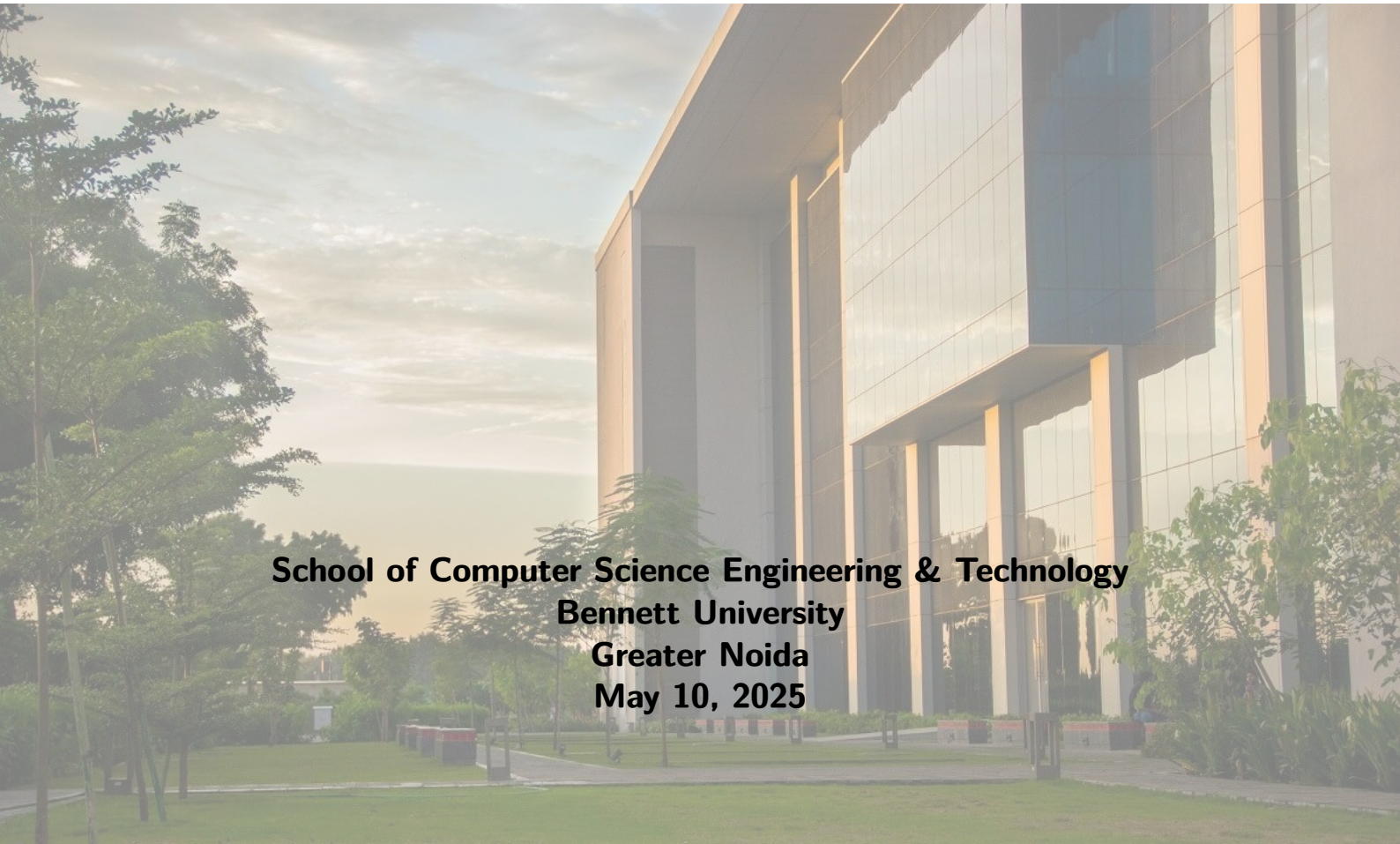
GitshowcaseAPI

CSET463:AWS Cloud Support Associate

Kartikey Pandey	E22CSEU0940
Ritika Sharma	E22CSEU0902

Mentor: Dr. Naween Kumar



School of Computer Science Engineering & Technology
Bennett University
Greater Noida

May 10, 2025

Contents

1 | Executive Summary

The GitShowcaseAPI is an advanced, serverless web application meticulously engineered and deployed on Amazon Web Services (AWS), designed to empower developers by providing a seamless platform to showcase their GitHub profiles. Hosted globally at https://d3tbtv7bxs3vbw.cloudfront.net, this application allows users to register a GitHub username through a highly responsive and user-friendly web interface, subsequently presenting detailed profile metrics that include top repositories sorted by stargazers, total stars across all repositories, total commits made, counts of public repositories, and timestamps of recent activities. The application's frontend is statically hosted on an AWS S3 bucket named `gitshowcase-frontend`, with global distribution ensured through AWS CloudFront, which leverages edge locations to provide low-latency access to users worldwide. On the backend, the system is powered by a suite of AWS services, including AWS Lambda for serverless compute, API Gateway for managing RESTful endpoints, and DynamoDB for scalable NoSQL data storage. Automation is a key feature, with AWS EventBridge facilitating hourly data refreshes to keep profile information current, and GitHub Actions enabling continuous integration and deployment (CI/CD) for streamlined updates. Security is rigorously enforced, with sensitive credentials such as the GitHub API token securely stored in AWS Secrets Manager and access tightly controlled through AWS Identity and Access Management (IAM) roles, ensuring that only authorized components can interact with critical resources.

This project was developed as a capstone for the **CSET463: AWS Cloud Support Associate** course at Bennett University, under the expert guidance of Dr. Naween Kumar. The GitShowcaseAPI addresses a critical need in the technology sector for efficient, automated tools to visualize GitHub profiles, serving as a valuable asset for developers seeking to highlight their work, recruiters evaluating potential candidates, and portfolio enthusiasts curating professional showcases within the tech industry. By adopting a serverless architecture, the application eliminates the complexities of traditional server management, significantly reduces operational costs, and supports dynamic scaling to accommodate varying user demands seamlessly. Given the possibility that the application may be decommissioned in the future, this report is crafted as a comprehensive archival document, meticulously detailing every aspect of the project across 12 carefully structured sections. These sections cover the design, implementation, configuration, operation, testing, and future potential of GitShowcaseAPI. To enhance clarity and provide tangible evidence of the system's setup, the report includes a suite of visual aids: an architecture diagram (Figure 5.1) that maps out the system's infrastructure, and 12 detailed screenshots capturing key components (Figures 9.1 to 9.12). Together, these elements make the report not only a professional project deliverable but also a definitive record of the application's functionality, ensuring its legacy for future developers, researchers, and educators.

2 | Introduction

2.1 | Project Context and Motivation

The GitShowcaseAPI is a cloud-native, serverless web application meticulously crafted to simplify and enhance the presentation of GitHub developer profiles, addressing a pressing and well-documented need in the technology sector for accessible, automated, and user-friendly portfolio visualization tools. GitHub, recognized globally as the leading platform for code collaboration and version control, hosts millions of developer profiles, each rich with extensive data on repositories, contributions, and activity timelines. However, the process of extracting this data and presenting it in a concise, visually appealing, and user-friendly format poses a significant challenge, often requiring substantial manual effort and technical expertise. Developers urgently require efficient tools to showcase their work to potential employers, collaborators, or open-source communities, enabling them to highlight their skills, projects, and contributions effectively. Similarly, recruiters and project managers need accessible, consolidated summaries to evaluate candidates or contributors swiftly, without navigating the often overwhelming volume of raw data on GitHub. GitShowcaseAPI directly addresses these pain points by automating the entire process: users can register a GitHub username through an intuitive web interface and access a comprehensive profile summary that includes top repositories sorted by stars, total stars, total commits, counts of public repositories, and recent activity, all presented in a responsive, visually appealing format that caters to diverse user needs.

