Scaling Web Applications:
Architecture Patterns

Jason Kuruzovich

RPI

April 14, 2026

Readings

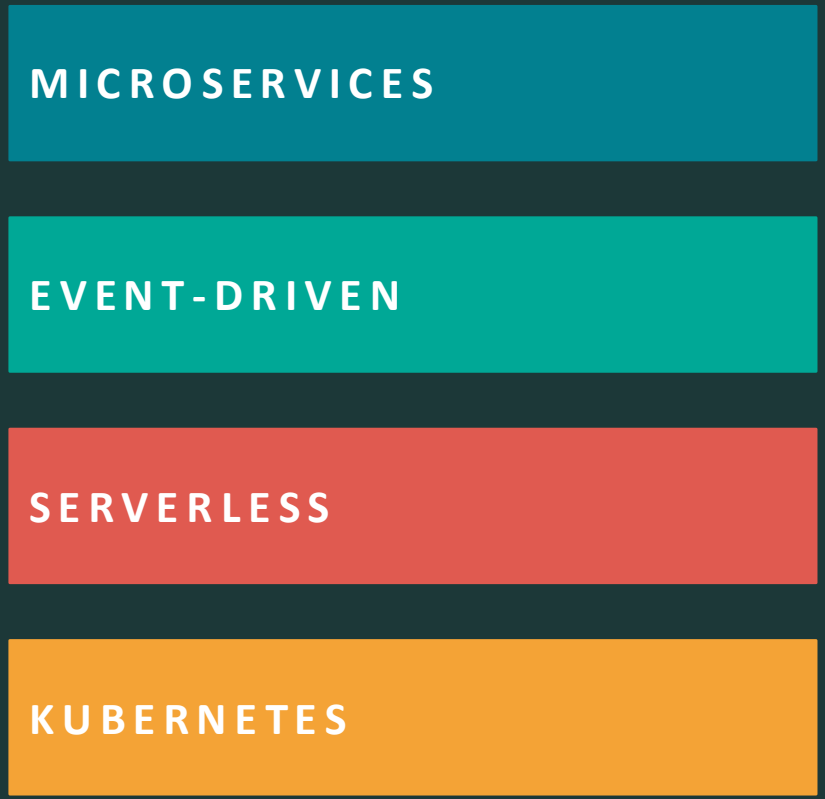
Serverless. https://www.serverless.com/framework/docs/providers/aws/guide/intro
Vercel https://vercel.com/docs/deployments

Web Science Application Development

Scaling Web Applications

Architecture Patterns

Microservices · Event-Driven · Serverless



Jason Kuruzovich, PhD
kuruzj@rpi.edu
https://biggermatrix.com

Where We Are in the Course

Today's focus: how applications scale — choosing between static hosting, containers, and serverless functions.

PART 1

The Scaling Problem

Why architecture decisions matter early

Two Ways to Scale

Vertical Scaling (Scale Up)

- Add more CPU / RAM to existing server
- Simplest path — no code changes needed
- Single point of failure remains
- Hard ceiling: largest instance available
- Expensive at high end
- Downtime often required for upgrades

Horizontal Scaling (Scale Out)

- Add more servers; distribute load
- Near-linear capacity growth
- Eliminates single point of failure
- Requires stateless application design
- Load balancer distributes traffic
- Foundation for cloud-native architectures

→ *Modern cloud architecture defaults to horizontal. Your app design must support it.*

The Monolith: Where Scaling Breaks Down

The Monolith

- Deploy everything to change one line
- Scale the whole app for one bottleneck
- One failure can bring down everything
- Tech lock-in across the entire codebase
- Large teams conflict on the same repo

These pain points are exactly what microservices and event-driven architectures are designed to solve.

