Serverless Architecture

COMP2511, CSE, UNSW

Introduction to Serverless Architecture

- Serverless computing allows developers to build and run applications without managing infrastructure.
- Developers focus on deploying individual functions without managing servers.
- Cloud provider dynamically manages server allocation.
- Function is executed in response to events.
- Also known as Function-as-a-Service (FaaS).

- Example Platforms:
  - AWS Lambda: Most popular serverless platform, integrated with the entire AWS ecosystem
  - Azure Functions: Serverless platform for Microsoft Azure users
  - Google Cloud Functions: Lightweight solution for Google Cloud services
  - IBM Cloud Functions: Based on Apache OpenWhisk

How Serverless Works

- User sends request (e.g., API call)
- API Gateway receives and triggers a Lambda/Function
- Function processes data and interacts with services (DB, storage, web services)
- Result returned to user

- Example:
  1. An S3 bucket (cloud storage) uploads an image
  2. The event triggers a Lambda function to resize the image
  3. The function stores resized image in another S3 location (cloud storage)

Example: AWS Lambda

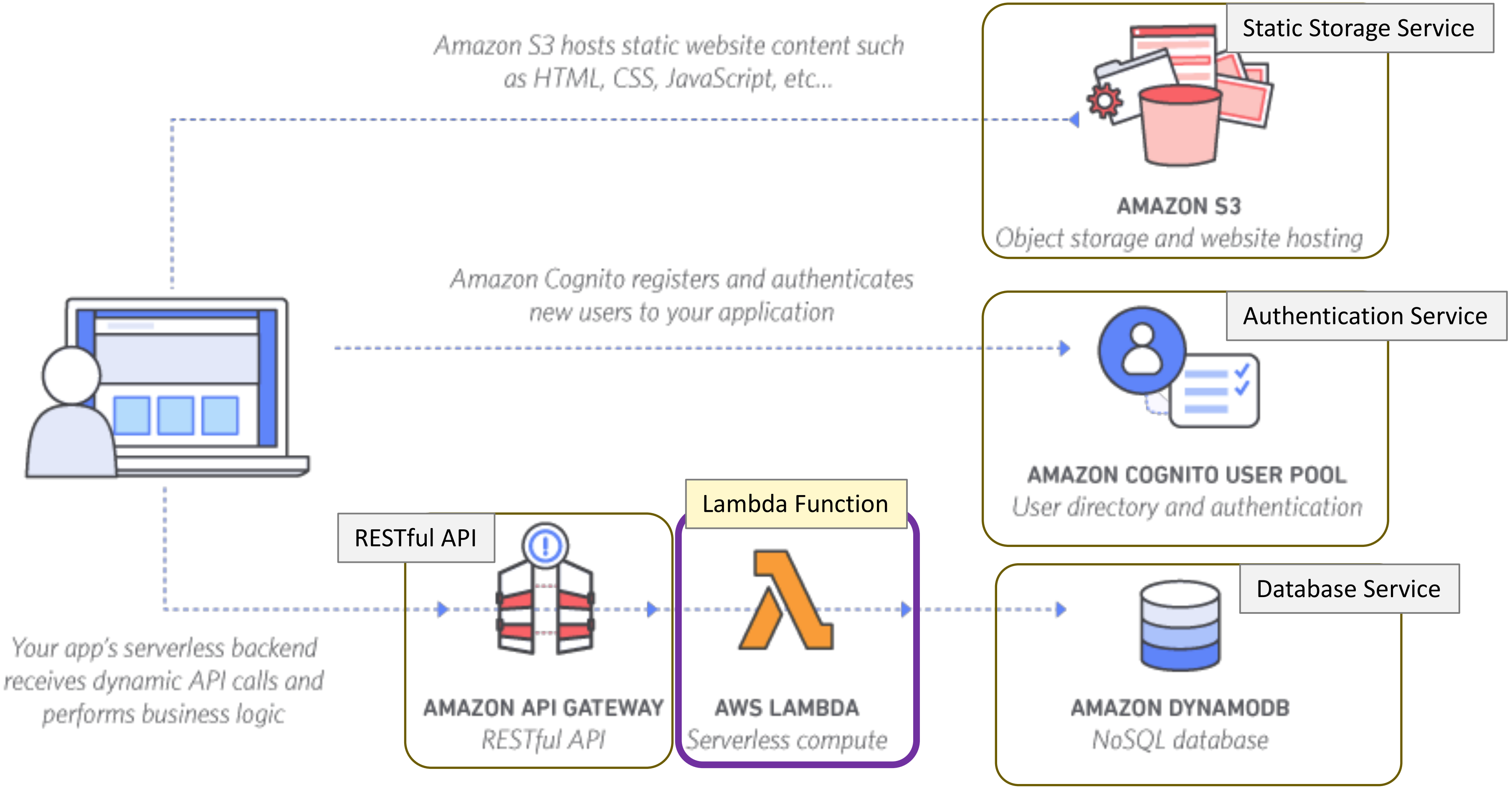

Diagram from https://aws.amazon.com/lambda/web-apps/

Key Characteristics

- Auto-scaling: Instantly handles thousands of concurrent executions
- Faster time-to-market: Developers focus on business logic, not infrastructure
- High availability: Functions are distributed across multiple availability zones
- Event-driven: Executes on triggers like HTTP requests, file uploads, or database changes.
- Micro-billing: You pay only for execution time, usage-based cost.
- Short-lived functions: Ideal for tasks that complete quickly.

Serverless Use Cases

- Form submission triggers a Lambda to store data in DynamoDB.
- Google Cloud Functions reacts to Firebase database changes and sends real-time notifications to users.
- Lambda automatically resizes images uploaded to S3 for use in different display formats.
- An e-commerce website uses Azure Functions to handle inventory updates on-demand.
- AWS Lambda processes incoming JSON health data from IoT devices, generates alerts if required, and stores data in Amazon DynamoDB for further analysis.

Serverless Design Principles

- Stateless: Don’t rely on local memory; use shared storage (e.g., S3, DynamoDB)
- Event-driven: Design workflows around events, not request-response chains
- Minimal and composable functions: Keep single-responsibility per function
- Use queues/pubs/subs: Decouple flows using queues or Publish-subscribe messaging services

Limitations and Challenges

Cold starts:

- ❖ Latency when functions are idle for a while (especially for JVM/.NET)
- ❖ Mitigation: Use warm-up plugins or provisioned concurrency

Vendor lock-in:

- ❖ Tied to provider's ecosystem (e.g., AWS SDKs, IAM policies)

Observability:

- ❖ Harder to trace request flows across functions
- ❖ Solution: Use distributed tracing (e.g., AWS X-Ray, OpenTelemetry)

Resource limits:

- ❖ Timeout (after a few mins on AWS Lambda)
- ❖ Memory and ephemeral storage constraints

Comparison: Serverless vs. Microservices

Feature	Microservices (Containers)	Serverless (Functions)
Deployment Unit	Container	Function
Management	DevOps / CI/CD pipeline	Fully managed by provider
Cost Model	Fixed per compute unit	Per request, per execution time
Scaling	Container autoscaling	Scales with invocations
Startup Time	Low latency (warm)	Cold starts may delay execution
Monitoring	Full stack observability	Requires custom integration

Summary

- Serverless abstracts server management and reduces operational burden
- Works best for stateless, event-driven, and high-concurrency use cases
- Challenges include observability, cold starts, and vendor-specific tooling
- Ideal as a lightweight, cost-effective architecture for modern cloud-native apps