This project was undertaken by Kartikey Pandey (E22CSEU0940) and Ritika Sharma (E22CSEU0902) as a capstone initiative for the **CSET463: AWS Cloud Support Associate** course at Bennett University. Under the mentorship of Dr. Naween Kumar, the development of GitShowcaseAPI aligns seamlessly with the course's objectives, which emphasize fostering expertise in designing, deploying, and

managing cloud-based applications using modern cloud computing principles. The project serves as a practical demonstration of serverless architectures, AWS service integration, and the application of cloud computing best practices to solve real-world problems. By leveraging a serverless design, GitShowcaseAPI eliminates the need for traditional server infrastructure, thereby reducing operational overhead, minimizing maintenance efforts, and supporting dynamic scaling to handle varying user loads efficiently. This makes the application both cost-efficient and highly scalable, capable of serving a global audience without the burden of managing physical servers. Recognizing the potential for the application to be decommissioned after the course, this report prioritizes exhaustive documentation to preserve its design, implementation, and functionality for future reference, serving as a definitive archive for developers, researchers, or instructors who may wish to study or replicate the system.

2.2 | Technical Overview

GitShowcaseAPI is built upon a robust serverless architecture, integrating a carefully selected suite of AWS services with the GitHub API to deliver a seamless, scalable, and fully automated user experience. The frontend is constructed using HTML for structural integrity, CSS for responsive and visually appealing styling, and JavaScript for dynamic, interactive features. It is hosted on an AWS S3 bucket named `gitshowcase-frontend`, configured for static website hosting, and distributed globally through AWS CloudFront, a content delivery network (CDN) that ensures low-latency access by caching content at edge locations worldwide. This setup guarantees that users, regardless of their geographic location, experience fast and reliable access to the application. On the backend, the application relies on four AWS Lambda functions, each designed to handle specific tasks:

- `GitShowcase_Register`: Processes user registrations by fetching GitHub profile data using the GitHub API and storing it in DynamoDB for persistent access.
- `GitShowcase_Showcase`: Retrieves stored profile data from DynamoDB and serves it to the frontend for display, enabling users to view their GitHub metrics.
- `RefreshGitHubData`: Runs hourly to update profile data, ensuring that the information displayed remains current by re-fetching data from the GitHub API.
- `GitShowcaseDeployLambda`: Automates frontend deployments through CI/CD pipelines, ensuring seamless updates to the application's user interface.

These Lambda functions are orchestrated through AWS API Gateway, which exposes two RESTful endpoints: `/register` for user registration and `/showcase` for profile retrieval. The backend interacts with a DynamoDB table named `GitShowcase`, which provides scalable, high-performance storage for GitHub profile data, with `username` as the primary key to ensure efficient lookups. Automation is a cornerstone of the system: AWS EventBridge schedules hourly triggers for the `RefreshGitHubData` function, ensuring that profile data remains fresh without requiring manual intervention. Meanwhile, GitHub Actions facilitates CI/CD by invoking the `GitShowcaseDeployLambda` function whenever code changes are pushed to the repository's `main` branch, streamlining the deployment of frontend updates. Security is meticulously enforced through AWS Secrets Manager, which securely stores the GitHub API token, and IAM roles that implement least-privilege access, ensuring that each component has only the permissions necessary to perform its tasks.
The system's architecture is visually represented in the architecture diagram (Figure 5.1), which provides a comprehensive overview of the interactions among AWS services, GitHub integrations, and the end-user. This diagram, along with a series of screenshots capturing key components (Figures 9.1 to 9.12), offers a tangible and detailed view of the application's infrastructure and configuration, which are elaborated upon in the subsequent sections of this report.

2.3 | Significance and Documentation Goals

GitShowcaseAPI stands as a fully functional, user-centric application that empowers developers to showcase their GitHub profiles with minimal effort, making it an invaluable tool for professional networking, recruitment, and portfolio presentation in the technology ecosystem. Its serverless architecture exemplifies best practices in cloud computing, showcasing scalability through automatic resource adjustment, cost-efficiency by eliminating server maintenance costs, and automation through scheduled data updates and CI/CD pipelines. However, recognizing that the application may not remain deployed indefinitely

due to course timelines or resource constraints, this report is designed to serve as a comprehensive archive, capturing every facet of the project's lifecycle—from design and implementation to configuration, operation, and testing. By including detailed narratives, visual aids, and endpoint documentation, the report ensures that future developers, researchers, or instructors can fully understand the system's intricacies and potentially recreate it if desired. The inclusion of 12 screenshots provides a tangible record of the system's configuration at the time of deployment, while the narrative explains the rationale behind critical design decisions, such as the choice of serverless architecture, the selection of AWS services, and the implementation of security measures. This dual approach makes the report both a technical archive for posterity and a professional deliverable for the **CSET463** course, reflecting the project's academic and practical significance.

3 | Objectives

The GitShowcaseAPI project was developed with a clear set of objectives that align with both the technical requirements of a modern cloud-based application and the academic goals of the **CSET463** course at Bennett University. These objectives were carefully defined to ensure that the project delivers tangible value to its users while providing a robust learning experience in cloud computing and serverless architectures. The objectives are as follows:

- **User Interface Development**: Design and implement a responsive web interface that enables users to register GitHub usernames and view detailed profile metrics in a visually appealing and accessible format. The interface should display key metrics such as top repositories sorted by stars, total stars across all repositories, total commits, public repository counts, and timestamps of recent activity, ensuring that the presentation is both intuitive and informative for diverse audiences, including developers and recruiters.
- **Serverless Backend Implementation**: Develop a serverless backend using AWS Lambda and API Gateway to process RESTful requests efficiently, fetching GitHub profile data via the GitHub API and serving it through secure, scalable endpoints. The backend should handle user registrations and profile retrievals with minimal latency, leveraging the benefits of serverless computing to eliminate server management overhead.
- **Data Management**: Utilize AWS DynamoDB to store and manage GitHub profile data, ensuring scalability to handle large datasets, reliability to prevent data loss, and high-performance query capabilities to support rapid data retrieval. The database should be structured to efficiently store complex profile metrics, including nested repository data, while maintaining cost-effectiveness through optimized throughput settings.
- **Automation**: Implement automated hourly data refreshes using AWS EventBridge to maintain up-to-date profile information without requiring manual intervention, thereby enhancing the user experience by ensuring that displayed data reflects the latest GitHub activity. This automation should be seamless, reliable, and capable of handling updates for multiple users concurrently.
- **CI/CD Pipeline**: Establish a robust CI/CD pipeline using GitHub Actions to automate frontend deployments and CloudFront cache invalidation, streamlining the process of updating the application and ensuring that users always access the latest version of the frontend. This pipeline should integrate seamlessly with AWS services, providing a smooth deployment experience for developers.
- **Security**: Secure sensitive credentials, such as the GitHub API token, using AWS Secrets Manager, and enforce strict access controls through IAM roles to protect the system's integrity. Security measures should include encryption of data at rest and in transit, input validation to prevent attacks, and logging practices that avoid exposing sensitive information, ensuring a robust defense against potential threats.
- **Global Accessibility**: Ensure global accessibility and minimal latency by hosting the frontend on AWS S3 and distributing it via CloudFront's content delivery network, leveraging edge locations to deliver content efficiently to users worldwide. The application should provide a consistent and fast user experience regardless of the user's geographic location.

- **Testing and Validation**: Conduct comprehensive testing and validation of all components—frontend, backend, data storage, automation, and security—to guarantee functionality, reliability, and performance under various conditions. Testing should include unit tests, integration tests, performance tests, and security audits to ensure that the application meets its objectives and performs reliably in production.

These objectives collectively guided the project's development, ensuring a balanced focus on technical innovation, user-centric design, and academic rigor, resulting in a system that is both practical for real-world use and educational for understanding cloud computing principles.

