WordMaps

Serverless Web Application for Word Graph Exploration

Juan López de Hierro

juanlopezdehierro@gmail.com

Project deployed on AWS
https://d2euump6bzpha5.cloudfront.net

December 17, 2025

Contents

Abstract

This document presents WordMaps, a serverless web application implementing a word graph exploration system for 3, 4 and 5-letter English words. The application uses graph search algorithms to find optimal paths between words that differ by a single character. The system is built with modern AWS-based architecture using Lambda, API Gateway, S3, and CloudFront, demonstrating the capabilities and advantages of serverless computing for specialized web applications.

1 Introduction

1.1 Motivation

The word transformation problem is a classic computer science challenge that combines graph theory, data structures, and search algorithms. WordMaps applies this concept in a modern web environment, allowing users to interactively explore relationships between words.

1.2 Objectives

The main objectives of the project are:

- Implement an efficient in-memory word graph where each node represents a 3, 4 or 5-letter word
- Develop optimal search algorithms to find paths between words
- Create an interactive and responsive web interface
- Demonstrate serverless architecture usage on AWS
- Minimize operational costs using AWS Free Tier services

1.3 Scope

The project encompasses:

- Backend developed in Spring Boot 3 with Java 17
- Frontend built with React 18 and Vite
- Fully serverless infrastructure on AWS
- Implementation of BFS algorithm for path finding
- RESTful API for graph operations

2 System Architecture

2.1 General Architecture

WordMaps implements a three-tier architecture distributed across AWS services:

1. **Presentation Layer**: React application served from S3 via CloudFront
2. **Application Layer**: Lambda Function running Spring Boot
3. **Routing Layer**: API Gateway HTTP API v2

2.2 AWS Components

2.2.1 CloudFront Distribution

CloudFront acts as a global CDN, distributing static frontend content from edge locations near users. It provides:

- Automatic SSL/TLS termination
- Static content caching
- GZIP/Brotli compression
- Basic DDoS protection

2.2.2 S3 Bucket

The S3 bucket stores frontend static files (HTML, CSS, JavaScript). Configuration includes:

- Restricted access via Origin Access Identity (OAI)
- Website configuration for SPA routing
- Versioning disabled to reduce costs

2.2.3 API Gateway HTTP API

API Gateway HTTP API v2 offers advantages over REST API:

- Lower latency (up to 60% faster)
- Lower cost (up to 70% cheaper)
- Native Lambda integration
- Optimized payload format 2.0

2.2.4 Lambda Function

The Lambda function runs the Spring Boot application with the following features:

- Runtime: Java 17 (Amazon Corretto)
- Memory: 2048 MB
- Timeout: 30 seconds
- SnapStart: Enabled to reduce cold starts
- Environment variables: JVM optimizations

2.3 Data Flow

The typical request flow is:

1. User accesses `https://d2euump6bzpha5.cloudfront.net`
2. CloudFront serves the frontend from S3
3. React application makes AJAX request to API
4. API Gateway receives request and transforms to HTTP API v2 format
5. Lambda processes request with Spring Boot
6. BFS algorithm finds optimal path
7. JSON response is returned to user

3 Technology Stack

3.1 Backend

3.1.1 Spring Boot 3

Spring Boot was chosen for:

- Mature and well-documented ecosystem
- Robust dependency injection
- Native AWS Lambda support via aws-serverless-java-container
- Ease of testing

3.1.2 Java 17

Java 17 LTS offers:

- Records for immutable data modeling
- Improved pattern matching
- Better performance than Java 11
- Full support in AWS Lambda

3.2 Frontend

3.2.1 React 18

React provides:

- Component-based architecture
- Virtual DOM for efficient rendering
- Hooks for state management
- Extensive library ecosystem

3.2.2 Vite

Vite replaces Webpack offering:

- Instant Hot Module Replacement
- Significantly faster build times
- Minimal configuration
- Optimized tree-shaking

3.2.3 TailwindCSS

TailwindCSS enables:

- Rapid development with utility classes
- Reduced bundle size with purging
- Responsive design without custom media queries
- Visual consistency

3.3 Infrastructure

3.3.1 AWS SAM

SAM (Serverless Application Model) facilitates:

