WEB ENGINEERING DÜSSELDORF

For those times when you need more than 1 server, or none

AWS Edition: Scalable Architectures

Stefan Christoph
Principal Solutions Architect
Amazon Web Services

John Mousa
Sr. Solutions Architect
Amazon Web Services

Why do we need to scale

No architecture is designed for high scalability on day 1.

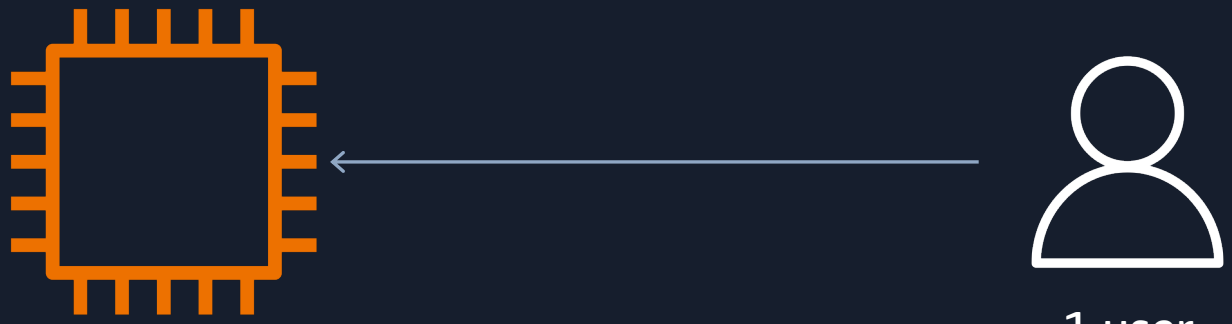
But we'll try.