4 | Technologies Used

GitShowcaseAPI leverages an extensive and sophisticated stack of cloud and web technologies, each carefully selected to deliver its functionality while adhering to principles of scalability, reliability, and serverless computing. The following technologies form the backbone of the application, with detailed descriptions of their roles and configurations:

- **AWS S3**: Serves as the hosting platform for the application's static frontend assets, which include HTML files for structure, CSS files for responsive styling, and JavaScript files for dynamic interactions. The S3 bucket, named `gitshowcase-frontend`, is configured for static website hosting, with public read access enabled to allow users to access the frontend. Server-side encryption (SSE-S3) is enabled to protect data at rest, and versioning is activated to maintain a history of changes, ensuring data integrity and facilitating rollback if needed.
- **AWS CloudFront**: Acts as a content delivery network (CDN) to distribute the frontend content globally, ensuring low-latency access for users worldwide. The CloudFront distribution is accessible at https://d3tbtv7bxs3vbw.cloudfront.net, with HTTPS enforcement to secure data in transit using TLS 1.2 or higher. A 24-hour cache policy (time-to-live, TTL) is implemented to optimize performance by caching static assets at edge locations, reducing load on the origin S3 bucket while ensuring that updates are reflected after cache invalidation.
- **AWS API Gateway**: Manages the application's RESTful endpoints, providing a secure and scalable interface between the frontend and backend services. Two endpoints are defined: `/register` (POST) for user registration and `/showcase` (GET) for profile retrieval. The API is deployed to a `dev` stage in the `us-east-1` region, with Cross-Origin Resource Sharing (CORS) enabled to allow the frontend to make cross-domain requests, ensuring seamless communication between the client and server components.
- **AWS Lambda**: Powers the backend with four serverless functions, each designed to handle specific tasks within the application. These functions are written in Python 3.9, configured with 128 MB of memory, and set to a 30-second timeout to ensure efficient execution:
  - `GitShowcase_Register`: Handles user registration by processing POST requests to the `/register` endpoint, fetching GitHub profile data, and storing it in DynamoDB.
  - `GitShowcase_Showcase`: Manages GET requests to the `/showcase` endpoint, retrieving profile data from DynamoDB and returning it as a JSON response for frontend rendering.
  - `RefreshGitHubData`: Executes hourly to update stored profile data, ensuring that the information remains current by re-fetching data from the GitHub API.
  - `GitShowcaseDeployLambda`: Automates frontend deployments by updating the S3 bucket with new code and invalidating CloudFront's cache, ensuring that users access the latest version of the frontend.
- **AWS DynamoDB**: Provides scalable, high-performance storage for GitHub profile data in a NoSQL table named `GitShowcase`. The table uses `username` as the primary key to uniquely identify each profile, with attributes such as `name`, `avatar_url`, `bio`, `total_repos`, `total_stars`, `total_commits`, `last_seen`, and `top_repos` to store comprehensive profile metrics. It is provisioned with 5 read and 5 write capacity units, with auto-scaling enabled to adapt to usage spikes, ensuring cost-efficiency while maintaining performance. Point-in-time recovery and encryption at rest are enabled to protect data against loss and unauthorized access.

- **AWS EventBridge**: Facilitates automation by scheduling hourly triggers for the `RefreshGitHubData` function. A rule named `HourlyDataRefresh` is configured with a cron expression (`0 * * * ? *`) to execute the function every hour, ensuring that profile data remains fresh without manual intervention, thereby enhancing the user experience by providing up-to-date information.
- **AWS Secrets Manager**: Securely stores the GitHub API token under the identifier `github-api-token`, ensuring that sensitive credentials are protected with encryption at rest and in transit. Access to the token is restricted to the `GitShowcase_Register` and `RefreshGitHubData` Lambda functions through IAM roles, minimizing the risk of unauthorized access and ensuring compliance with security best practices.
- **GitHub API**: Provides the core data source for the application, enabling the fetching of user and repository data via HTTPS requests. The API is accessed using an authenticated token to retrieve detailed metrics such as repository stars, commits, and activity timestamps, which are then processed and stored in DynamoDB for display.
- **GitHub Actions**: Automates the CI/CD pipeline by triggering the `GitShowcaseDeployLambda` function on pushes to the `main` branch of the GitHub repository. This ensures that frontend updates are deployed seamlessly, with the workflow handling authentication with AWS, code checkout, and Lambda invocation, streamlining the deployment process.
- **Frontend Technologies**: Utilizes a combination of HTML, CSS, and JavaScript to create a polished and responsive user interface. HTML provides the structural foundation for the registration form and profile display page, CSS ensures responsive styling with media queries for mobile compatibility, and JavaScript enables dynamic interactions such as form submission, data fetching, and rendering of profile metrics, delivering a seamless user experience across devices.

This technology stack collectively ensures that GitShowcaseAPI is robust, scalable, and maintainable, with each component carefully configured to support the application's objectives, as evidenced by the detailed screenshots provided throughout this report.

5 | System Architecture

GitShowcaseAPI is architected as a fully serverless system, leveraging a combination of AWS services and GitHub integrations to deliver a scalable, reliable, and automated platform for showcasing GitHub profiles. The architecture is designed to minimize operational overhead while maximizing performance, security, and user experience. The system comprises several interconnected components, each with a specific role in the overall workflow:

- **Frontend**: The user-facing component of the application, implemented as a static website hosted on an AWS S3 bucket named `gitshowcase-frontend`. The frontend includes a registration form for users to submit their GitHub usernames and a profile display page that presents fetched metrics such as top repositories, total stars, and commits in a structured, responsive layout. AWS CloudFront distributes these static assets globally, caching content at edge locations to ensure low-latency access for users worldwide.
- **Backend**: The backend is powered by two primary AWS Lambda functions: `GitShowcase_Register` and `GitShowcase_Showcase`. These functions handle requests sent to API Gateway endpoints (`/register` for POST requests and `/showcase` for GET requests). The `GitShowcase_Register` function fetches data from the GitHub API and stores it in DynamoDB, while the `GitShowcase_Showcase` function retrieves stored data for display on the frontend, ensuring efficient processing of user requests.
- **Data Refresh**: To keep profile data current, the `RefreshGitHubData` Lambda function is triggered hourly by an AWS EventBridge rule named `HourlyDataRefresh`. This function re-fetches data from the GitHub API for all stored usernames, updates the corresponding records in DynamoDB, and ensures that the displayed information reflects the latest GitHub activity without requiring manual intervention.
- **CI/CD**: The `GitShowcaseDeployLambda` function, invoked by a GitHub Actions workflow, automates frontend deployments. On every push to the `main` branch of the GitHub repository, this

function updates the S3 bucket with the latest frontend code (HTML, CSS, JavaScript) and invalidates CloudFront's cache, ensuring that users immediately access the updated version of the application without delays caused by caching.

- **Security**: Security is a priority, with AWS Secrets Manager used to store the GitHub API token securely. The token is accessed only by the `GitShowcase_Register` and `RefreshGitHubData` functions through IAM roles that enforce least-privilege access, minimizing the risk of unauthorized access. Additional security measures, such as HTTPS enforcement and input validation, are implemented to protect the system and its users.

The architecture diagram (Figure 5.1) provides a visual representation of these components and their interactions, illustrating the flow of data from user input to profile display, as well as the automated processes that maintain system integrity, such as data refreshes and deployments. Detailed configurations of each component are illustrated in subsequent sections, with screenshots providing tangible evidence of the setup.

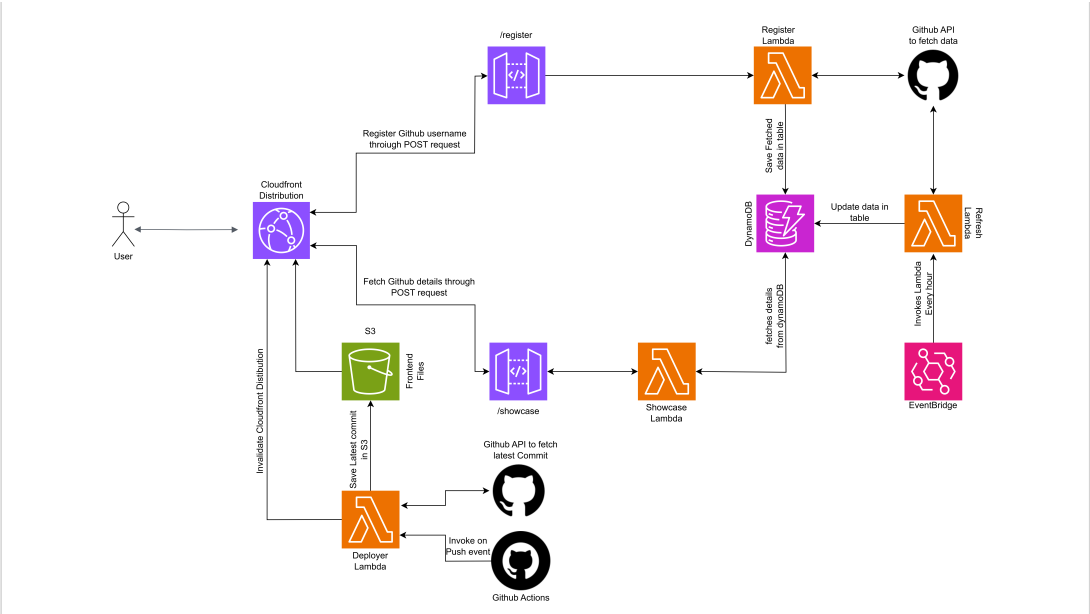


Figure 5.1: Architecture Diagram of GitShowcaseAPI, depicting the serverless infrastructure with AWS services (S3, CloudFront, Lambda, API Gateway, DynamoDB, EventBridge, Secrets Manager) and GitHub integrations (GitHub API, GitHub Actions). The diagram illustrates the flow of data from user input through frontend interactions to backend processing, storage, and automated updates, highlighting the system's scalability, automation, and global accessibility.

