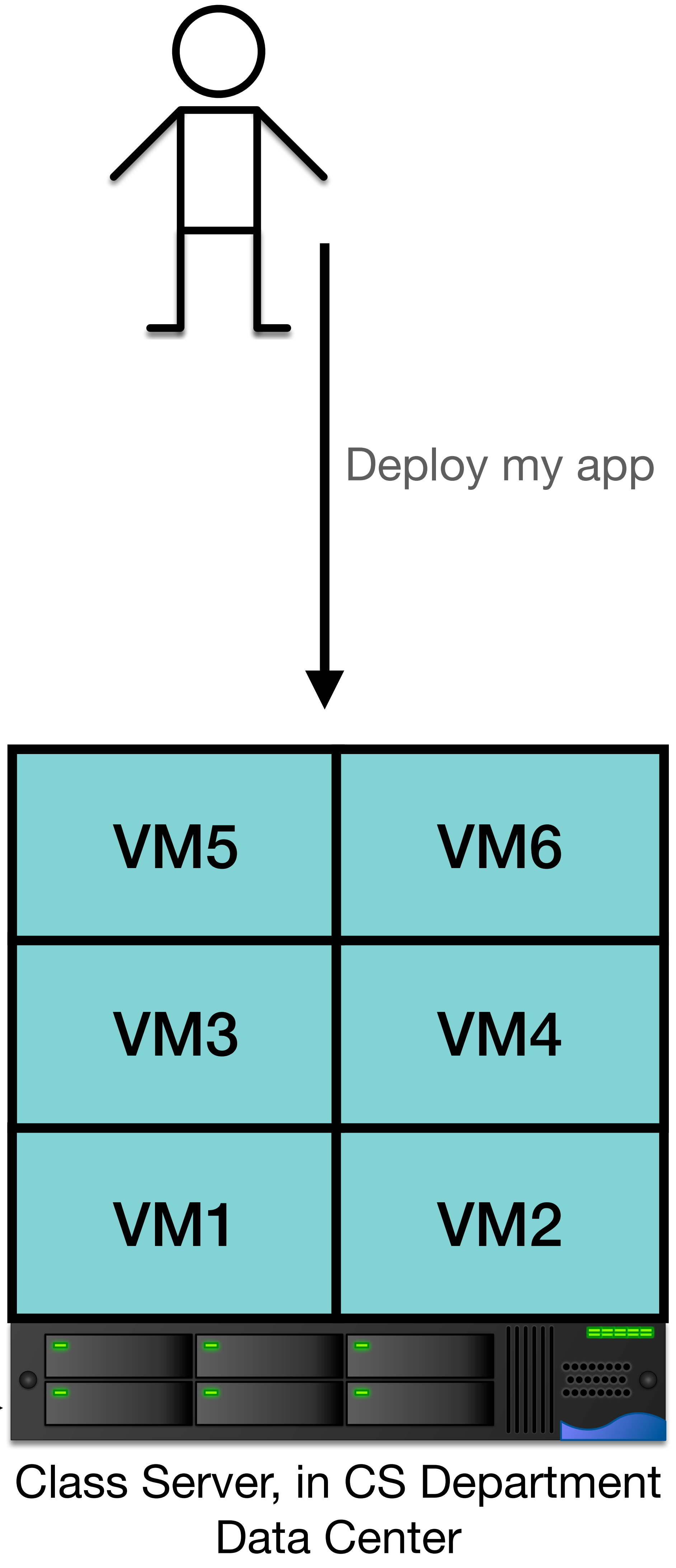
- What if the mapping of programs/users to machines is not 1:1?
- Example: 5 applications, each get 200MB of RAM, server has 1GB
- Problem: What happens if someone’s program goes awry?

