Grand Line Run

Serverless Cloud Native Gaming Application

Introduction

This project, Grand Line Run, is a multilevel flash game, based on the popular Japanese comic and Animated Series One Piece. The game consists of twelve distinct levels, each of which takes on the theme of a major story arc, progressing all the way from the first arc of the series to the arc that is currently being serialized.

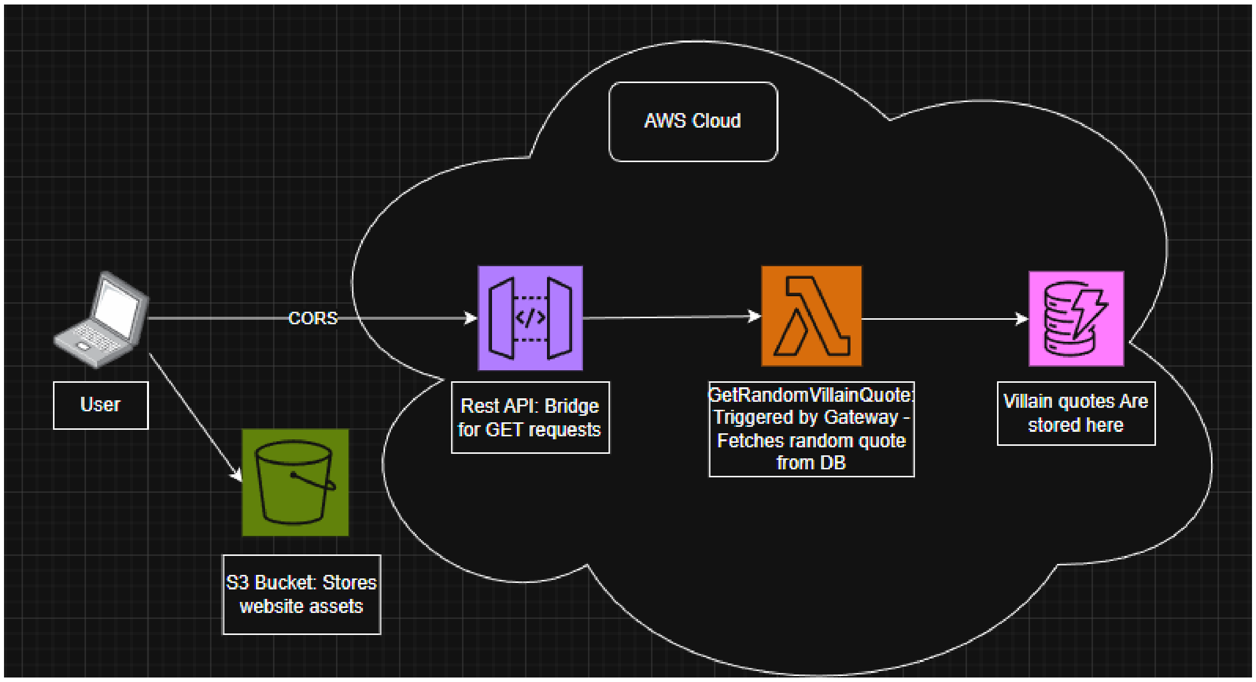
The main characters in One Piece are a pirate crew who sail the seas all in pursuit of their own individual dreams. Each member of the crew receives particular attention in different arcs, and as such in each level you play as a different member of the crew or as a crew ally, with the goal of capturing a delicious cut of meat which is bouncing all around the screen. At each level, the villain of that story arc is also bouncing around and contact with that villain immediately prompts a game over screen. At each game over, a sinister quote is randomly selected from a set of that villain's quotes and displayed to taunt the player, and the player is rewarded with something fitting the highest level that they cleared.

There are three possible quotes for each villain, at each death a random number between one and three is generated and the corresponding quote is fetched from a database to be displayed. The services I selected for Compute, Storage, Networking and Content Delivery, and Database respectively are listed in the following section.