6 | Implementation Details

This section provides an exhaustive account of the implementation of GitShowcaseAPI's core components, delving into their functionality, design decisions, and integration within the serverless architecture. Each component is described in detail to provide a comprehensive understanding of how the system was built and how it operates.

6.1 | Frontend Implementation

The frontend of GitShowcaseAPI is implemented as a static website, developed using a combination of HTML, CSS, and JavaScript to deliver a responsive and user-friendly interface. It is hosted on an AWS S3 bucket named `gitshowcase-frontend`, which is configured for static website hosting with public read access to allow users to access the application. The frontend comprises two primary interfaces designed to facilitate user interaction:

- **Registration Form**: This interface provides a user-friendly form where users can input their GitHub username and submit it for processing. The form is styled with CSS to ensure clarity,

accessibility, and visual appeal, with features such as input validation and error messaging to guide users. JavaScript handles the form submission process, making a POST request to the `/register` endpoint and displaying the response (e.g., success or error messages) to the user, ensuring a smooth and interactive experience.

- **Profile Display Page**: This dynamic page fetches profile data from the `/showcase` endpoint via a GET request and presents the retrieved metrics in a structured, visually appealing layout. Metrics displayed include the user's avatar, bio, total repositories, total stars, total commits, and a list of top repositories with details such as stars, forks, and commit counts. CSS Grid and media queries are employed to ensure responsiveness, providing a consistent user experience across a wide range of devices, from smartphones with small screens to desktops with large monitors.

AWS CloudFront enhances the frontend's performance by caching assets for 24 hours at edge locations worldwide, ensuring low-latency access for users regardless of their geographic location. The frontend's design prioritizes simplicity and maintainability, with clean code separation between HTML (structure), CSS (styling), and JavaScript (functionality), making it easy to update and extend. Its integration with the backend via API Gateway ensures seamless data retrieval, allowing users to interact with their GitHub profile data effortlessly.

6.2 | Backend Implementation

The backend of GitShowcaseAPI is powered by two AWS Lambda functions, integrated with API Gateway to handle user requests efficiently and securely. These functions are designed to process RESTful requests, interact with external APIs and databases, and return responses to the frontend. The details of each function are as follows:

- **GitShowcase_Register**: This function processes POST requests to the `/register` endpoint, accessible at https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/register. It performs several key tasks: first, it validates the submitted GitHub username to ensure it is well-formed and exists; then, it retrieves the GitHub API token from AWS Secrets Manager to authenticate requests; next, it fetches user and repository data from the GitHub API, processing the data to extract key metrics such as total stars, total commits, and top repositories sorted by stars; finally, it stores the processed profile data in the `GitShowcase` DynamoDB table for persistent storage. The function is configured with Python 3.9 runtime, 128 MB of memory, and a 30-second timeout, ensuring efficient execution even under load.

- **GitShowcase_Showcase**: This function handles GET requests to the `/showcase` endpoint, accessible at https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/showcase. It accepts a `username` query parameter (e.g., `?username=xitikx`), queries the `GitShowcase` DynamoDB table for the corresponding profile data, and returns the data as a JSON response. The frontend then renders this data into a structured display for the user. Like the `GitShowcase_Register` function, it uses Python 3.9 runtime, 128 MB of memory, and a 30-second timeout, with IAM permissions limited to DynamoDB read operations to enforce least-privilege access.

AWS API Gateway serves as the entry point for these functions, routing incoming requests to the appropriate Lambda function, handling response formatting, and enabling CORS to allow cross-domain requests from the frontend. The API Gateway setup includes detailed logging and monitoring to track request performance and errors, ensuring that the backend operates reliably in production.

6.3 | Data Storage

AWS DynamoDB serves as the persistent storage layer for GitShowcaseAPI, hosting all GitHub profile data in a NoSQL table named `GitShowcase`. The table is designed with the following characteristics to ensure scalability, performance, and data integrity:

- **Primary Key**: The `username` attribute (string type) serves as the primary key, ensuring that each profile is uniquely identified and can be efficiently retrieved using a single query. This design choice optimizes lookup performance, which is critical for the `/showcase` endpoint's responsiveness.

- **Attributes**: The table stores a comprehensive set of attributes for each profile, including `name` (user's full name), `avatar_url` (URL to the user's GitHub avatar), `bio` (user's GitHub bio, if available), `total_repos` (count of public repositories), `total_stars` (total stars across all repositories),

`total_commits` (total commits across repositories), `last_seen` (timestamp of the user's most recent activity), and `top_repos` (a list of repository objects, each containing `name`, `stars`, `forks`, `commits`, and `url`). This schema allows the application to store and display rich profile data in a structured format.

- **Throughput**: The table is provisioned with 5 read capacity units (RCUs) and 5 write capacity units (WCUs), with auto-scaling enabled to dynamically adjust capacity based on usage spikes. This configuration ensures that the application remains cost-efficient during periods of low activity while scaling seamlessly to handle high demand, such as during bulk registrations or data refreshes.
- **Data Protection**: Point-in-time recovery is enabled to safeguard against data loss, allowing the table to be restored to any point within a 35-day window. Encryption at rest, using AWS-managed keys, is also enabled to ensure compliance with security best practices, protecting sensitive user data from unauthorized access.

The table's schema supports efficient queries by username, enabling rapid data retrieval for the `/showcase` endpoint. A sample item for the user `xitikx` illustrates the data structure, showing how metrics are stored and retrieved for display.

6.4 | Data Refresh Mechanism

To ensure that profile data remains current, GitShowcaseAPI implements an automated data refresh mechanism using the `RefreshGitHubData` Lambda function, which is triggered hourly by an AWS EventBridge rule named `HourlyDataRefresh`. This function performs the following tasks to keep the data up-to-date:

- Queries the `GitShowcase` DynamoDB table to retrieve a list of all stored usernames, ensuring that every registered user's profile is updated during each cycle.
- Retrieves the GitHub API token from AWS Secrets Manager to authenticate requests, ensuring secure access to the GitHub API without hardcoding sensitive credentials in the function code.
- Fetches updated user and repository data for each username from the GitHub API, retrieving the latest metrics such as repository stars, commits, and activity timestamps to reflect any changes since the last update.
- Processes the fetched data to compute updated metrics (e.g., total stars, top repositories) and updates the corresponding DynamoDB items with the latest information, ensuring that the stored data accurately represents the user's current GitHub profile.

This automation eliminates the need for manual updates, ensuring that displayed profiles always reflect the most recent GitHub activity. The EventBridge rule uses a cron schedule (`0 * * * ? *`) to trigger the function every hour, providing a consistent and reliable update cycle that enhances the user experience by delivering fresh data.

6.5 | CI/CD Pipeline

The `GitShowcaseDeployLambda` function forms the core of GitShowcaseAPI's CI/CD pipeline, automating frontend deployments to ensure that updates to the application's codebase are seamlessly reflected in production. This function is triggered by a GitHub Actions workflow and performs the following steps:

- Retrieves the latest frontend code (HTML, CSS, JavaScript files) from the GitHub repository's `main` branch, ensuring that the most recent changes are deployed.
- Uploads the updated files to the `gitshowcase-frontend` S3 bucket, overwriting existing assets to reflect the new version of the frontend. This process ensures that the bucket always contains the latest code, ready for distribution via CloudFront.
- Invalidates the CloudFront cache to ensure that users access the updated version of the frontend immediately, bypassing the 24-hour cache TTL that would otherwise delay the visibility of changes.