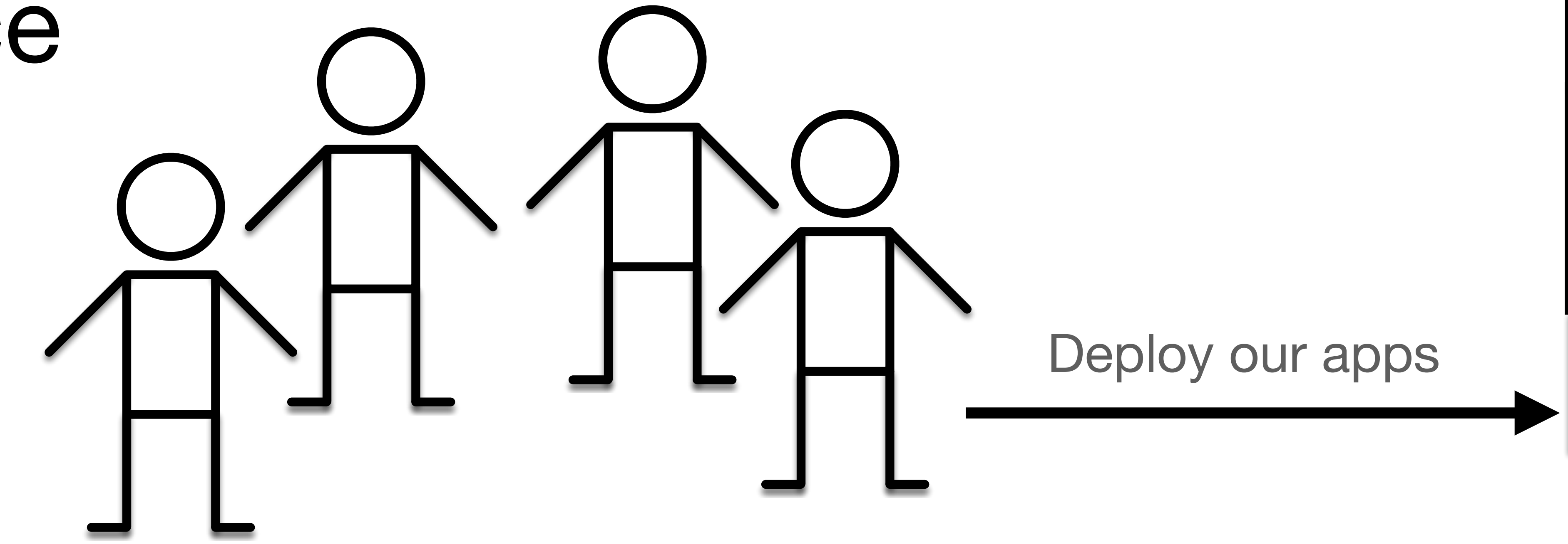


Multi-Tenancy

Virtualization to the rescue

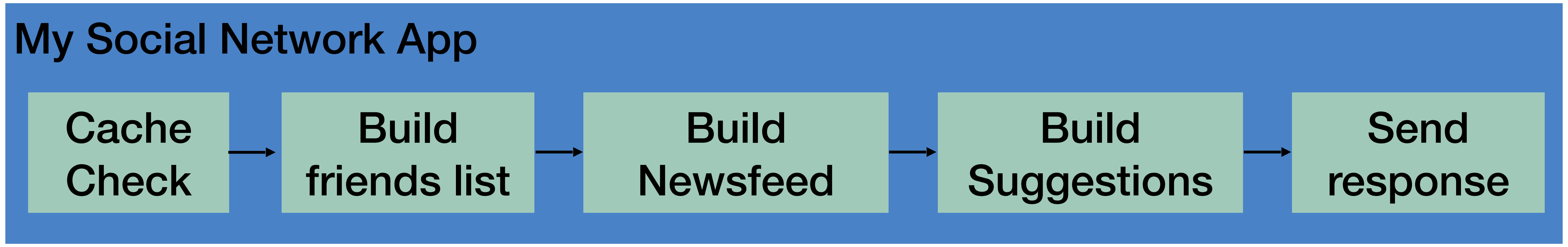
- Solution: Each app gets its own Virtual Machine (VM)
- OS provides resource limits and quality guarantees per-VM
- Each VM runs its own OS - not an efficient use of resources (5x200MB RAM for each app PLUS 5x500MB RAM to run 5 OS's)
- Lightweight containers (e.g. Docker) provide isolation, but run in same OS, less resource utilization