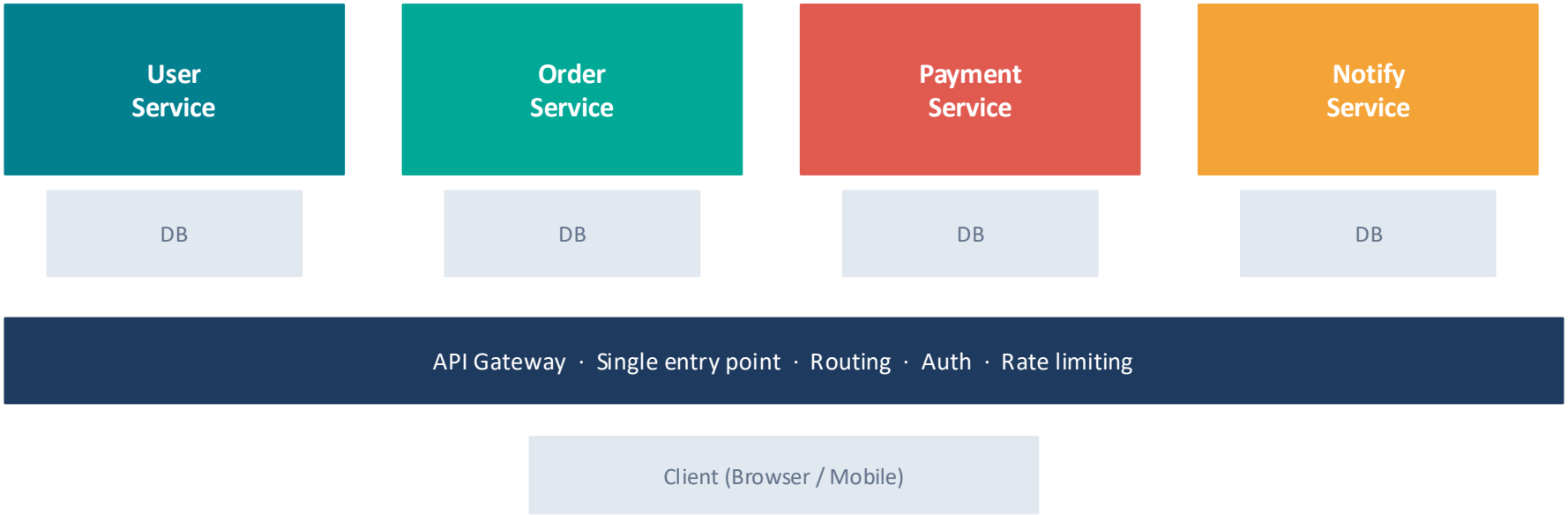
PART 2

Microservices Architecture

Small, independent, deployable services

What Are Microservices?

Each service owns one bounded domain, has its own database, and is independently deployable.



Microservices Design Principles

Single Responsibility

Each service does one thing and owns its data.

Independent Deployability

Deploy any service without touching others.

Decentralized Data

Each service has its own database — no shared schema.

Failure Isolation

A crashed payment service doesn't kill the UI.

Technology Heterogeneity

Python service + Go service + Node service — all fine.

Design for Failure

Circuit breakers, retries, and timeouts are first-class.

Service Communication: Sync vs Async

Synchronous (REST / gRPC)

- Client waits for response
- Simple to reason about — request → response
- Tight coupling between caller and service
- Cascading failures if downstream is slow
- Good for: read queries, auth checks, payment confirm
- Tools: REST (HTTP), gRPC, GraphQL

Asynchronous (Message/Event)

- Caller fires and moves on — no waiting
- Services are loosely coupled
- Durable queues absorb traffic spikes
- Harder to trace and debug end-to-end
- Good for: notifications, order processing, data sync
- Tools: SQS, SNS, Kafka, EventBridge

Rule of thumb: sync for user-facing reads; async for writes and side-effects.

PART 3

Event-Driven Architecture

Loose coupling through events

Event-Driven Architecture (EDA)

Services communicate by publishing and consuming events — no direct calls between producers and consumers.

Decoupling

Producer never knows who consumes the event

Scalability

Consumers scale independently based on queue depth

Resilience

Events are durable — consumers can lag and catch up

Extensibility

Add new consumers without touching producers

Key Event-Driven Patterns

Pub/Sub

One publisher → many subscribers. Decoupled fan-out.
AWS: SNS topics + SQS subscribers.

Event Sourcing

Store every state change as an immutable event log.
Replay to reconstruct any past state.

CQRS

Command Query Responsibility Segregation.
Separate read and write models for scale + clarity.

Saga Pattern

Coordinate multi-step distributed transactions
via events — no two-phase commit needed.

Discussion

Q1 A user places an order. The system must charge payment, reserve inventory, and send a confirmation email. Would you design this synchronously or with events? What are the tradeoffs?

Q2 What happens to pending events if the Notification Service crashes? How does your design handle that?