The GitHub Actions workflow is configured to run on every push to the `main` branch, using repository secrets (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) to authenticate with AWS. The workflow checks out the repository, invokes the `GitShowcaseDeployLambda` function, and monitors the deployment for success, ensuring a streamlined and automated deployment process that minimizes downtime and manual effort.

7 | API Endpoints

GitShowcaseAPI exposes two RESTful endpoints via AWS API Gateway, enabling seamless interaction between the frontend and backend components. This section provides detailed documentation of the `/register` and `/showcase` endpoints, covering their URLs, HTTP methods, content types, input and output formats, functionality, and error handling to ensure a thorough understanding of their operation.

7.1 | Register Endpoint

The `/register` endpoint allows users to submit a GitHub username for processing and storage, initiating the profile creation process. Its details are as follows:

- **URL**: https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/register
- **Method**: POST
- **Content Type**: `application/json`
- **Input Body**: A JSON object containing the GitHub username, e.g., {`"username": "xitikx"`}. The username must be a valid GitHub handle, as the endpoint validates its existence before proceeding.
- **Output**: A JSON response confirming successful registration or indicating an error, e.g., {`"message": "User registered successfully"`} for success, or {`"error": "Invalid GitHub username"`} for failure.
- **Functionality**: The endpoint triggers the `GitShowcase_Register` Lambda function, which performs a series of steps: it validates the username by checking its existence on GitHub; retrieves the GitHub API token from Secrets Manager for authentication; fetches user and repository data from the GitHub API, including metrics like total stars, commits, and top repositories; processes the data to extract relevant metrics; and stores the resulting profile in the `GitShowcase` DynamoDB table, making it available for retrieval via the `/showcase` endpoint.

Field	Value
Request URL	https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/register
Request Method	POST
Content Type	`application/json`
Request Body	{`"username": "xitikx"`}
Response Status	200 OK
Response Body	{`"message": "User registered successfully"`}

Table 7.1: Sample request and response for the `/register` endpoint, demonstrating a successful registration for the user ‘xitikx‘. The request submits the username in JSON format, and the response confirms that the profile data has been processed and stored in DynamoDB, ready for retrieval.

7.2 | Showcase Endpoint

The `/showcase` endpoint retrieves and returns a user’s stored profile data for display on the frontend, enabling users to view their GitHub metrics in a structured format. Its details are as follows:

- **URL**: https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/showcase
- **Method**: GET

- **Query Parameter**: `username`, e.g., `?username=xitikx`, which specifies the GitHub username whose profile data is to be retrieved.
- **Output**: A JSON response containing the user's profile data, including metrics such as total stars, total commits, and a list of top repositories, or an error message if the username is not found, e.g., {`"error": "Username not found"`}.
- **Functionality**: The endpoint invokes the `GitShowcase_Showcase` Lambda function, which queries the `GitShowcase` DynamoDB table using the provided username as the primary key. If a matching record exists, the function returns the profile data as a JSON object, which the frontend renders into a structured layout displaying the user's avatar, bio, repository counts, and other metrics.

Field	Value
Request URL	https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/showcase?username=xitikx
Request Method	GET
Response Status	200 OK
Response Body	{"total_stars": 11, "bio": null, "total_repos": 20, "avatar_url": "https://avatars.githubusercontent.com/u/143159504?v=4", "last_seen": "2025-05-02T07:59:52Z", "username": "xitikx", "top_repos": [{"name": "AYNA", "forks": 0, "commits": 22, "stars": 2, "url": "https://github.com/xitikx/AYNA"}, {"name": "Confessions", "forks": 0, "commits": 21, "stars": 2, "url": "https://github.com/xitikx/Confessions"}, {"name": "Edge-Networks-Research-Paper", "forks": 0, "commits": 17, "stars": 2, "url": "https://github.com/xitikx/Edge-Networks-Research-Paper"}, {"name": "KidWise", "forks": 0, "commits": 61, "stars": 2, "url": "https://github.com/xitikx/KidWise"}, {"name": "performanceAnalysis", "forks": 0, "commits": 3, "stars": 2, "url": "https://github.com/xitikx/performanceAnalysis"}, {"name": "portfolio", "forks": 0, "commits": 8, "stars": 1, "url": "https://github.com/xitikx/portfolio"}, {"name": "Calculator", "forks": 0, "commits": 6, "stars": 0, "url": "https://github.com/xitikx/Calculator"}], "total_commits": 138, "name": "Ritika Sharma"}

Table 7.2: Sample request and response for the `/showcase` endpoint, retrieving the profile data for the user 'xitikx'. The response includes comprehensive metrics, such as total stars, total commits, and a list of top repositories with their details, which the frontend renders for user visualization.

7.3 | Error Handling

Both endpoints implement robust error handling mechanisms to ensure reliability and a positive user experience, addressing various failure scenarios gracefully:

- **Invalid Username**: If the provided username does not exist on GitHub, the `/register` endpoint returns a 400 Bad Request response with a message such as {`"error": "Invalid GitHub username"`}, informing the user of the issue and preventing further processing of an invalid request.
- **Missing Query Parameter**: If the `/showcase` endpoint is called without the required `username` query parameter, it returns a 400 Bad Request response with a message like {`"error": "Username parameter is required"`}, ensuring that the frontend provides all necessary information for a successful request.

- **Server Errors**: If DynamoDB or the GitHub API becomes unavailable due to connectivity issues or service outages, both endpoints return a 500 Internal Server Error with a generic message such as {`"error": "Internal server error"`}. This approach avoids exposing sensitive internal details while informing the user that the issue is server-side, allowing them to retry the request later.
- **API Rate Limits**: The GitHub API imposes a rate limit of 5,000 requests per hour for authenticated users. If this limit is exceeded, the `/register` endpoint gracefully degrades by queuing the request for processing during the next hourly refresh cycle by the `RefreshGitHubData` function, ensuring that the application remains operational even under rate-limiting constraints.

These error handling strategies ensure that the application remains robust and user-friendly under various failure conditions, maintaining user trust and system integrity by providing clear feedback and preventing crashes.

8 | Configuration and Setup

This section provides a detailed guide to the configuration and setup of all AWS services and GitHub Actions required to deploy and operate GitShowcaseAPI, ensuring reproducibility and clarity for future reference. Each component is described with its specific settings, permissions, and deployment steps.

8.1 | AWS Services Configuration

Each AWS service used in GitShowcaseAPI was meticulously configured to meet the application's requirements, with settings optimized for performance, security, and cost-efficiency:

- **S3**: The `gitshowcase-frontend` bucket was created in the `us-east-1` region, configured for static website hosting to serve the frontend. The bucket policy grants public read access to HTML, CSS, and JavaScript files, with `index.html` set as the default document to ensure proper routing. Versioning is enabled to maintain a history of changes, and server-side encryption (SSE-S3) is activated to protect data at rest, ensuring compliance with security best practices.
- **CloudFront**: A CloudFront distribution was configured with the `gitshowcase-frontend` S3 bucket as its origin, enforcing HTTPS-only access to secure data in transit using TLS 1.2 or higher. The cache policy is set to a 24-hour TTL to optimize performance by caching static assets at edge locations, with `index.html` as the default root object. The distribution's domain, https://d3tbtv7bxs3vbw.cloudfront.net, ensures global accessibility with minimal latency for users worldwide.
- **API Gateway**: A REST API named `GitShowcaseAPI` was created in the `us-east-1` region, defining two resources: `/register` (POST) for user registration and `/showcase` (GET) for profile retrieval. Each resource is integrated with its respective Lambda function (`GitShowcase_Register` and `GitShowcase_Showcase`), and the API is deployed to a `dev` stage with CORS enabled to support cross-domain requests from the frontend. Logging and monitoring are enabled to track request performance and errors.
- **Lambda**: Four Lambda functions were configured with Python 3.9 runtime, 128 MB memory, and a 30-second timeout to ensure efficient execution:
  - `GitShowcase_Register`: Granted IAM permissions for DynamoDB writes, Secrets Manager reads, and GitHub API access, allowing it to fetch and store profile data securely.
  - `GitShowcase_Showcase`: Granted permissions for DynamoDB reads only, ensuring least-privilege access for retrieving profile data.
  - `RefreshGitHubData`: Granted permissions for DynamoDB reads and writes, Secrets Manager reads, and GitHub API access, enabling it to update profile data hourly.
  - `GitShowcaseDeployLambda`: Granted permissions for S3 writes and CloudFront cache invalidation, allowing it to update the frontend and ensure immediate visibility of changes.
- **DynamoDB**: The `GitShowcase` table was created with `username` as the partition key, provisioned with 5 RCUs and 5 WCUs, and auto-scaling enabled to handle variable loads efficiently. Point-in-time recovery and encryption at rest are activated to protect data, ensuring that the table can be restored in case of accidental deletion and that data remains secure.