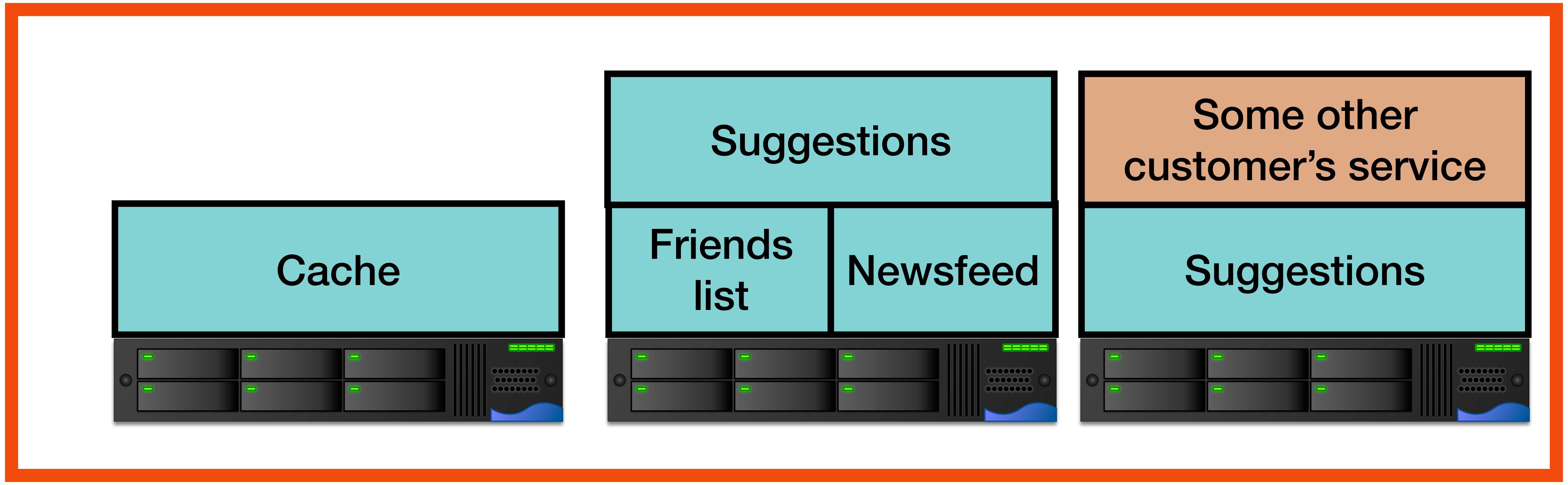


Automating Deployment of Complex Infrastructure

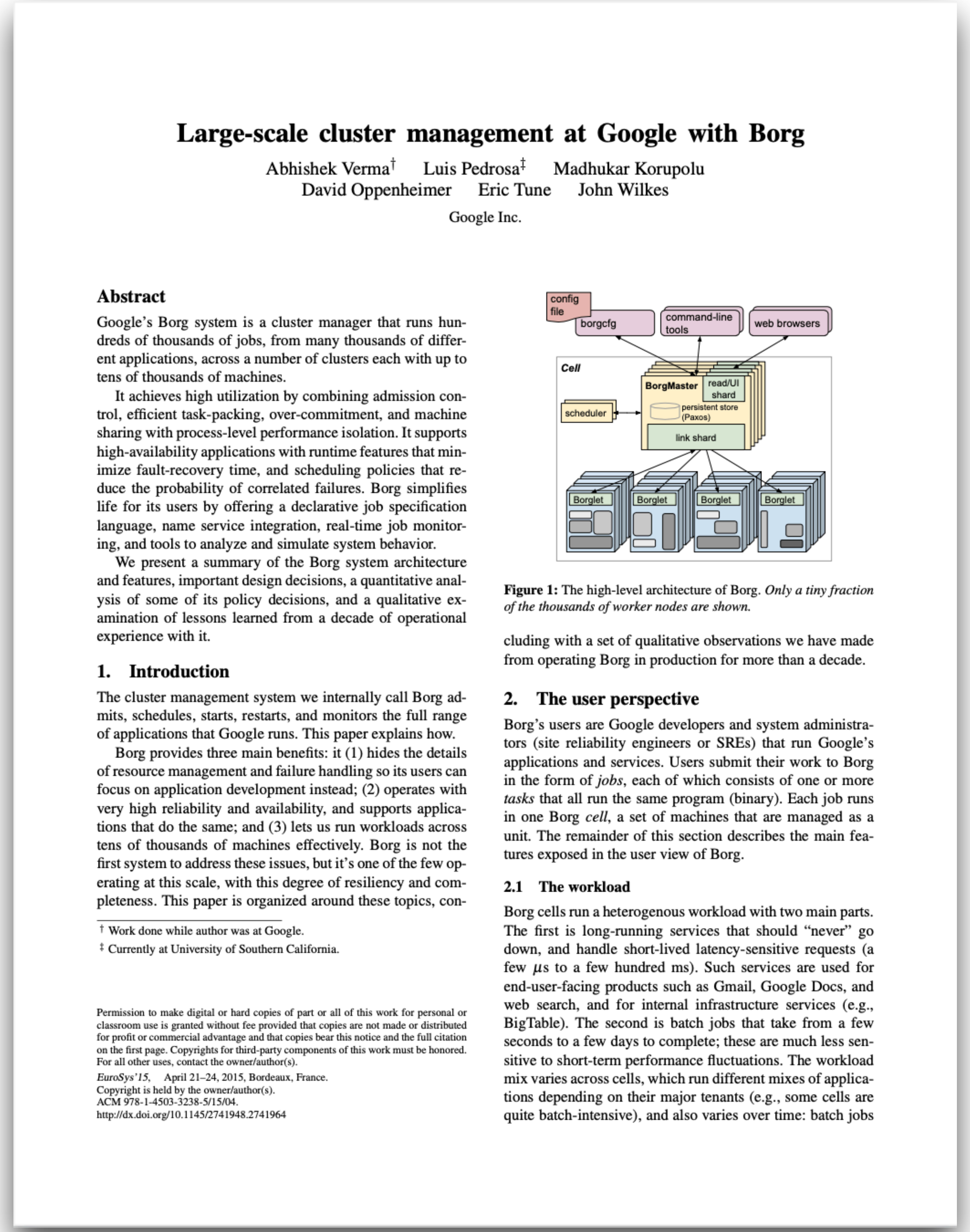
Automation + Multi-tenancy: Kubernetes



“Give me at least 1 of each of these app services in their own docker containers, and if the load gets above a threshold, spin up more of them”



Managed by Kubernetes



Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes
Google Inc.

Abstract

Google’s Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it’s one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, concluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

2. The user perspective

Borg’s users are Google developers and system administrators (site reliability engineers or SREs) that run Google’s applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

2.1 The workload

Borg cells run a heterogenous workload with two main parts. The first is long-running services that should “never” go down, and handle short-lived latency-sensitive requests (a few μs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

[†] Work done while author was at Google.
[‡] Currently at University of Southern California.

EuroSys’15, April 21–24, 2015, Bordeaux, France.