ME

Today

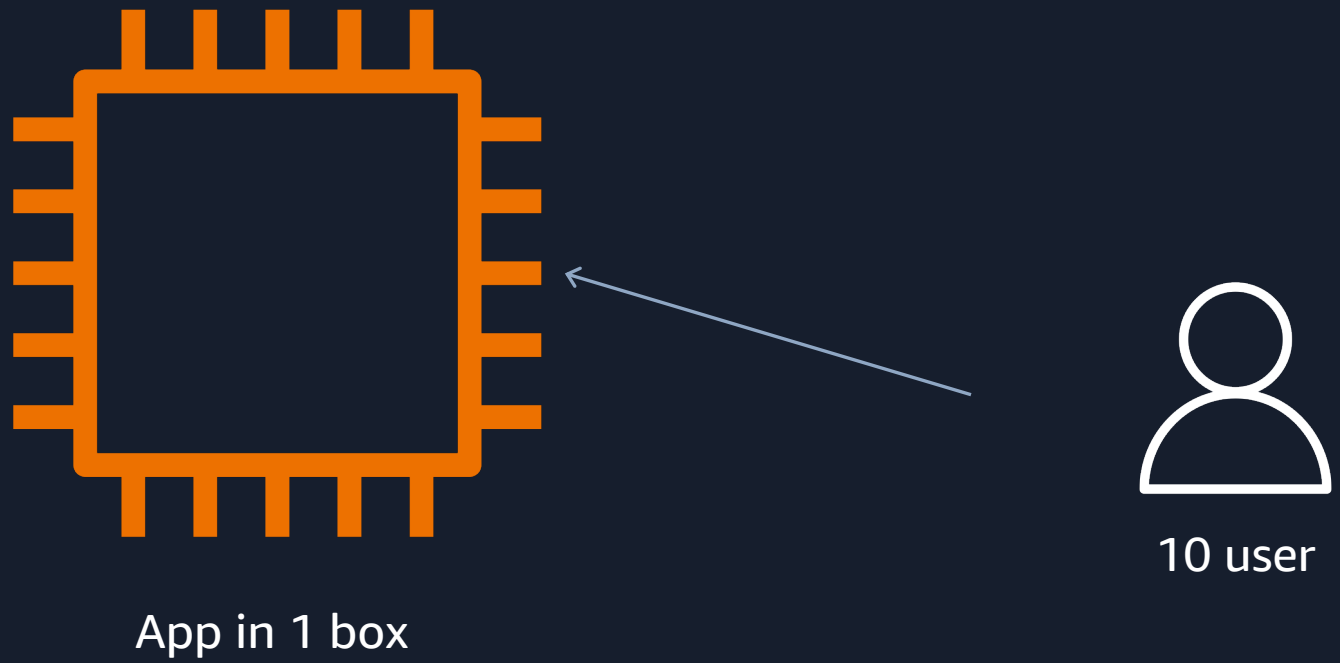
1 User

Simple architecture rules



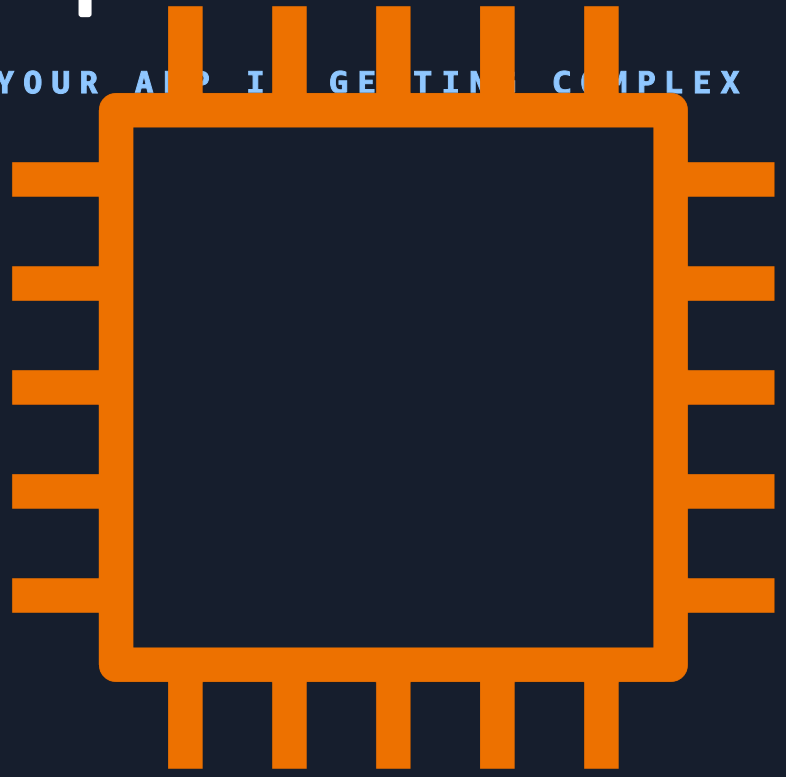
Simple architecture rules

BUT YOUR APP IS GETTING COMPLEX



Simple architecture rules

BUT YOUR AP I GE TIN C MPLEX



App in 1 box

100 user

Simp

BUT YOUR

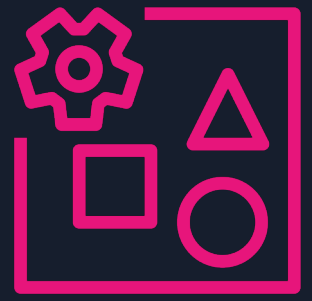
Why do we need to scale

Physical limits

Economy

Functional

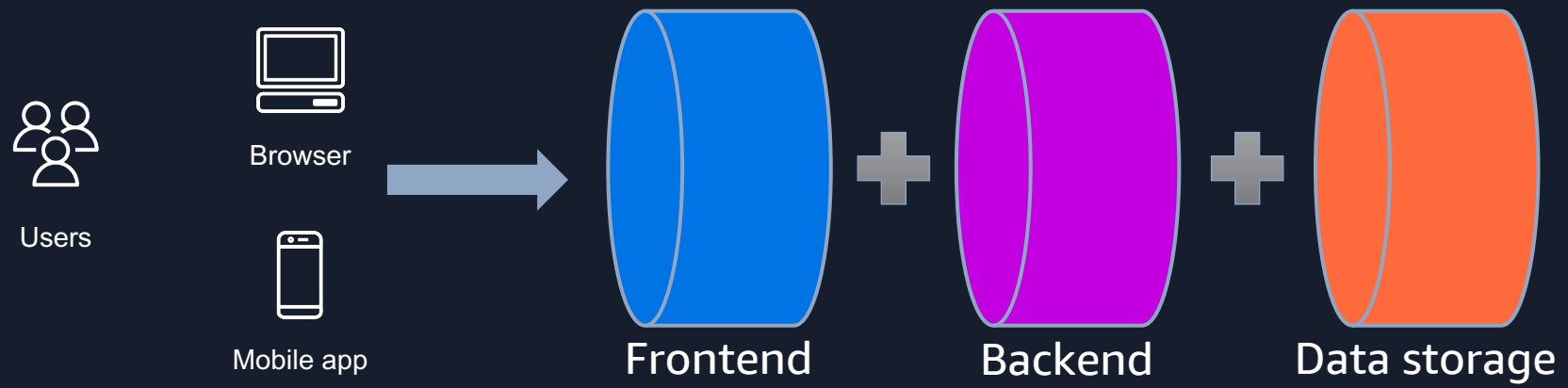
Non Functional



Architecture

Our application revisited

AN ORGANIZATIONAL SEPARATION FOR ECONOMICAL REASONS



Users: >1

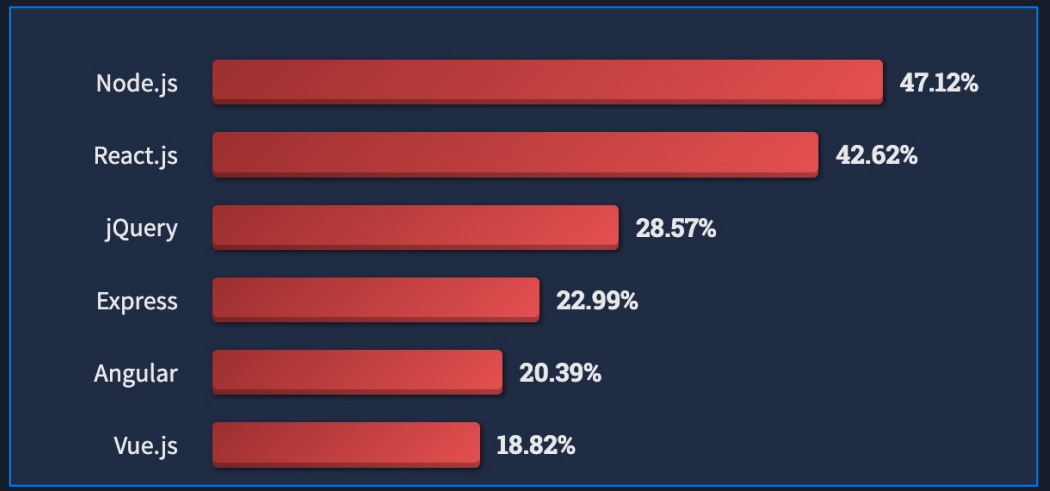
So you and your first customer

What’s changed?

This old way of starting host-based has grown out of favor due mostly to the popularity of frontend framework technology:

- Node.js React, jQuery, Angular, and Vue.js round out top web frameworks and technologies in the Stack Overflow Developer Survey 2022 (see chart/QR link)
  - Flutter and React Native are the two popular cross-platform frameworks
- Growing ecosystem on top of these frameworks
- Growing consulting populations

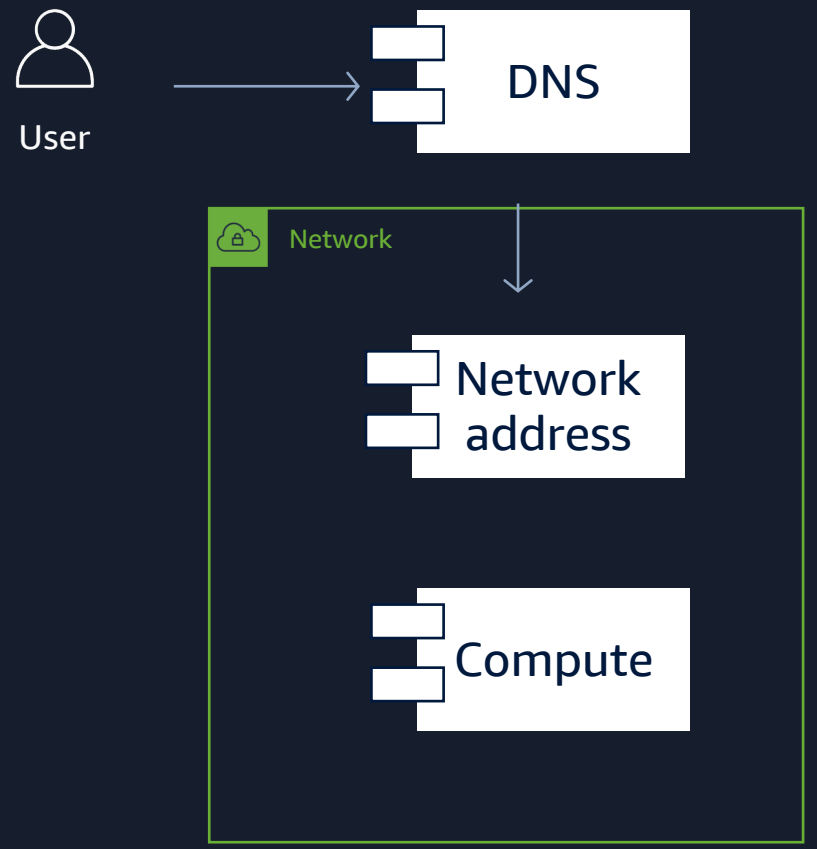
Static frameworks and Server-side rendering (SSR) seemingly de facto standard now



Day 1 the old way – Single instance

Previously you would see a single-instance-based starting architecture that would host the 3 main layers of an application:

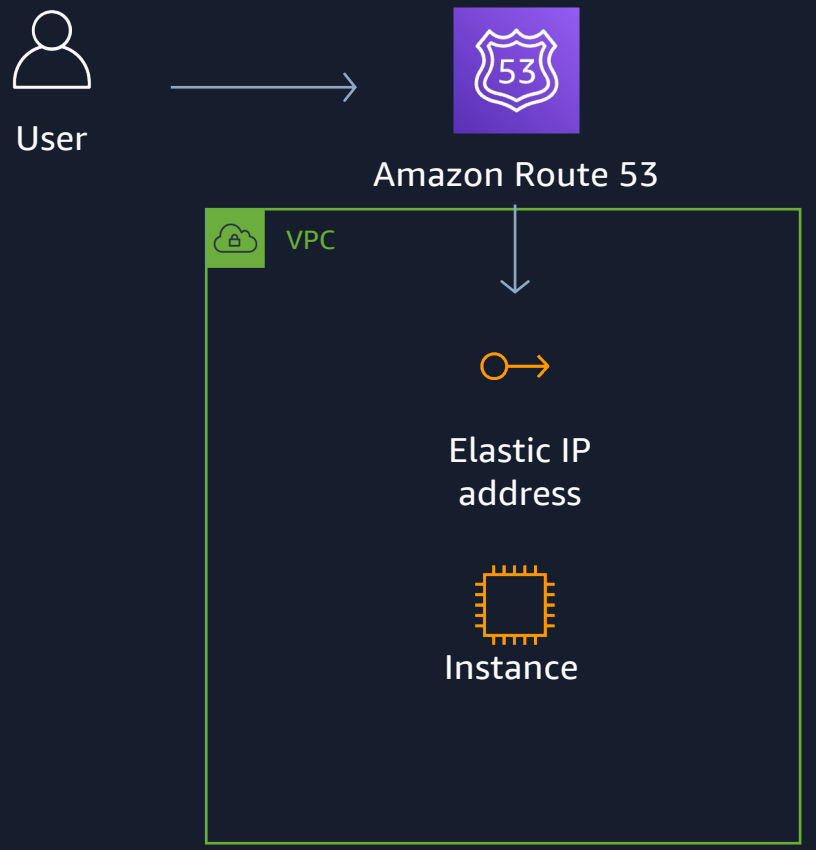
1. Presentation
2. Business
3. Data



In a sense this still happens, but now it's growing "out of style" for modern application development