Q3 How do you debug a bug that spans three event-driven services with no direct call chain?

PART 4

Choosing the Right Deployment Model

Static hosting vs. containers vs. serverless

The Deployment Decision Framework

Ask these questions first:

Does the page change without user input?	→ **Static**
Does it need a backend / database?	→ **Dynamic**
Does traffic spike unpredictably?	→ **Serverless / K8s**
Do you need persistent connections (WebSockets)?	→ **Containers / K8s**
Is there complex business logic per request?	→ **Containers / Lambda**

Static Hosting: S3 + CloudFront

Best for: HTML/CSS/JS single-page apps, blogs, portfolios, documentation sites, marketing pages

How It Works

- Build step outputs pure HTML/CSS/JS files
- Upload to S3 bucket (enable static hosting)
- CloudFront CDN serves from edge globally
- Route 53 maps your custom domain
- No servers to manage — ever
- Scales to millions of users automatically

When to Choose This

- React/Vue/Angular apps with separate API
- Documentation (like Gatsby, Hugo, Next static)
- Cost: effectively zero at low traffic
- Deploys in seconds from GitHub Actions
- SSL free via ACM + CloudFront
- Cannot run server-side code — that goes in Lambda

⚡ *$0.023/GB storage + $0.0085/10k requests — vs. $0.023/hr minimum for a t3.micro EC2*

Complex Applications: Containers & Kubernetes

Best for: full-stack apps, APIs with state, microservices, long-running processes, WebSocket servers

Docker

- Package app + dependencies together
- Runs identically dev → staging → prod
- Dockerfile = reproducible build
- Push to ECR, pull anywhere

ECS / Fargate

- AWS managed container platform
- Fargate: no EC2 to manage
- Auto-scales on CPU/memory/custom metrics
- Good for: teams new to Kubernetes

Kubernetes (EKS)

- Industry standard orchestration
- Rolling deploys, canary, blue-green
- Horizontal pod autoscaling built in
- Best for: large microservice fleets

Kubernetes adds operational complexity — right-size the tool for your team and traffic volume.

Serverless: AWS Lambda

Best for: event-triggered logic, APIs with variable traffic, background jobs, data transforms, IoT pipelines

Lambda Characteristics

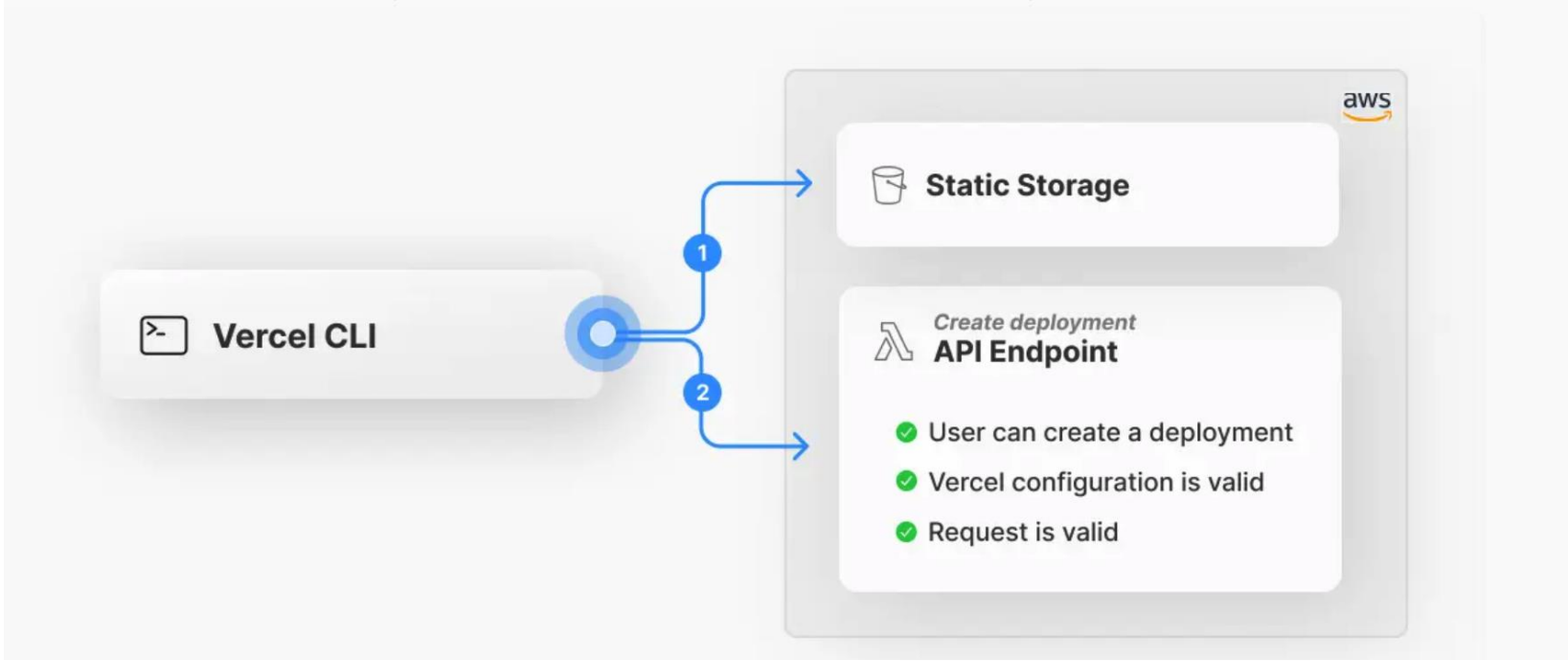
- No server provisioning — ever
- Runs in response to events (HTTP, S3, SQS, cron…)
- Billed per 1ms of execution — zero idle cost
- Scales from 0 to 10,000 concurrent executions
- Max 15-minute execution timeout
- Cold starts: 100ms–1s (mitigated by provisioned concurrency)