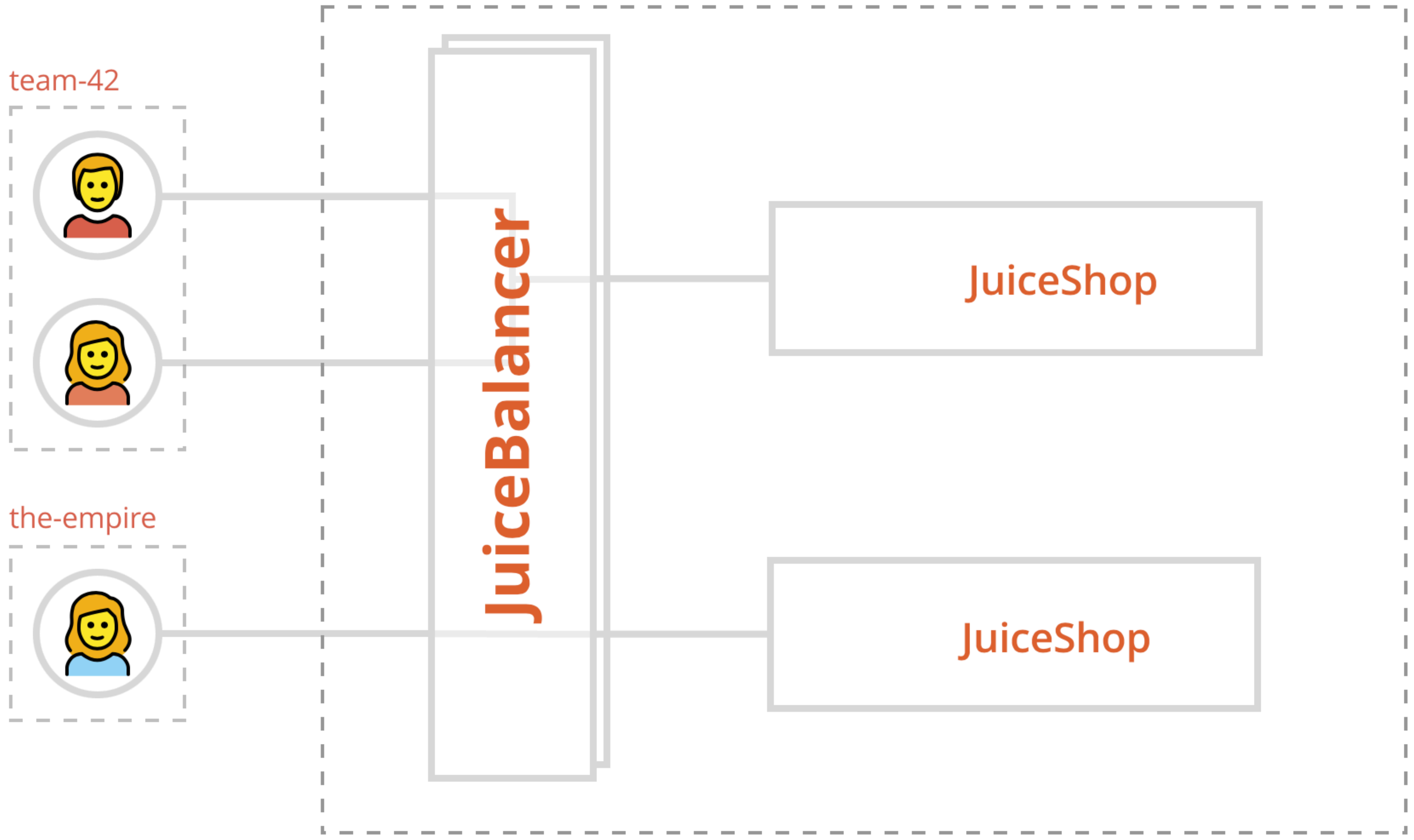
ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741964

https://research.google/pubs/pub43438/

Automation + Multi-tenancy: Kubernetes

Example: Multi-Juicer, the Juice Shop Framework

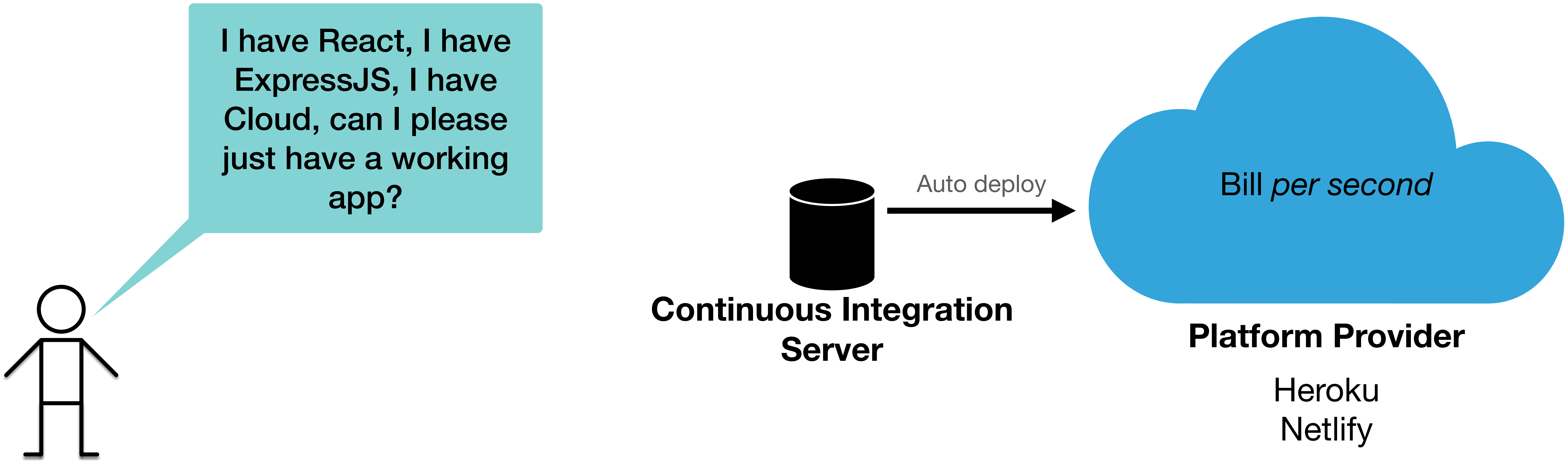
- Each team gets a JuiceShop instance
- Each JuiceShop instance is a docker container
- Multiple docker containers run on the same VM
- A load balancer provisions new containers
- As VMs get full, new VMs are booted