Day 1 the old way – Single instance

Architecture realized on AWS

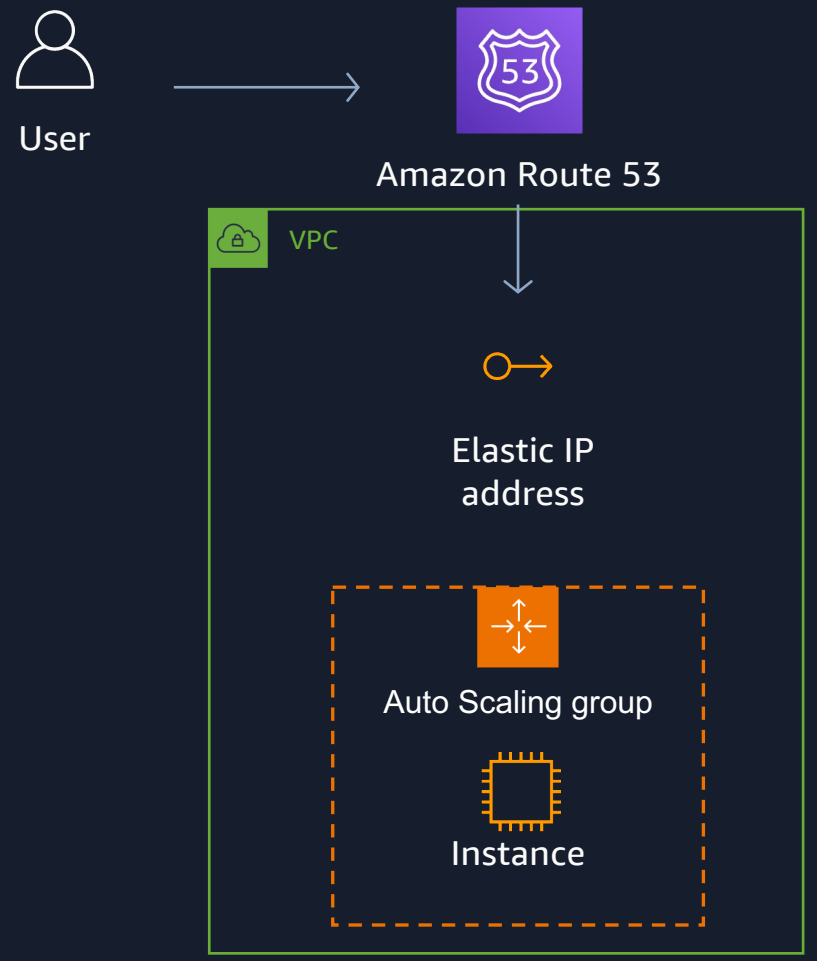


Stop
You're back to the single instance thing

Day 1 the old way – Single instance

Auto scaling

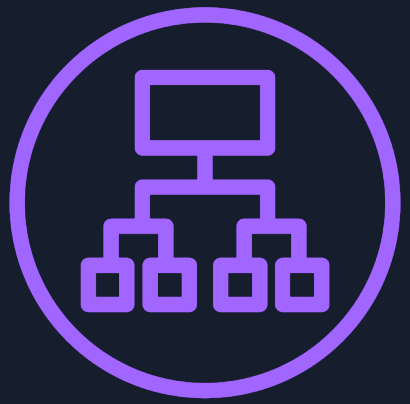
- An *Auto Scaling group* contains a collection of EC2 instances that are treated as a logical grouping
- Now you can add or remove instances based on various metrics like CPU or memory utilization



Exposing business logic to the frontend

THREE OPTIONS FOR EXPOSING AN API

Amazon API Gateway



Application Load Balancer

AWS AppSync

What load balancing choices do we have?

Application Load Balancer (ALB)

Layer 7

Targets
IP, instances, AWS Lambda, containers

Protocols
HTTP, HTTPS, gRPC

Network Load Balancer (NLB)

Layer 4

Targets
IP, instances, ALB, containers

Protocols
TCP, UDP, TLS

Gateway Load Balancer (GWLB)

Layer 3 gateway/ 4 load balancer

Targets
IP, instances

Protocols
IP

Classic Load Balancer (CLB)

Layer 4/7

Targets
EC2-Classic

Protocols
TCP, SSL/TLS, HTTP, HTTPS

AWS Global Accelerator

TCP/UDP

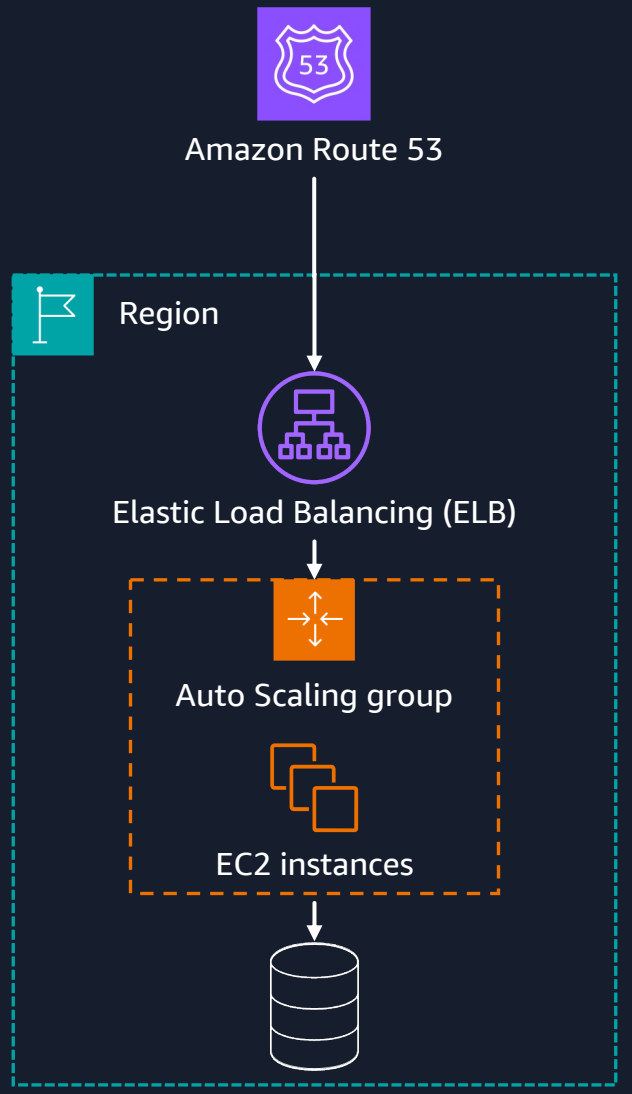
Targets
IP, ALB, NLB

Protocols
TCP, UDP

Users >1: Traditional frontend hosting

Traditional frontend hosting would have you serve your frontend content (HTML, CSS, JavaScript, images, and so on) off of a simple web-serving stack. That stack would minimally be composed of:

- Hosting tier for the webserver app (Nginx, Apache, and so on)
  - Optionally, a shared storage layer
- A load balancer
- A CDN for edge caching



To NoSQL, or not to NoSQL?

Start with SQL databases

Why start with SQL?

Established and well-known technology

Lots of existing code, communities, books, and tools

You aren’t going to break SQL databases with your first millions of users.
No, really, you won’t.*

Clear patterns to scalability

*Unless you are doing something super peculiar with the data or you have massive amounts of it, but even then SQL will have a place in your stack

Ah ha!

You said, “massive amounts of data.”

That’s me.

Multiple terabytes of data in year 1?

Incredibly data-intensive workload?

OK!

You *might* need NoSQL

Why else might you need NoSQL?

- Super low-latency applications
- Metadata-driven data sets
- Highly nonrelational data
- Need schema-less data constructs*
- Rapid ingestion of data (thousands of records per second)
- Massive amounts of data (again, in the multiple terabyte range)

*"Need" != "It's easier to do development without schemas"

But this isn’t most of you. So . . .