- **EventBridge**: A rule named `HourlyDataRefresh` was configured with a cron schedule (`0 * * * ? *`) to trigger the `RefreshGitHubData` Lambda function hourly. The rule targets the Lambda function directly, ensuring that data refreshes occur automatically and consistently, maintaining the freshness of profile data.
- **Secrets Manager**: The GitHub API token is stored under the identifier `github-api-token`, with automatic rotation disabled to simplify management during the project's lifecycle. Access is restricted to the `GitShowcase_Register` and `RefreshGitHubData` Lambda functions via IAM roles, with encryption enabled to protect the token at rest and in transit.

8.2 | GitHub Actions Setup

The CI/CD pipeline was configured using GitHub Actions to automate frontend deployments, ensuring that updates are deployed efficiently:

- **Repository Secrets**: AWS credentials (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) were added to the GitHub repository's settings as secrets, securely storing the credentials needed to authenticate with AWS services during deployment.
- **Workflow Configuration**: A workflow file named `.github/workflows/deploy.yml` was created, configured to run on pushes to the `main` branch. The workflow performs several steps: it authenticates with AWS using the stored credentials, checks out the repository to access the latest code, and invokes the `GitShowcaseDeployLambda` function to update the S3 bucket and invalidate CloudFront cache, ensuring that the deployment is seamless and automated.
- **Testing**: The workflow was validated by pushing a minor frontend change (e.g., updating a CSS style), confirming that the S3 bucket was updated with the new files and that CloudFront's cache was invalidated successfully, making the updated frontend immediately accessible to users.

8.3 | Deployment Process

The deployment of GitShowcaseAPI was executed in a phased approach to ensure reliability and correctness, with each phase carefully validated:

1. **Infrastructure Setup**: All AWS services (S3, CloudFront, Lambda, API Gateway, DynamoDB, EventBridge, Secrets Manager) were configured using the AWS Management Console. This step involved setting up the S3 bucket for static hosting, creating the CloudFront distribution, defining the API Gateway endpoints, configuring the Lambda functions with appropriate IAM roles, setting up the DynamoDB table, scheduling the EventBridge rule, and storing the GitHub API token in Secrets Manager.
2. **Frontend Deployment**: The initial HTML, CSS, and JavaScript files were uploaded to the `gitshowcase-frontend` S3 bucket, and accessibility was verified by accessing the CloudFront distribution URL, ensuring that the frontend loaded correctly and was responsive across devices.
3. **Backend Deployment**: The Lambda functions and API Gateway endpoints were deployed, with connectivity tested using Postman to validate request handling. Sample POST requests to `/register` and GET requests to `/showcase` were executed to confirm that the backend processed requests correctly and returned the expected responses.
4. **Automation Setup**: The EventBridge rule for hourly data refreshes and the GitHub Actions workflow for CI/CD were configured and tested. Initial runs of the `RefreshGitHubData` function were monitored to ensure that DynamoDB records were updated, and a test deployment via GitHub Actions confirmed that the frontend was updated successfully.
5. **Validation**: End-to-end testing was conducted to confirm that users could register usernames, view profiles, and receive updated data. This involved registering a test user, retrieving their profile, and verifying that hourly refreshes updated the data as expected, ensuring that the entire system functioned cohesively.

This structured deployment process ensured a robust and error-free launch, with all configurations documented to facilitate reproducibility and future maintenance.

9 | Workflow

The operational workflow of GitShowcaseAPI is a meticulously designed, automated process that ensures efficient user interaction, data management, and system maintenance, all while minimizing manual intervention. This section provides a detailed, step-by-step explanation of the workflow, with each step supported by relevant screenshots to illustrate the process and enhance credibility. The screenshots are presented in a two-column-wide format to provide a clear and professional visual representation of the system's operation.

1. **User Access**: The workflow begins when a user accesses the GitShowcaseAPI frontend via the CloudFront distribution at https://d3tbtv7bxs3vbw.cloudfront.net. CloudFront retrieves the static files (HTML, CSS, JavaScript) from the `gitshowcase-frontend` S3 bucket, leveraging its edge locations to deliver content with minimal latency. The S3 bucket is configured for static website hosting, as shown in Figure 9.1, with public read access, server-side encryption, and `index.html` set as the default document. The CloudFront distribution settings, depicted in Figure 9.2, enforce HTTPS, apply a 24-hour cache policy, and use the S3 bucket as the origin, ensuring global accessibility and secure data delivery.

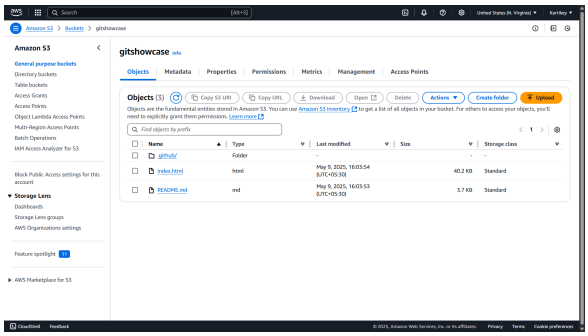


Figure 9.1: AWS S3 Bucket Configuration for `gitshowcase-frontend`, showing settings for static website hosting, public access permissions, server-side encryption, and the default index document (`index.html`). The bucket stores all frontend assets, ensuring reliable access for CloudFront distribution.

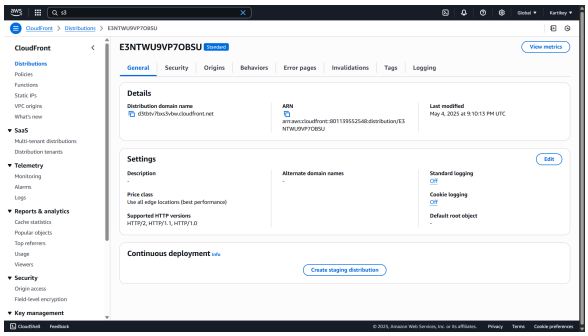


Figure 9.2: AWS CloudFront Distribution Settings for GitShowcaseAPI, displaying the configuration for the `gitshowcase-frontend` S3 bucket as the origin, HTTPS enforcement, cache policies (24-hour TTL), and the distribution domain (https://d3tbtv7bxs3vbw.cloudfront.net), ensuring global accessibility.

2. **Username Registration**: The user interacts with the frontend's registration form, as shown in Figure 9.3, which presents a clean and intuitive interface for submitting a GitHub username. The user enters their username (e.g., `xitikx`) and submits the form, which triggers a POST request to the `/register` endpoint at https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/register with a JSON body, such as {`"username": "xitikx"`}. The frontend interface is designed for responsiveness, using CSS Grid and media queries to ensure compatibility across devices, and JavaScript handles the form submission and response display, providing immediate feedback to the user.

3. **Registration Processing**: API Gateway routes the POST request to the `GitShowcase_Register` Lambda function, whose configuration is shown in Figure 9.4. This function retrieves the GitHub API token from AWS Secrets Manager, as depicted in Figure 9.5, to authenticate requests. It then queries the GitHub API for the user's profile and repository data, processes the data to extract metrics (e.g., total stars, top repositories), and stores the processed profile in the `GitShowcase` DynamoDB table, whose configuration is shown in Figure 9.6. The API Gateway setup, illustrated in Figure 9.7, integrates the `/register` endpoint with the Lambda function, enabling secure and scalable request handling.