Selected AWS Services

- **Compute:** AWS Lambda – The project's lambda function was used to access the villain quote database, fetching a quote corresponding to the level's villain. Given that the application is highly event-driven, lambdas per request scaling is the best fit. Financially, the choice of lambda makes the most sense for such a lightweight application since there are no idle costs, cost buildup only occurs at each invocation.
- **Storage:** Amazon Simple Storage Service (S3) – S3 was chosen primarily for its static web hosting, avoiding the operational overhead of services like EBS. Its integration with API Gateway for CORS also made it a no brainer since this trivialized access of the quote table.
- **Networking and Content Delivery:** Amazon API Gateway – API Gateway was chosen for its CORS, integration with S3 and its frontend HTTP request to lambda trigger translation.
- **Database:** Amazon DynamoDB – This was the best fit for the project's serverless architecture. The NoSQL key-based retrieval was the best fit for my simple database since I did not require any of the relational capabilities of RDS for example.

Architecture


AWS Cloud
CORS
User
Rest API: Bridge for GET requests
GetRandomVillainQuote: Triggered by Gateway - Fetches random quote from DB
Villain quotes Are stored here
S3 Bucket: Stores website assets

I will briefly demonstrate how the components integrate by describing the end-to-end request cycle initiated by the client. The client accesses the game via a standard HTTPS request, accessing the game's assets from S3. The client-side JavaScript integrates with the cloud backend, sending REST API requests to a Gateway endpoint. The API gateway automatically triggers the Lambda function upon the event of the client's defeat. The lambda function queries the DynamoDB database, retrieving one of the quotes that corresponds to the level at which the client was defeated. This quote is then displayed to the client on the UI.

Application Data

- S3 (one-piece-game-mh245882):
  - Index.html (940.0B): Basic structure and layout of the UI.
  - assets/ (25.34MB): Contains the game's visual assets; 13 JPG files and 26 PNG files.
  - js/ (18.171KB): Contains four JavaScript files which handle the game's logic, user interaction, and communication with the backend API.
- DynamoDB (VillainQuotes):
  - Partition key (VillainID (N)): There are 12 unique enemies numbered 1-12, each with a corresponding VillainID.
  - Sort key (QuoteID (N)): Each villain has three unique quotes, numbered by QuoteID, 1-3.

    - Quote (String): String corresponding to each valid VillainID/QuoteID combination, storing that VillainID's QuoteIDth quote. For example (6,2) corresponds to the sixth unique enemy's second quote, the string "How unlucky for you to meet me...".
- Lambda (***lambda_function.py/***GetRandomVillainQuote, 1.1KB): The lambda function is responsible for randomly selecting a quote from the current level's corresponding VillainID.

Programming Languages

The programming languages used were JavaScript, Python, and HTML. The entirety of the game's logic apart from the quote selection was written in JavaScript. I had initially programmed the game using Python's game development library Pygame, but since Pygame is not natively supported for internet play, I pivoted and did my best to translate all the game's logic to JavaScript. Python is used in the lambda function, which queries the DynamoDB table and fetches quotes. HTML was used to provide the basic UI structure. Code was required for the backend API logic, the database querying, and the client-side game mechanics which allowing the player to interact with the assets stored in S3.\

Application's Deployment

The presentation tier of my application consists of a static website that is hosted on S3, serving the game's assets and code to the player's browser. The application tier consists of the Restful API Gateway as an entry point that triggers the Lambda function (GetRandomVillainQuote). The Data tier consists of the DynamoDB database (VillainQuotes), storing three quotes for each level's enemy to be selected via the lambda function.