Start with SQL databases

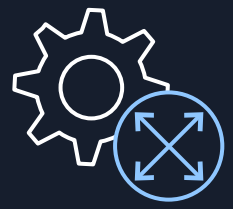
Amazon Aurora

Relational databases built for the cloud – performance and availability of commercial databases at 1/10th the cost

Performance and scalability

Several times faster than standard MySQL and PostgreSQL

15 read replicas



Availability and durability

Fault-tolerant self-healing storage

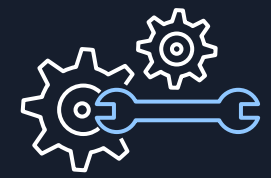
6 copies of data across 3 AZs

Single global database with cross-Region replication

Highly secure

Network isolation

Encryption at rest/transit



Fully managed

Managed by Amazon RDS: no hardware provisioning, software patching, setup, configuration, or backups

The fastest growing service in the history of AWS

Amazon Aurora Serverless v2

On-demand and auto scaling configuration

Automatically scales capacity based on application needs

Simple pay-per-use pricing per second

Next version scales instantly to support demanding applications

Worry-free database capacity management

Users: >100

Users: >1000

Users: > 10,000

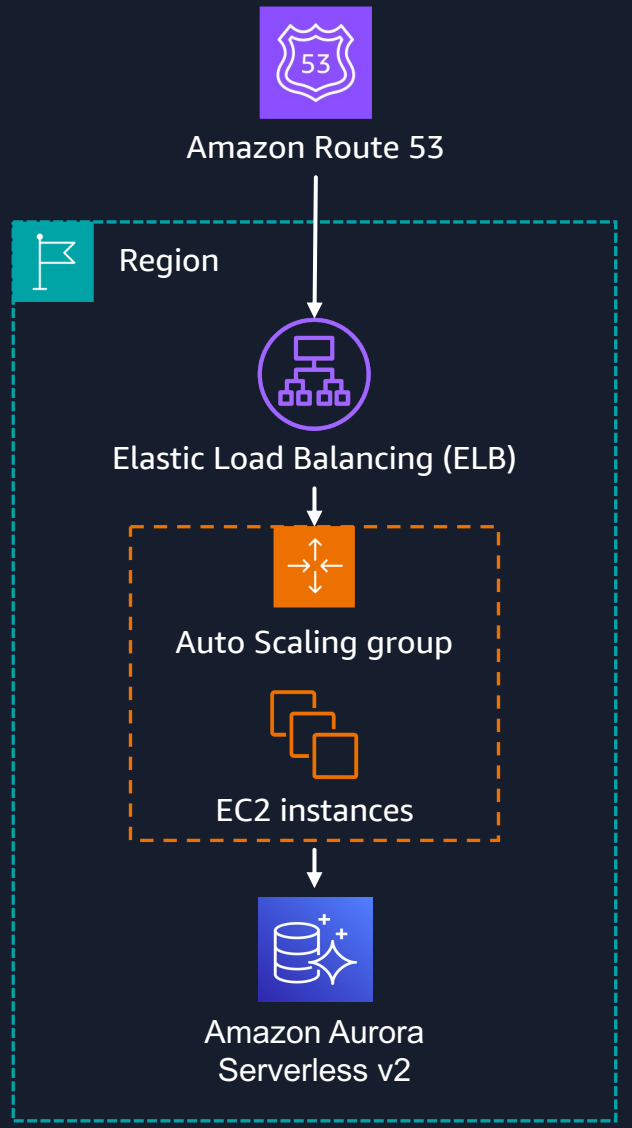
Users > 10,000

maybe?

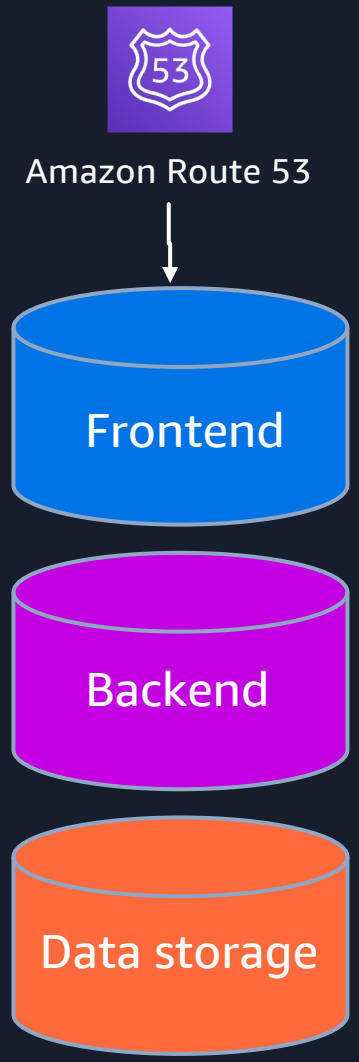
Users: >10,000. What starts to go wrong?

The current stack will scale incredibly far, but the scaling of single tier/monolithic applications can sometimes only go so far. You'll eventually run into issues common in most architectures:

- Varied needs of the product complicating others
- Poor performance in one part impacting other parts
- Slowing queries in the database due to large table sizes/index growth



Let’s dive in



Scaling the frontend

Amazon CloudFront

Built on top of the 410+ Amazon CloudFront PoPs globally

Performance typically comes from

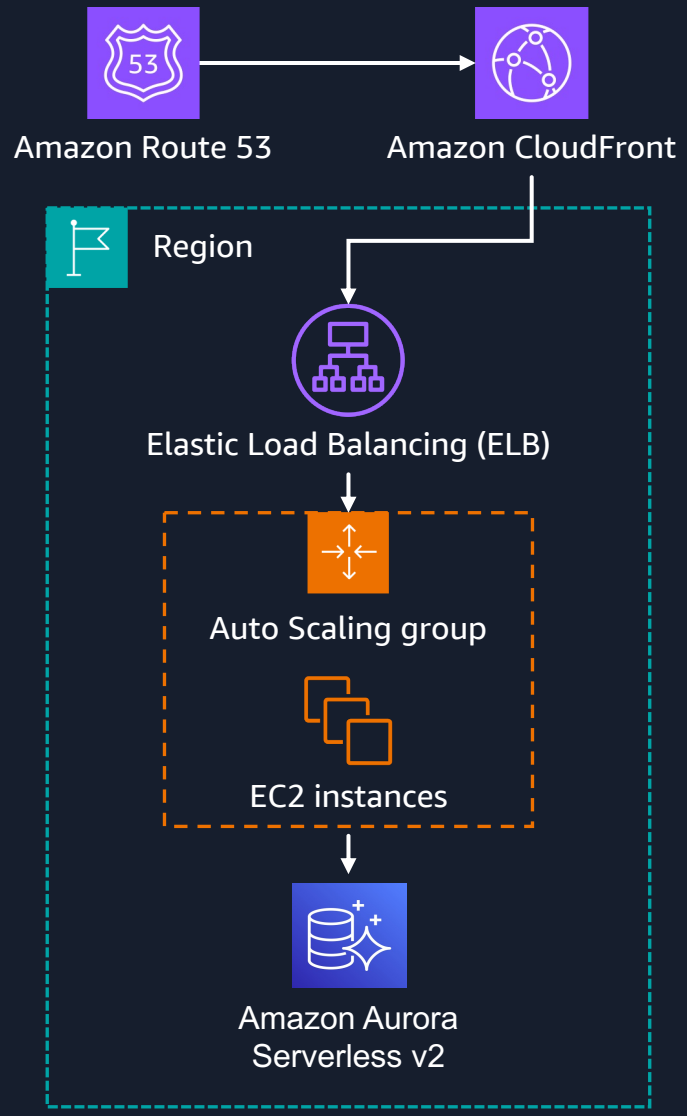
- Tuning frontend code
- Reducing the number of backend calls
- Caching images/JavaScript/CSS effectively
- **Use cases for dynamic content delivery with the CDN**

Amazon CloudFront Points of Presence (PoP)

Users: >10,000. What starts to go wrong?

The current stack will scale incredibly far, but the scaling of single tier/monolithic applications can sometimes only go so far. You'll eventually run into issues common in most architectures:

- Varied needs of the product complicating others
- Poor performance in one part impacting other parts
- Slowing queries in the database due to large table sizes/index growth

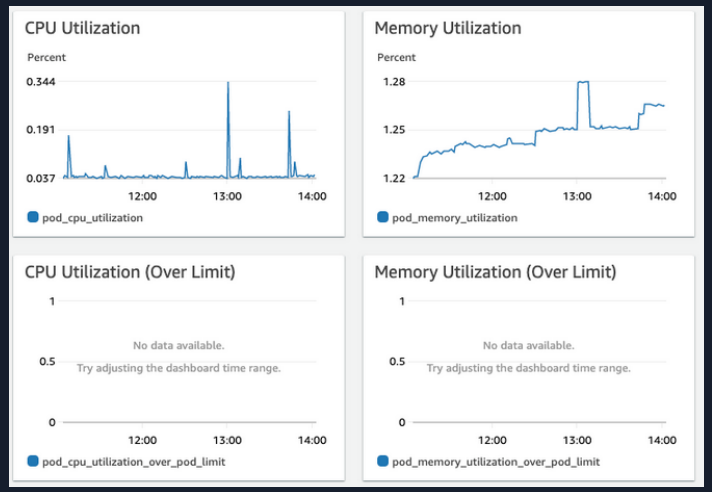


Before we go too much further!