- Declarative infrastructure definition
- Local Lambda function testing
- Automated deployment
- CloudFormation integration

4 Implementation

4.1 Graph Structure

The word graph is represented using an adjacency list:

```java
private Map<String, List<String>> adjacencyList;
public void buildGraph(Set<String> words) {
    for (String word : words) {
        adjacencyList.put(word, new ArrayList<>());
        for (String otherWord : words) {
            if (differsByOneLetter(word, otherWord)) {
                adjacencyList.get(word).add(otherWord);
            }
        }
    }
}
```

4.2 BFS Algorithm

The Breadth-First Search implementation guarantees finding the shortest path:

```java
public Route findFastestRoute(String origin, String destination) {
    Queue<String> queue = new LinkedList<>();
    Map<String, String> parent = new HashMap<>();
    Set<String> visited = new HashSet<>();

    queue.offer(origin);
    visited.add(origin);

    while (!queue.isEmpty()) {
        String current = queue.poll();

        if (current.equals(destination)) {
            return reconstructPath(origin, destination, parent);
        }

        for (String neighbor : getNeighbors(current)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                parent.put(neighbor, current);
                queue.offer(neighbor);
            }
        }
    }

    return null; // No path found
}
```

4.3 Lambda Handler

The Lambda handler integrates Spring Boot with AWS:

```java
public class LambdaHandler implements RequestStreamHandler {
    private static SpringBootLambdaContainerHandler<
        HttpApiV2ProxyRequest, AwsProxyResponse> handler;

    static {
        try {
            handler = SpringBootLambdaContainerHandler
                .getHttpApiV2ProxyHandler(WordMapsApplication.class);
        } catch (ContainerInitializationException e) {
            throw new RuntimeException(
                "Could not initialize Spring Boot application", e);
        }
    }

    @Override
    public void handleRequest(
            InputStream inputStream,
            OutputStream outputStream,
            Context context) throws IOException {
        handler.proxyStream(inputStream, outputStream, context);
    }
}
```

4.4 API Endpoints

The API exposes the following endpoints:

Method	Route	Description
GET	/api/routes/fastest	Find optimal path
GET	/api/graph/stats	Graph statistics
GET	/api/graph/top-connected	Most connected words
GET	/api/graph/isolated	Isolated words
GET	/api/words/{word}/exists	Verify existence
GET	/api/words/{word}/neighbors	Get neighbors
GET	/api/words/search	Pattern search

Table 1: Available API endpoints

5 Deployment Process

5.1 Deployment Pipeline

Deployment follows these steps:

5.1.1 Backend Build

```bash
cd wordmaps-backend
mvn clean package -DskipTests
```

Generates an all-in-one JAR with all dependencies included.

5.1.2 SAM Build and Deploy

```
sam build
sam deploy
```

SAM packages the application and creates/updates the CloudFormation stack.

5.1.3 Frontend Build

```
cd wordmaps-frontend
npm install
npm run build
```

Vite generates optimized assets in the `dist/` directory.

5.1.4 S3 Sync

```
aws s3 sync ./wordmaps-frontend/dist \
    s3://wordmaps-frontendbucket-wfghnd4ynw5r --delete
```

5.1.5 CloudFront Invalidation

```
aws cloudfront create-invalidation \
    --distribution-id E3PT8S6ST3CUNS \
    --paths "/*"
```

5.2 Issues Resolved During Deployment

5.2.1 502 Bad Gateway Error

Cause: Incompatibility between `PayloadFormatVersion: "1.0"` in template and `HttpApiV2ProxyHa` in code. **Solution:** Update template.yaml to `PayloadFormatVersion: "2.0"`.

5.2.2 YAML Indentation

Cause: `Environment` block incorrectly indented under `Architectures`. **Solution:** Correct indentation by moving `Environment` to same level as `Architectures`.