https://github.com/iteratec/multi-juicer

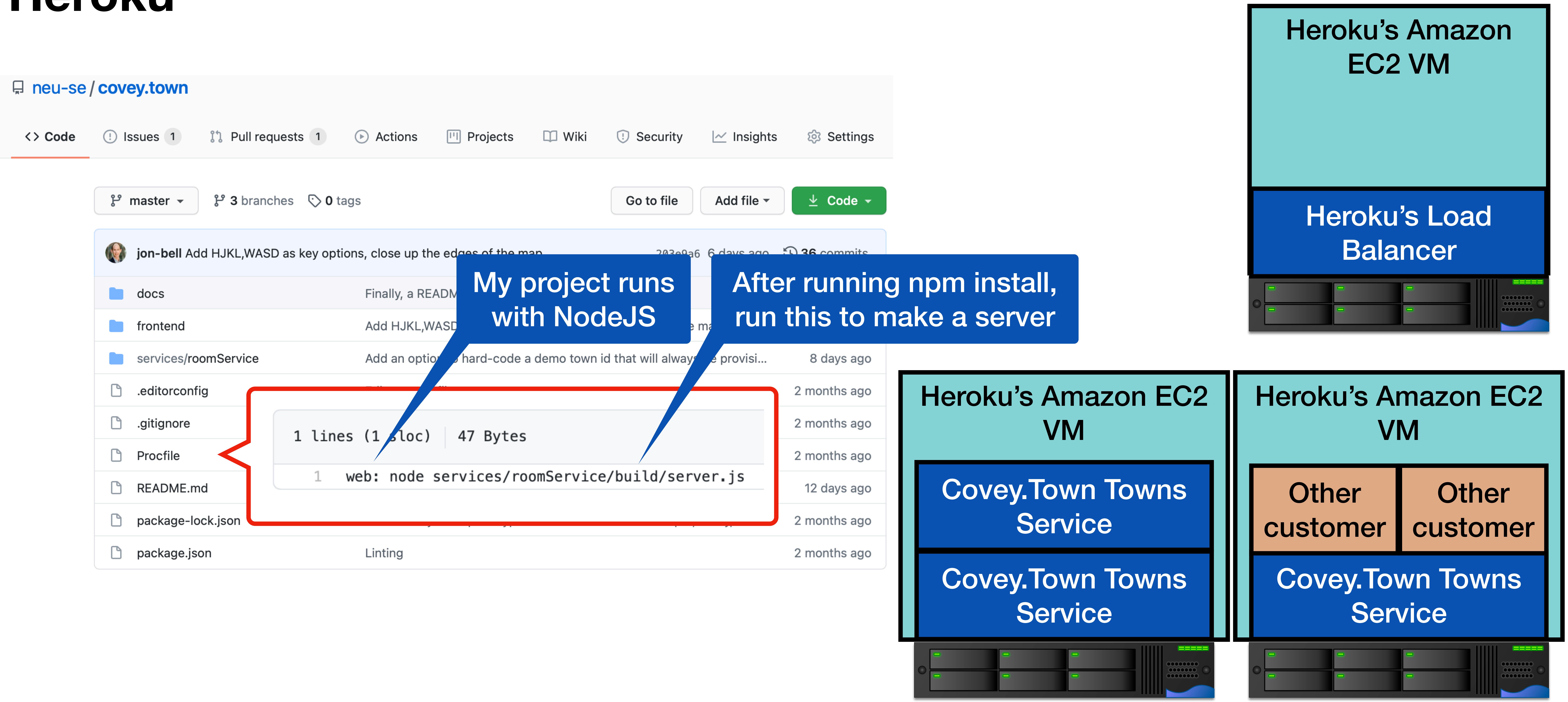
Multi-Tenancy

Platform-as-a-service: What if we *don't care* about the infrastructure?