We can’t tune what we aren’t measuring

AWS services for observability

Amazon CloudWatch



AWS X-Ray

Leverage machine learning (ML) to assist you

AWS services for ML-assisted DevOps

Amazon DevOps Guru

Detect unusual behavior, analyze performance, and drive correction of issues

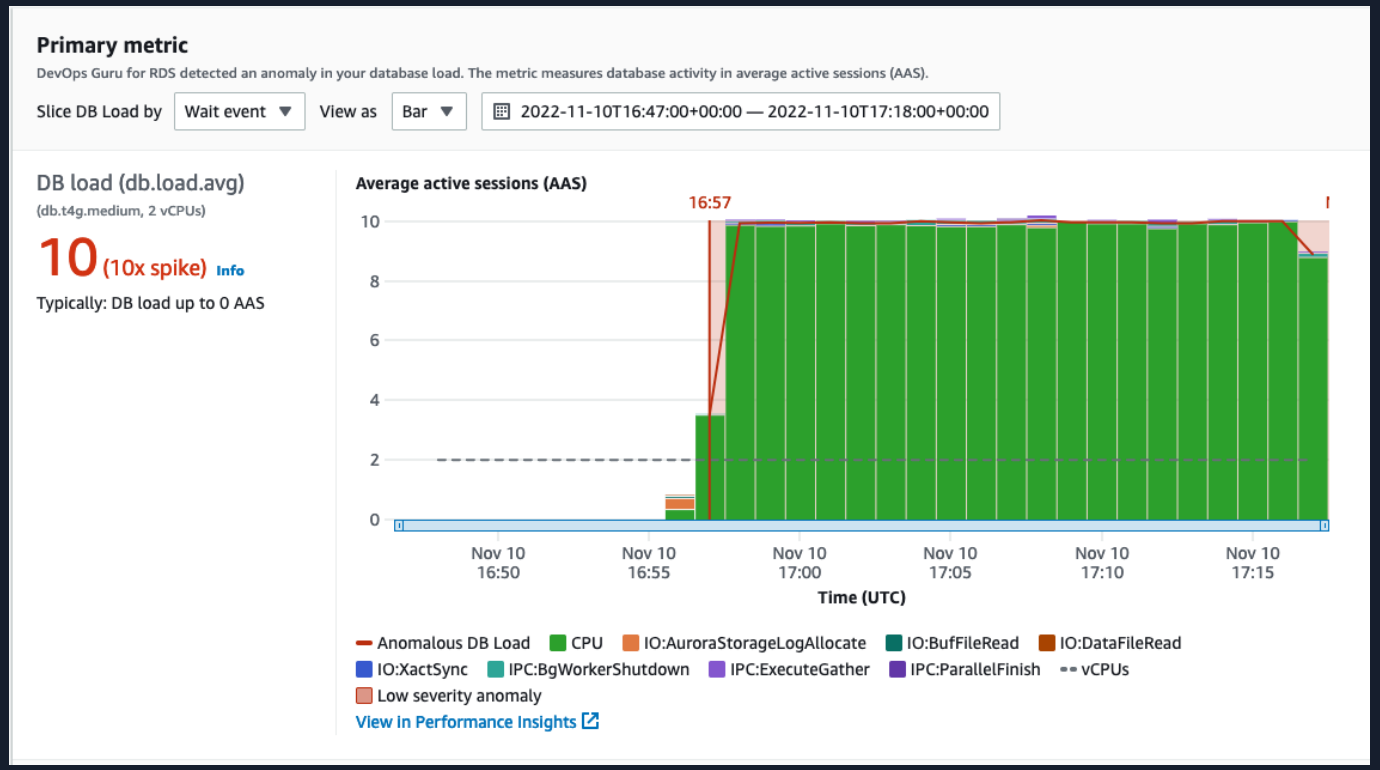
Amazon CodeGuru

Analyze application code for common issues, performance, and cost improvements

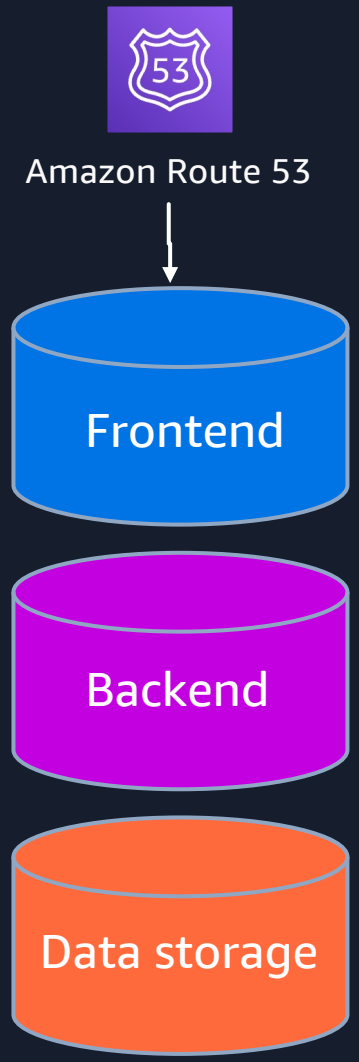
Tuning for scale

With data in hand you can now begin to tackle some of the most common pain points in scaling your application:

- Slow database queries
- Slow API requests
- Failures due to increased traffic
- Service-to-service communication



Let’s dive in



Aurora Serverless v2: scaling

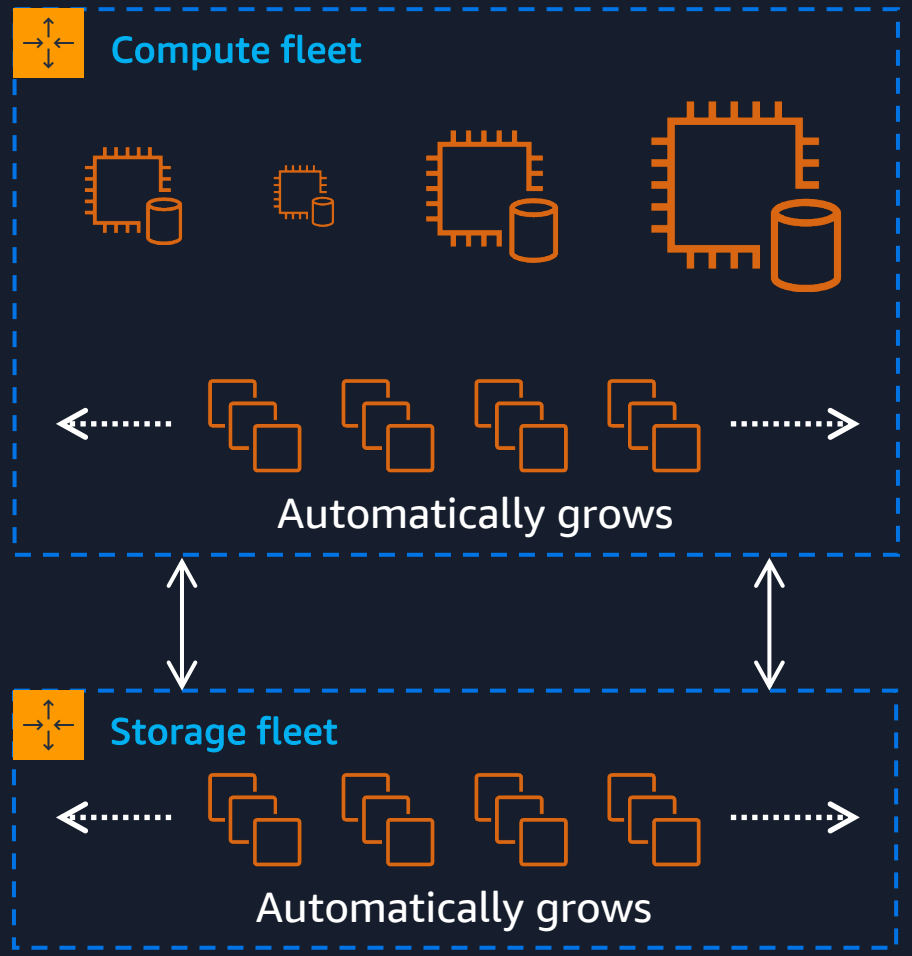
Scales in place in under a second by adding more CPU and memory resources