5.2.3 CloudFront Cache

Cause: CloudFront serving cached versions of old frontend. **Solution:**

- Configure `Cache-Control` headers on index.html
- Invalidate CloudFront cache after each deployment

6 Performance Metrics

6.1 Response Times

Measurements show:

Endpoint	Cold Start	Warm Start
/api/routes/fastest	2.5s	120ms
/api/graph/stats	2.3s	65ms
/api/words/{word}/neighbors	2.2s	45ms

Table 2: Measured latencies (with SnapStart enabled)

6.2 Applied Optimizations

1. **SnapStart**: Reduces cold starts by ˜60%
2. **Tiered Compilation**: JVM optimization for fast startup
3. **2048 MB Memory**: Balance between cost and performance
4. **CloudFront Caching**: 24-hour TTL for static assets

6.3 Operational Costs

Under AWS Free Tier, the project operates at no cost:

- Lambda: 1M invocations/month (free)
- API Gateway: 1M requests/month (free first 12 months)
- S3: 5GB storage (free)
- CloudFront: 1TB transfer/month (free first 12 months)

7 Results

7.1 Verified Functionality

The production-deployed application demonstrates:

- Successful path finding between words (e.g., CAT → FAT)
- Consistent and correct BFS algorithm responses
- High availability through AWS managed services
- Acceptable performance for end users

7.2 Transformation Example

For the CAT to FAT search, the system returns:

```
{
  "origin": "CAT",
  "destination": "FAT",
  "path": ["CAT", "FAT"],
  "steps": 1,
  "routeType": "FASTEST",
  "difficulty": "EASY",
  "transformations": [
    {
      "from": "CAT",
      "to": "FAT",
      "stepNumber": 1,
      "description": "Change 'C' -> 'F' at position 1"
    }
  ]
}
```

7.3 Complexity Analysis

7.3.1 Space Complexity

- Graph: $O(V + E)$ where V = number of words, E = connections
- BFS: $O(V)$ for auxiliary structures (queue, visited, parent)

7.3.2 Time Complexity

- Graph construction: $O(V^2 \cdot L)$ where L = word length (3)
- BFS: $O(V + E)$ worst case
- Path reconstruction: $O(V)$ worst case

8 Conclusions

8.1 Project Achievements

1. Successful implementation of complete serverless architecture on AWS
2. Effective integration of Spring Boot with AWS Lambda
3. Modern frontend development with React and Vite
4. Resolution of complex technical issues during deployment
5. Comprehensive documentation of process and architectural decisions

8.2 Lessons Learned

- Version compatibility between API Gateway and Lambda handlers is critical
- SnapStart offers significant improvements in cold start times for Java
- CloudFront caching requires careful invalidation strategies
- SAM considerably simplifies serverless application deployment

8.3 Future Work

Possible project extensions:

1. **Database**: Migrate from in-memory graph to Amazon Neptune
2. **Authentication**: Implement AWS Cognito for user management
3. **Caching**: Add ElastiCache Redis for frequent results
4. **Monitoring**: Configure CloudWatch Dashboards and alarms
5. **CI/CD**: GitHub Actions for automated deployment
6. **Multi-language**: Support for Spanish and other languages
7. **Algorithms**: Implement Dijkstra, A* for different metrics

9 References

1. AWS Lambda Documentation: `https://docs.aws.amazon.com/lambda/`
2. Spring Boot AWS Serverless: `https://github.com/awslabs/aws-serverless-java-container`
3. AWS SAM Documentation: `https://docs.aws.amazon.com/serverless-application-model/`
4. React Documentation: `https://react.dev/`
5. Vite Documentation: `https://vitejs.dev/`

10 Appendices

10.1 Appendix A: SAM Template Configuration

Key fragment from `template.yaml`:

```yaml
Globals:
  Function:
    Timeout: 30
    MemorySize: 2048
    Runtime: java17
    Environment:
      Variables:
        JAVA_TOOL_OPTIONS: >
          -XX:+TieredCompilation
          -XX:TieredStopAtLevel=1
Resources:
  WordMapsFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: wordmaps-backend/
      Handler: com.wordmaps.config.LambdaHandler::handleRequest
      SnapStart:
        ApplyOn: PublishedVersions
      Events:
        ApiEvents:
          Type: HttpApi
          Properties:
            PayloadFormatVersion: "2.0"
            Path: /{proxy+}
            Method: ANY
```

10.2 Appendix B: Project URLs

- **Frontend**: https://d2euump6bzpha5.cloudfront.net
- **API Backend**: https://e23y9088lc.execute-api.us-east-1.amazonaws.com/api
- **GitHub Repository**: https://github.com/JuanLopezdeHierro/wordmaps_project