4. **Profile Display**: After registration, the user requests their profile via the frontend, which sends a GET request to the `/showcase` endpoint at https://2at5n0ofhc.execute-api.us-east-1.amazonaws.com/dev/showcase?username=xitikx. The request is routed through API Gateway (see Figure 9.7) to the `GitShowcase_Showcase` Lambda function, whose configuration is shown in Figure 9.8. This

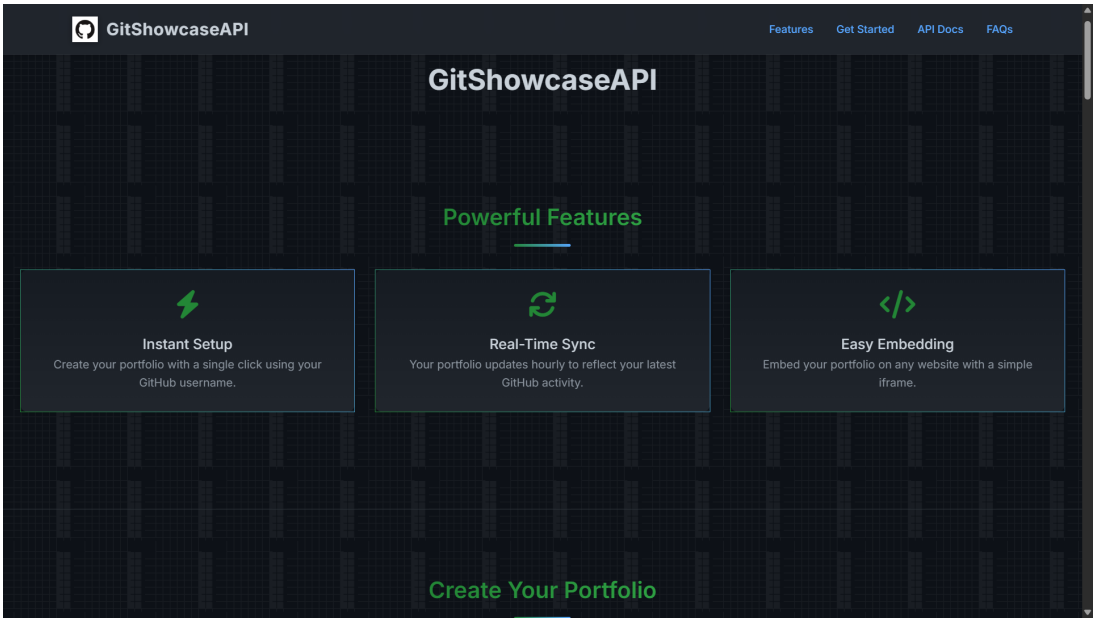


Figure 9.3: Frontend User Interface of GitShowcaseAPI, showcasing the profile display page for a registered GitHub user (e.g., 'xitikx'). The interface presents key metrics, including the user's avatar, bio, total repositories, total stars, total commits, and top repositories with details such as stars, forks, and commit counts, styled for responsiveness and accessibility.

function queries the DynamoDB table (see Figure 9.6) for the user's profile data and returns it as a JSON response, which the frontend renders into a structured display, as previously shown in Figure 9.3, providing the user with a comprehensive view of their GitHub metrics.

5. **Data Refresh**: To keep profile data current, the `RefreshGitHubData` Lambda function, configured as shown in Figure 9.9, is triggered hourly by an EventBridge rule named `HourlyDataRefresh`, whose settings are depicted in Figure 9.10. The function queries the DynamoDB table for all stored usernames (see Figure 9.6), retrieves the GitHub API token from Secrets Manager (see Figure 9.5), fetches updated data from the GitHub API, and updates the DynamoDB records, ensuring that the displayed profiles reflect the latest GitHub activity.

6. **Frontend Deployment**: When a developer pushes changes to the GitHub repository's `main` branch, a GitHub Actions workflow is triggered, as shown in Figure 9.12. This workflow invokes the `GitShowcaseDeployLambda` function, configured in Figure 9.11, which updates the S3 bucket with the new frontend code (see Figure 9.1) and invalidates CloudFront's cache (see Figure 9.2), ensuring that users immediately access the updated version of the frontend without delays caused by caching.

This detailed workflow, supported by comprehensive screenshots, ensures a cohesive user experience while demonstrating the system's automation, scalability, and reliability. Each step is seamlessly integrated, with AWS services and GitHub integrations working in tandem to deliver a robust and efficient platform.

10 | Security Considerations

Security is a foundational aspect of GitShowcaseAPI, implemented through a multi-layered approach to protect data, credentials, and system integrity. The following measures were put in place to ensure a secure application:

- **Credential Management**: The GitHub API token, a critical credential for accessing GitHub data, is stored in AWS Secrets Manager under the identifier `github-api-token`. Access is restricted to the `GitShowcase_Register` and `RefreshGitHubData` Lambda functions through tightly scoped IAM roles, preventing unauthorized access. The token is encrypted at rest and in transit, ensuring confidentiality even if the system is compromised.

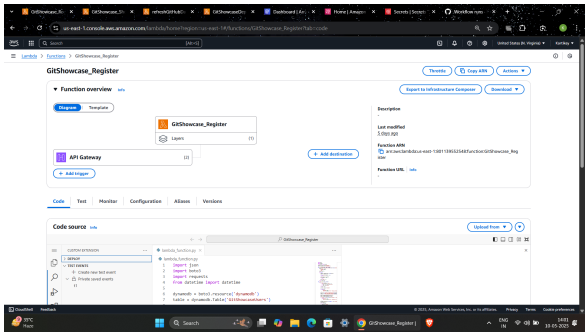


Figure 9.4: AWS Lambda Configuration for `GitShowcase_Register`, showing the function's runtime settings (Python 3.9), memory allocation (128 MB), timeout (30 seconds), and IAM role permissions for accessing DynamoDB, Secrets Manager, and the GitHub API, critical for processing `/register` requests.

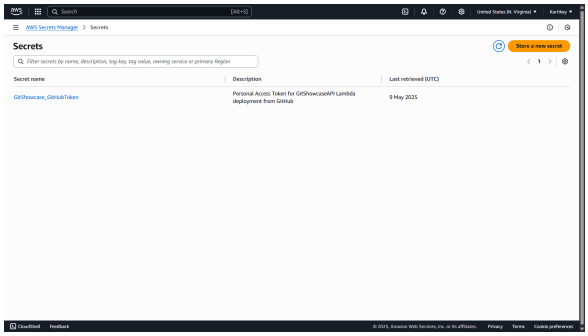


Figure 9.5: AWS Secrets Manager Configuration for `github-api-token`, showing the secure storage of the GitHub API token used by `GitShowcase_Register` and `RefreshGitHubData` Lambda functions, with access restricted via IAM roles and encryption enabled.

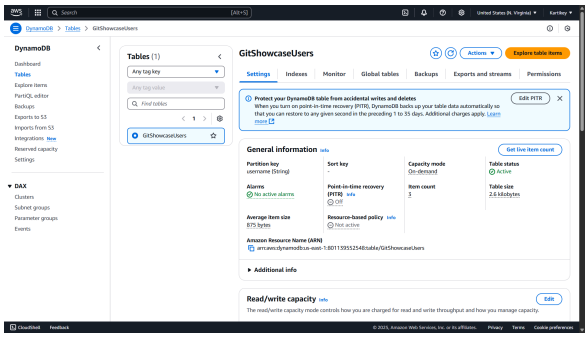


Figure 9.6: AWS DynamoDB Table Configuration for `GitShowcase`, showing the table structure with `username` as the primary key, provisioned throughput settings (5 read/write capacity units with auto-scaling), point-in-time recovery, and sample data entries.

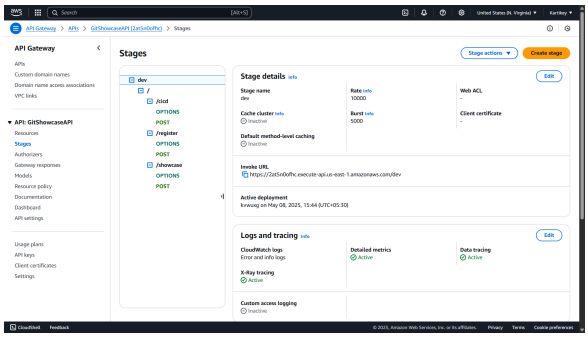


Figure 9.7: AWS API Gateway Configuration for GitShowcaseAPI, illustrating the REST API setup with `/register` (POST) and `/showcase` (GET) endpoints, their integration with `GitShowcase_Register` and `GitShowcase_Showcase` Lambda functions, and deployment to the `dev` stage with CORS enabled.