No impact due to scaling even when running hundreds of thousands of transactions

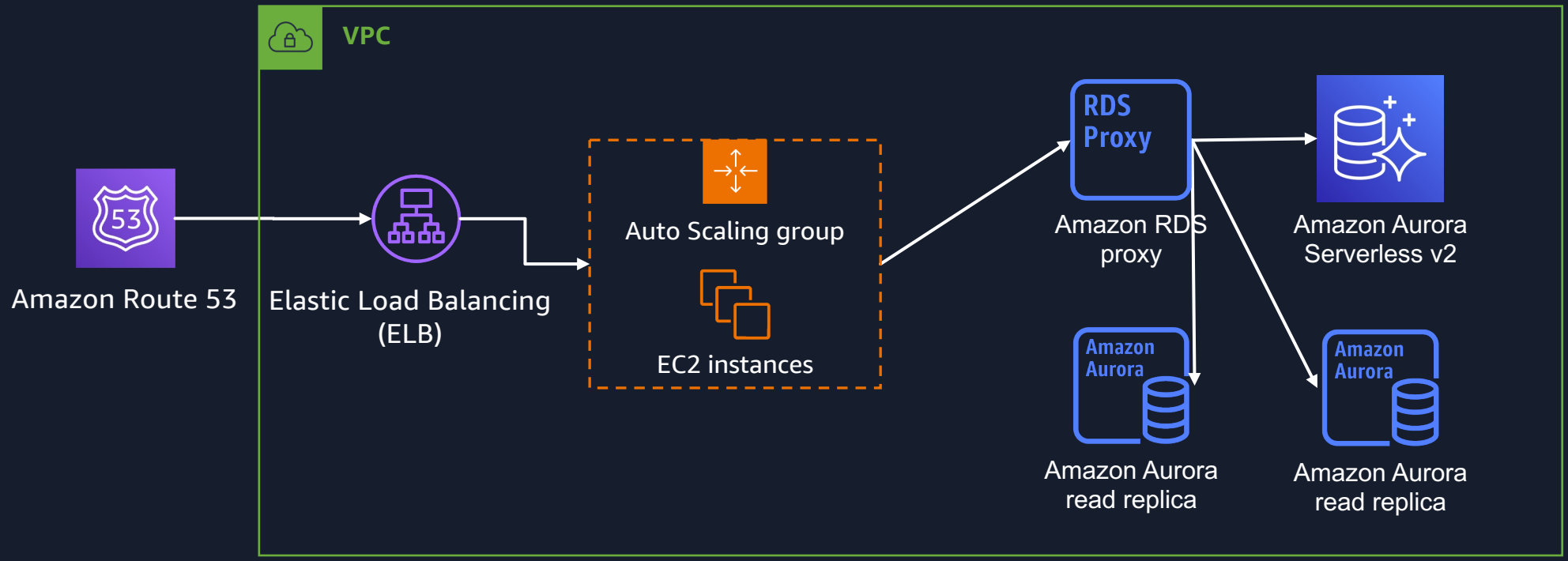
Compute fleet continuously monitored and scaled horizontally for heat management

Background movement of idling instances while preserving state (e.g., buffer pool, connections)

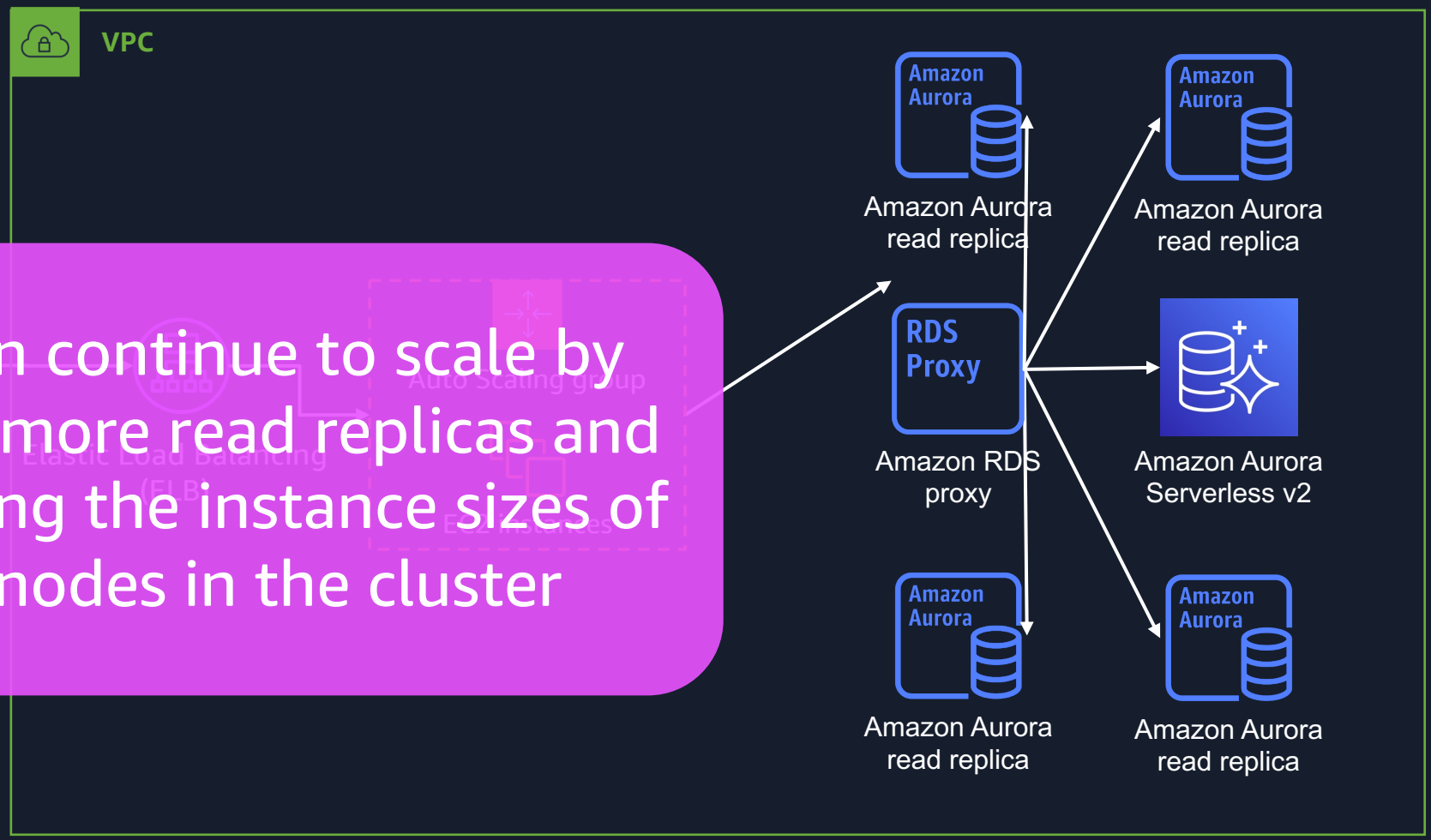
Up to 15x faster scale downs



Scaling Aurora Serverless v2



Scaling Aurora Serverless v2



We can continue to scale by adding more read replicas and increasing the instance sizes of the nodes in the cluster

The best database queries are the ones you never need to make (often).