Lambda Invocation Triggers

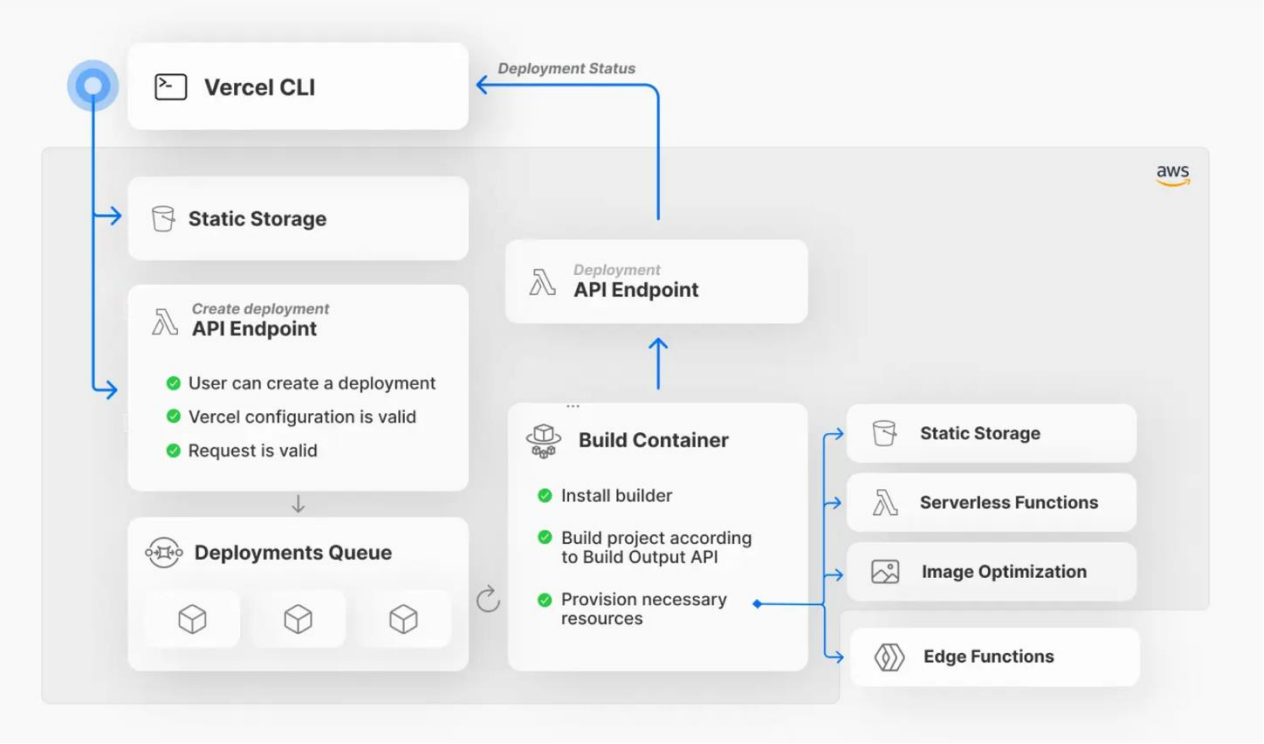
- API Gateway → HTTP request handler
- S3 event → process uploaded file
- SQS / SNS → consume message queue
- DynamoDB Streams → react to DB changes
- EventBridge → scheduled cron rules
- CloudWatch → alert-driven automation