Platform-as-a-Service: Covey.Town Deployment

Heroku

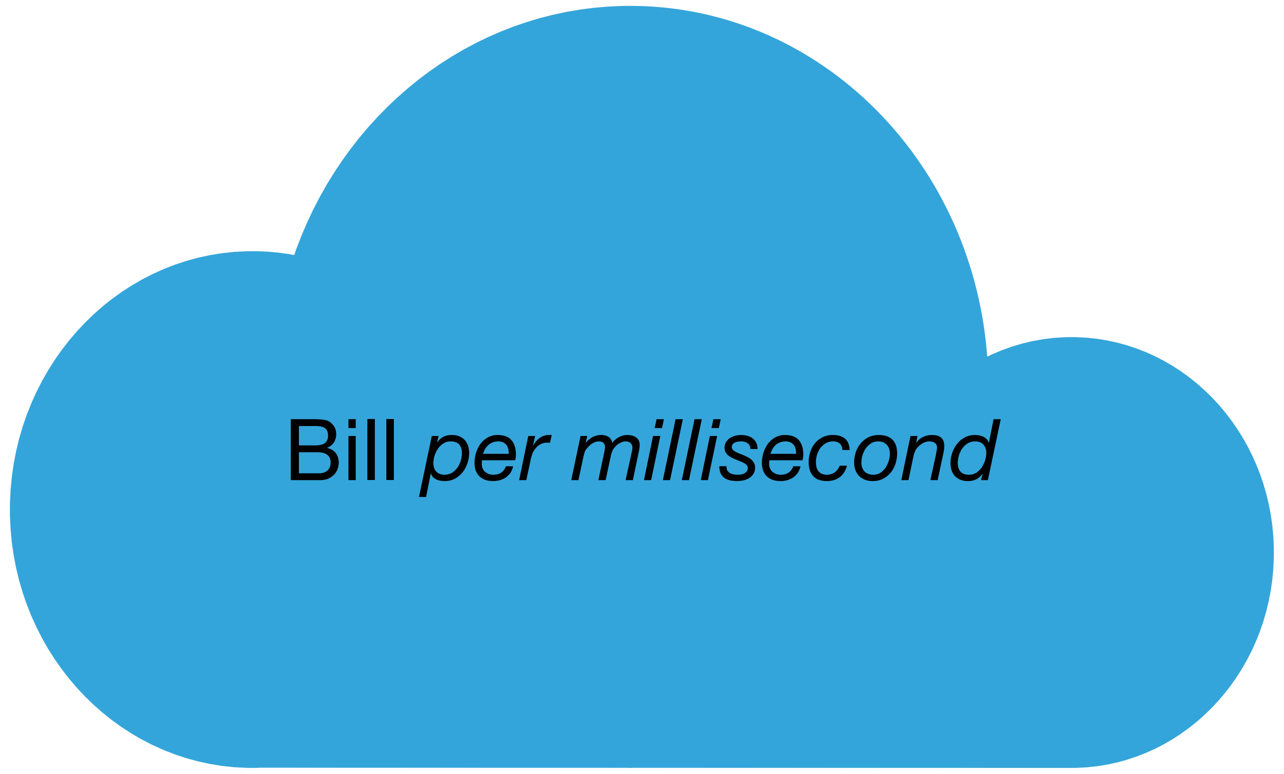


Platform-as-a-Service: Covey.Town Deployment

Netlify

Multi-Tenancy

Functions-as-a-Service: What if we just have a few functions that get called irregularly?



Serverless Provider

AWS Lambda
Google Cloud Functions
Azure Functions
Cloudflare Workers
Apache OpenWhisk

Computing Infrastructure

Choosing an abstraction for your application

- Centralization vs customization: “machines as cattle vs machines as pets”
- How do we manage state?
- What is our expected scale?
- How much management overhead do we want to take on?

Computing Infrastructure

Summary of the options

- Deploy VMs: Greatest degree of control, greatest cost, greatest latency
- Deploy containers: Better resource utilization
- Platform-as-a-service: Minimal degree of control, YMMV with cost
- Function-as-a-service: Minimal degree of control, least latency, YMMV with cost