Me

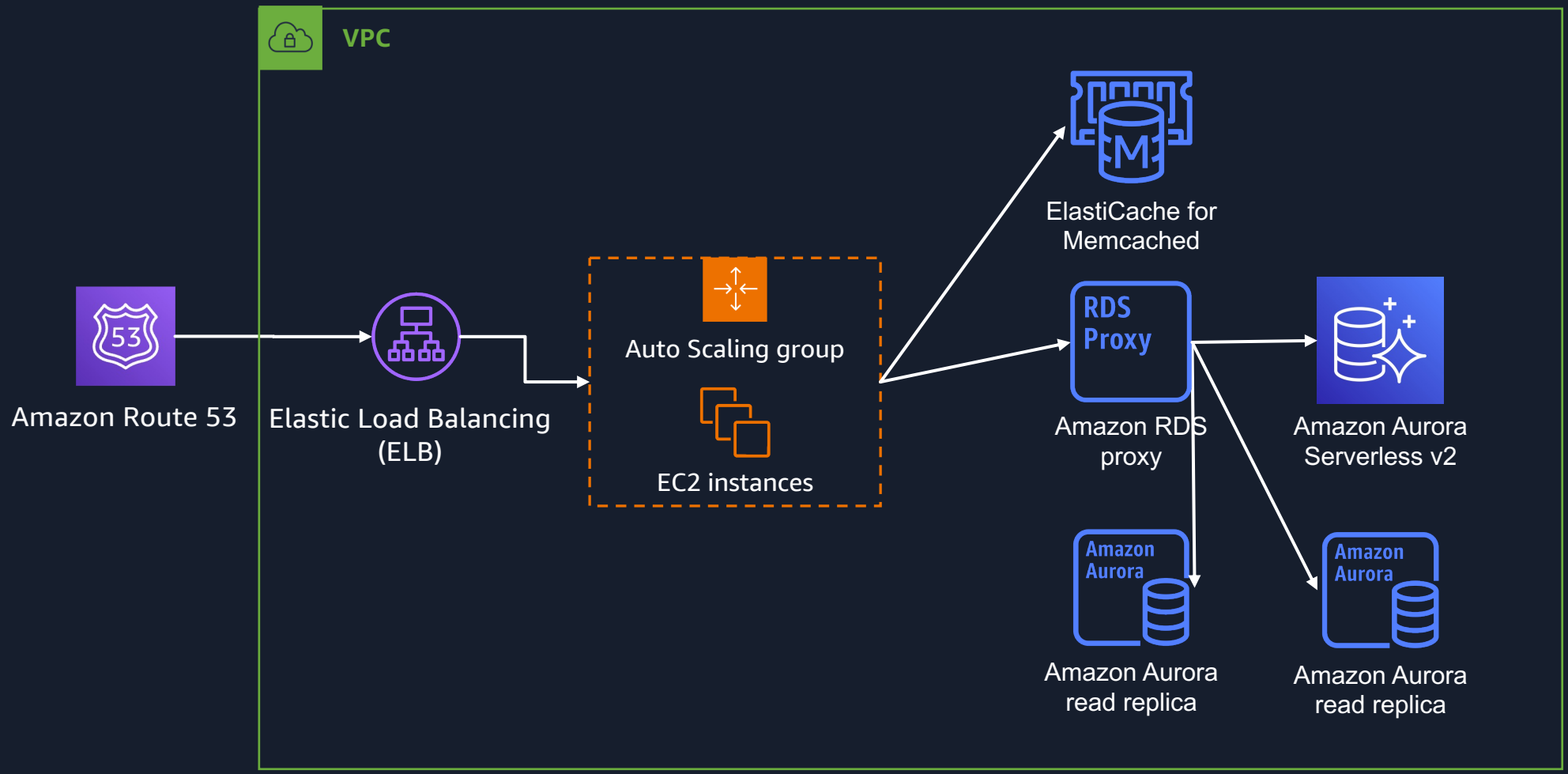
Today

Amazon ElastiCache

Amazon ElastiCache

- Managed Memcached or Redis
- Scale from one to many nodes
- Self-healing (replaces dead instance)
- Single-digit millisecond speeds (usually)
- Multi-AZ deployments for availability

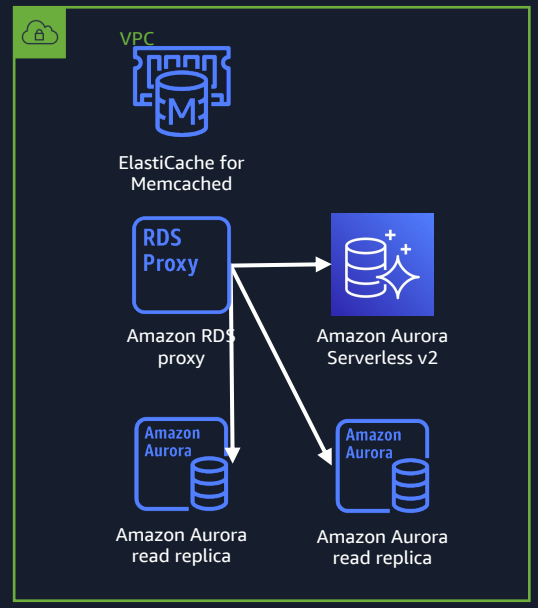
Scaling Aurora Serverless v2



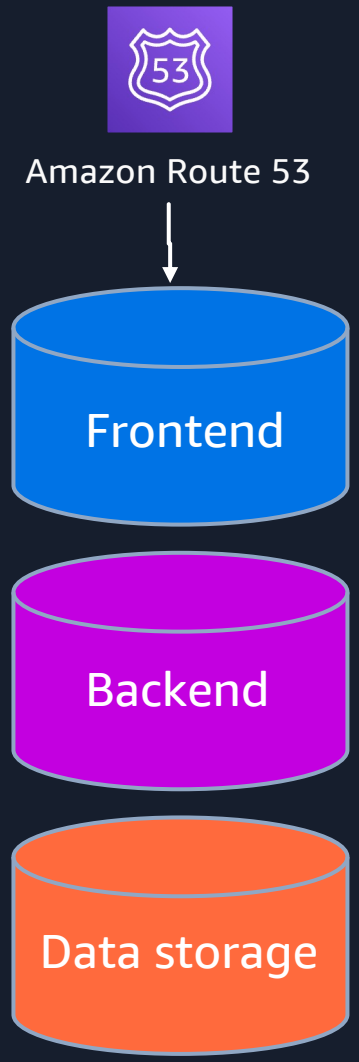
Scaling the data tier

Three main methods for scaling our data tier:

- Increasing the size of the instance(s) used
- Adding read replicas and a proxy to help scale read queries
  - Typically minor application changes
- Using caches to remove queries from even needing to be made
  - requires more significant application changes and new logic to handle



Let’s dive in



Functional

Non Functional

Architecture

You probably guessed it but we need to bring in the architecture hat and do some hard stuff

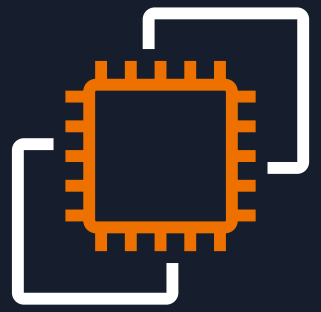
Coupling – integration's magic word

Coupling is a measure of dependencies between connected systems

Coupling isn't binary

Coupling isn't one-dimensional

Options for compute



Amazon EC2

Virtual server instances in the cloud

Amazon ECS, Amazon EKS, and AWS Fargate

Container management service for running Docker on a managed cluster of EC2

AWS Lambda

Serverless compute for stateless code execution in response to triggers

K8s is not a solutions for scale
It’s a solution for resource utilization

Your backend scaling problems will be waiting for you also in containers

OK, I got it!

What should I consider?

Evaluating managed compute on AWS

state

	AWS manages	Customer manages
AWS Lambda Serverless functions	• Data source integrations • Physical hardware, software, networking, and facilities • Provisioning	• Application code
AWS Fargate Serverless containers	• Container orchestration, provisioning • Cluster scaling • Physical hardware, host OS/kernel, networking, and facilities	• Application code • Data source integrations • Security config and updates, network config, management tasks
ECS/EKS Container-management as a service	• Container orchestration control plane • Physical hardware software, networking, and facilities	• Application code • Data source integrations • Work clusters • Security config and updates, network config, firewall, management tasks
EC2 Infrastructure-as-a-Service	• Physical hardware software, networking, and facilities	• Application code • Data source integrations • Scaling • Security config and updates, network config, management tasks • Provisioning, managing scaling, and patching of servers

Users: > 100,000

Users: > 1,000,000

Going the microservices route

Moving to a service-oriented or microservices based architecture is a refactor that requires deep planning across all layers.