Lambda is the glue layer of event-driven microservices — integrates with virtually every AWS service.

Vercel – Managed Services



Vercel – Managed Services



Vercel vs AWS Lambda comparison:

	Vercel	AWS Lambda
Deploy	`vercel` (1 command)	`npm run build:lambda && serverless deploy`
Config files	Zero	4+ (serverless.yml, handler.js, run.sh, build script)
Cold starts	Optimized by platform	2-5 seconds
Static assets	Automatic CDN edge caching	Served through Lambda (slow) unless you add S3+CloudFront
Custom domain	Built-in + free HTTPS	Route53 + ACM certificate setup
Control	Less (managed platform)	Full (you own everything)
Tradeoff	Vendor lock-in	More complexity

Deployment Model Comparison

Dimension	S3 Static	Containers / K8s	Lambda Serverless
Server management	None	High (ECS) / Very High (K8s)	None
Cost model	Per GB + request	Per hour (running)	Per 1ms execution
Cold start	None	Seconds (pull image)	100ms – 1s
Max runtime	N/A	Unlimited	15 minutes
Auto-scale	Built-in (CDN)	HPA / KEDA	0 → 10k instant
State / sessions	No server state	Yes (Redis, DB)	Stateless only
Best traffic pattern	Steady / bursty	Steady predictable	Bursty / event-driven

PART 5

AWS Lambda Deep Dive

How serverless actually works

Lambda: Under the Hood

Lambda manages the execution environment lifecycle — your job is to write stateless handler code.

Cold Start (First Invocation Only)

- Download code + create execution environment
- Typically 100ms – 1s depending on runtime
- Mitigate: provisioned concurrency, keep-warm pings, small packages

Warm Invocations (Reuse)

- Container reused — init code already ran
- Only handler() executes — very fast
- /tmp (512MB) persists within warm container
- Never store auth tokens in module scope

Authoring a Lambda Function (Python)

Structure your Lambda into init code (runs once) and handler code (runs per invocation):

```python
import json, boto3

# Init code: runs once per cold start
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MenuItems')

def lambda_handler(event, context):
    # Handler: runs every invocation
    response = table.scan()
    return {
        'statusCode': 200,
        'body': json.dumps(response['Items'])
    }
```

Lambda + API Gateway: REST API Pattern

API Gateway Handles

- TLS termination (HTTPS)
- Request routing to Lambda ARN
- Auth (Cognito / Lambda authorizer)
- Rate limiting and throttling
- Request/response transformation

Lambda Handles

- Business logic execution
- Input validation
- Database reads and writes
- Response formatting (JSON)
- Error handling and logging

Lab: Creating Lambda Functions with Python SDK

1

Create the Lambda function

Author a Python handler to query a DynamoDB table of menu items.

2

Configure IAM permissions

Attach an execution role that grants DynamoDB read access.

3

Deploy a function package

Zip the handler and upload via AWS Console or CLI.

4

Test with a mock event

Invoke the function manually and inspect the response payload.

5

Replace the mock endpoint

Wire API Gateway to your Lambda — remove the static stub.

6

Observe with CloudWatch / X-Ray

Trace cold starts, execution duration, and errors end-to-end.

Key Takeaways

1. Static sites belong on S3 + CloudFront — zero servers, near-zero cost, infinite scale.
2. Microservices solve the monolith's scale ceiling by making each domain independently deployable.
3. Event-driven architecture decouples services — producers never know their consumers.
4. Containers (ECS/K8s) are right for long-running, stateful, or CPU-heavy workloads.
5. Lambda is right for event-triggered, short-lived, unpredictably bursty workloads.
6. Pick the simplest model that meets the requirement — over-engineering is a real cost.