Application Security

- The Presentation tier(S3): The application's assets are automatically protected via SSE-S3 by default [5]. Since the S3 website endpoint is used to access the game, it is served over HTTP, making data in transit vulnerable since it is unencrypted. In the future, to protect the data in transit and enforce HTTPS I would use Amazon CloudFront [6].
- The Application tier (API Gateway and Lambda): I implemented a CORS policy that allows requests only from the S3 website URL, ensuring that only calls from the game to the API are allowed. Since the API endpoint is unauthenticated, however, it is susceptible to potential DoS attacks via high-volume requests from the S3 URL. In the future I might consider implementing a user sign-up with Amazon Cognito to ensure that requests only come from authenticated users [7].
- Data Tier (DynamoDB): Currently access to the table is managed via IAM roles, with the Lambda function using the default lab role to access the table. This is in violation of the principle of least privilege since the lab role's capabilities are much broader than just access of my VillainQuotes table. Unfortunately, I did not have the necessary credentials to create a custom IAM policy, but in the future, I would create a policy where only get requests to the specifically required table were allowed, enforcing the principle of least privilege.

Cost Analysis

Up-front costs for a serverless architecture are $0. To estimate the cost of the application I will estimate average monthly requests to be 10000.

- S3 (Costs $0.023/GB) [1]:
  - Our stored data = 25.36MB = 0.00248GB
  - The monthly cost comes out to 0.00248GB x $0.023/GB = $0.000570
- Lambda (Costs $0.20/1Million Requests) [2]:
  - The monthly cost comes out to 10000/1000000 x 0.20 = $0.002
- DynamoDB (Costs $0.125 per million read request units)[3]:
  - The monthly cost comes out to 10000/1000000 x 0.125 = $0.00125
- API Gateway (Costs $3.50 per million requests)[4]:
  - The monthly cost comes out to 10000/1000000 x 3.50 = $0.035

Combining the costs of each utilized AWS technology we have: $0.000570+$0.002+$0.00125+$0.035= $0.03882 per month.

Future Evolution and Continued Development

First and foremost, to enhance the application's security following the Application Security section of this report, I would like to use Amazon CloudFront to encrypt data in transit, Amazon Cognito to authenticate users and requests, and I would assign a security role to the lambda function allowing only GET requests to the VillainQuotes table.

The original plan for this project was to implement a sign-up feature where a user's high score would be stored in the cloud and displayed alongside their username on the game's title screen for all players to see. This also would have been done via Amazon DynamoDB. As previously mentioned, signup and sign in would be implemented with Amazon Cognito and user credentials would be stored in a Cognito User Pool. I would also randomize aspects of the game such as player and villain images, such that at each level, you could pull a random player image and villain image, adding a unique flare to each attempt. This could also be done using DynamoDB, storing the unique image URLs as attributes in table entries.

From a development standpoint, I would like to find a way to host the original Python version of the game since the physics and controls were much smoother than the JavaScript counterpart. I will likely be going in and changing all the assets to images that are free of any copyright as obviously I don't have any rights to the massive One Piece IP, but I did want to include this theme in the project and demo as I really love the series.

References

[1] “S3 pricing,” aws.amazon.com/s3/pricing, https://aws.amazon.com/s3/pricing/ (accessed Dec. 18, 2025).

[2] “Aws Lambda Pricing,” aws.amazon.com/lambda/pricing, https://aws.amazon.com/lambda/pricing/ (accessed Dec. 18, 2025).

[3] “Amazon dynamodb pricing | nosql key-value database | Amazon Web Services,” aws.amazon.com/dynamodb/pricing, https://aws.amazon.com/dynamodb/pricing/ (accessed Dec. 18, 2025).

[4] “Amazon API gateway pricing,” aws.amazon.com/api-gateway/pricing, https://aws.amazon.com/api-gateway/pricing/ (accessed Dec. 18, 2025).

[5] S. Stormacq, “Amazon S3 encrypts new objects by Default | AWS News Blog,” aws.amazon.com/blogs/aws/amazon-s3-encrypts-new-objects-by-default, https://aws.amazon.com/blogs/aws/amazon-s3-encrypts-new-objects-by-default/ (accessed Dec. 18, 2025).

[6] “Use HTTPS with CloudFront - amazon Cloudfront,” docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/using-https.html, https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/using-https.html (accessed Dec. 18, 2025).

[7] “Authentication service - customer iam (CIAM) - amazon cognito - AWS,” aws.amazon.com/cognito/features, https://aws.amazon.com/cognito/ (accessed Dec. 18, 2025).