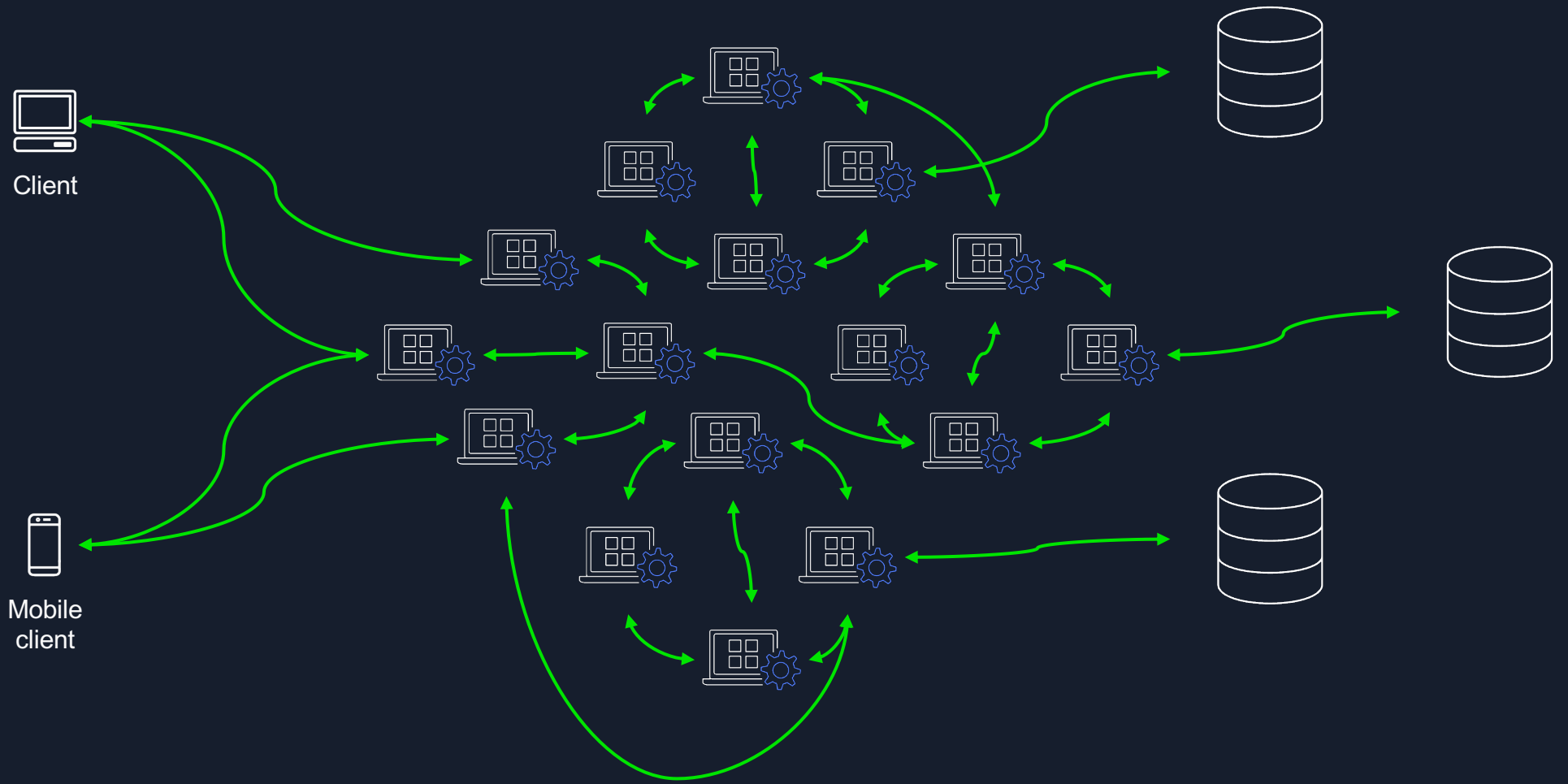
- Start with with the easiest to cut away features/capabilities that don’t involve too many cross-function ties
  - Data domain mapping
  - Business function mapping
- Good time to evaluate other compute technologies for specific needs
- Will need to think about how to “glue” everything together

Shifting functionality to NoSQL

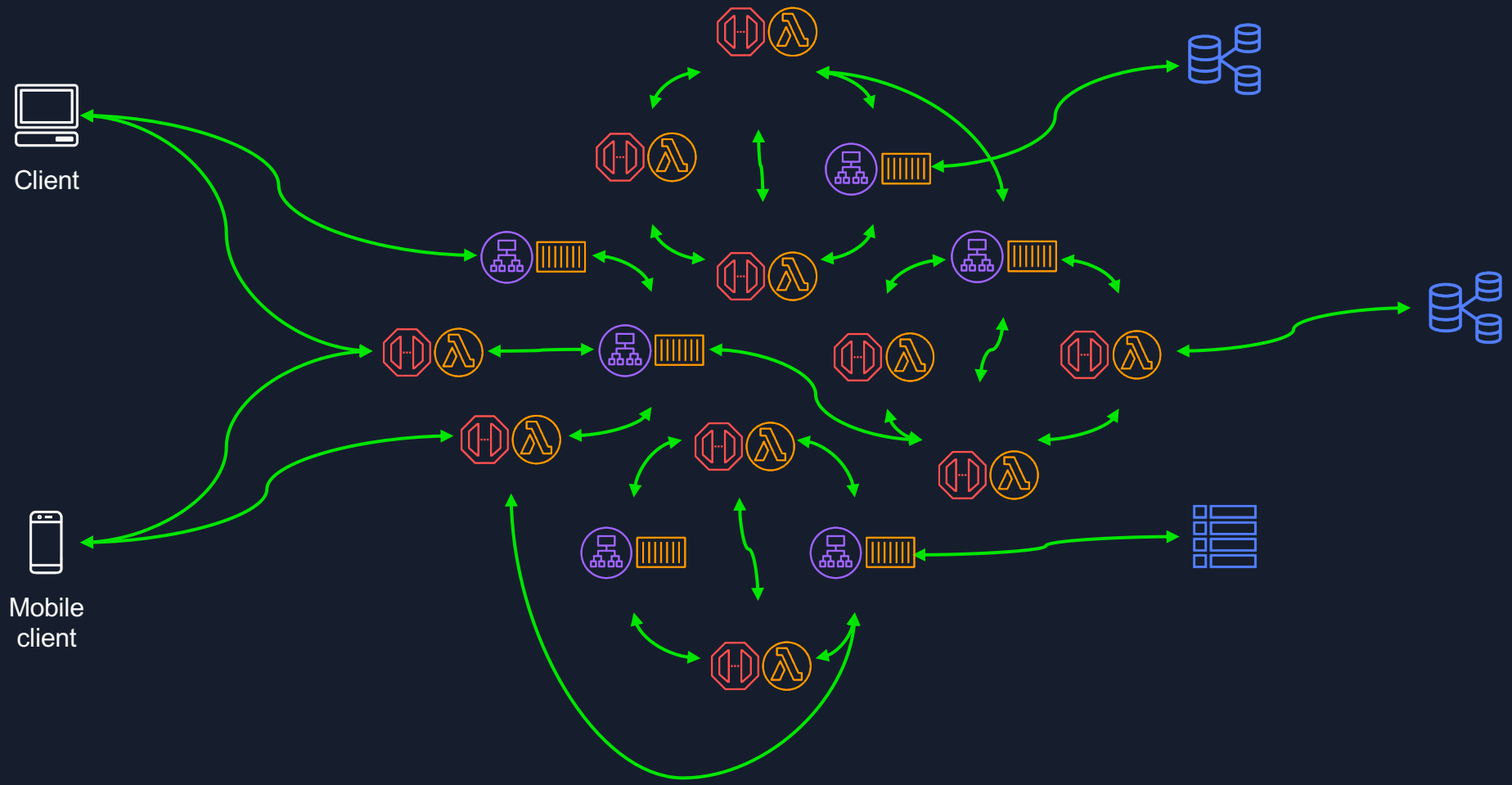
Amazon DynamoDB

- Leverage managed services such as DynamoDB
- Supports massive scale with consistent low latency
- Example use cases
- “Hot” tables
- Metadata/lookup tables
- Leaderboards/scoring
- Temporary data needs (cart data)

The microservices architecture



The microservices architecture



How do I integrate all this mess?

Well that’s why we have Dirk

10 million+ users

Users: >10 million

- Scale smartly by applying different techniques to different services
- One solution may not fit all your use cases
- Work backwards from your functional and non functional requirements
- Look for “best fit” technologies based on need
- The bulk of scaling wins come from doing less, look for opportunities to leverage managed services

To infinity . . .

Thank you!

Please complete the event survey

Stefan Christoph

/in/stefanchristoph

/stechr

John Mousa

/in/johnmousa

/johnmousa