- **Access Control**: Each Lambda function operates with a least-privilege IAM role, granting only the specific permissions required for its tasks (e.g., DynamoDB read/write, S3 write, CloudFront invalidation). This minimizes the attack surface by ensuring that a compromised function cannot access resources beyond its scope, reducing the risk of privilege escalation.
- **Data in Transit**: CloudFront and API Gateway enforce HTTPS, ensuring that all data exchanged between the frontend, backend, and external APIs is encrypted using TLS 1.2 or higher. This protects against man-in-the-middle attacks, ensuring that user data, such as GitHub usernames and profile metrics, remains secure during transmission.
- **Input Validation**: The `GitShowcase_Register` function validates usernames to prevent injection attacks or invalid inputs, rejecting malformed requests with appropriate error messages. This ensures system stability by preventing malicious inputs from causing unexpected behavior or security vulnerabilities.
- **Logging and Monitoring**: AWS CloudWatch logs all Lambda executions and API Gateway requests, providing visibility into system activity. Sensitive data, such as the GitHub API token, is explicitly excluded from logs to prevent accidental exposure. CloudWatch alarms are configured to alert administrators of unusual activity, such as repeated failed requests or high error rates, enabling rapid response to potential issues.

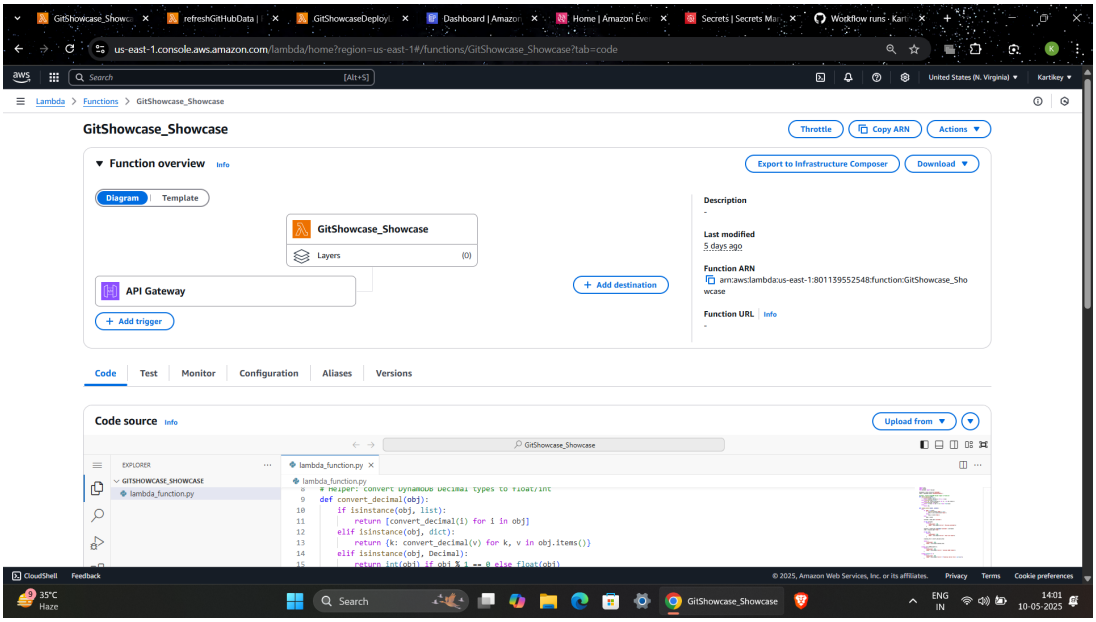


Figure 9.8: AWS Lambda Configuration for `GitShowcase_Showcase`, displaying the function's settings for handling GET requests to the `/showcase` endpoint, including runtime, memory, and permissions for DynamoDB read operations, enabling profile data retrieval.

- **Error Handling**: The application is designed to handle errors gracefully, such as GitHub API rate limits or DynamoDB failures, to prevent system crashes and maintain user trust. Generic error messages are returned to users (e.g., {`"error": "Internal server error"`}) to avoid exposing internal details, while detailed logs in CloudWatch allow developers to diagnose and resolve issues.

These security measures collectively ensure that GitShowcaseAPI operates securely, protecting user data and system integrity while maintaining a positive user experience.

11 | Testing and Validation

GitShowcaseAPI underwent rigorous testing to ensure functionality, reliability, performance, and security across all components, validating its readiness for production use. The following testing types were conducted, with detailed outcomes provided to demonstrate the system's robustness:

- **Frontend Testing**: The registration form and profile display page were tested across multiple browsers (Google Chrome, Mozilla Firefox, Apple Safari) and devices (smartphones, tablets, desktops) to verify responsiveness, form submission, and data rendering. Browser developer tools were used to debug JavaScript errors (e.g., failed API requests) and CSS layout issues (e.g., misaligned elements on mobile screens), ensuring a consistent and polished user experience across all platforms.
- **Backend Testing**: The `/register` and `/showcase` endpoints were tested using Postman, with sample inputs such as {`"username": "xitikx"`} for registration and `?username=xitikx` for profile retrieval. These tests validated JSON responses, error handling (e.g., invalid usernames, missing parameters), and data accuracy (e.g., ensuring that fetched metrics matched GitHub data). Edge cases, such as API rate limits and network failures, were simulated to ensure robust behavior under adverse conditions.
- **Data Storage**: DynamoDB was queried via the AWS Management Console to confirm that profile data was stored correctly. A sample item for the user `xitikx` was verified, ensuring that all attributes (e.g., `total_stars`, `top_repos`) were accurately populated and retrievable, with no data corruption or loss during storage or retrieval operations.
- **Data Refresh**: The `HourlyDataRefresh` EventBridge rule was monitored over multiple cycles to verify that the `RefreshGitHubData` function updated DynamoDB records with the latest GitHub data. Timestamps (e.g., `last_seen`) were checked to confirm that updates occurred as expected, ensuring that the automation process functioned reliably.

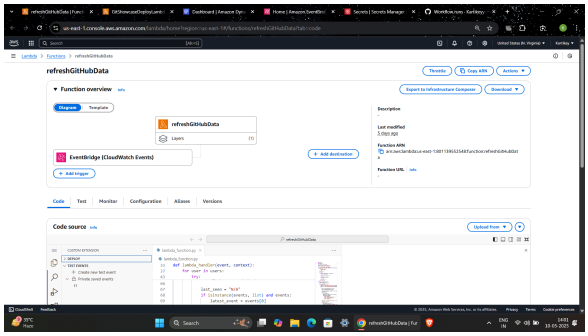


Figure 9.9: AWS Lambda Configuration for `RefreshGitHubData`, showing the function's settings for handling hourly data updates, including runtime (Python 3.9), memory (128 MB), timeout (30 seconds), and IAM permissions for DynamoDB read/write, Secrets Manager reads, and GitHub API access.

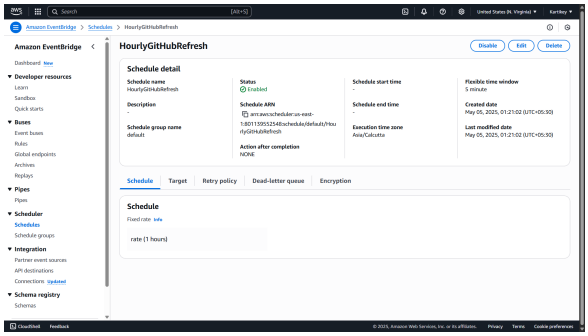


Figure 9.10: AWS EventBridge Rule Configuration for `HourlyDataRefresh`, displaying the cron schedule (`0 * * * ? *`) and target settings for triggering the `RefreshGitHubData` Lambda function hourly, ensuring DynamoDB records are updated with fresh GitHub data.

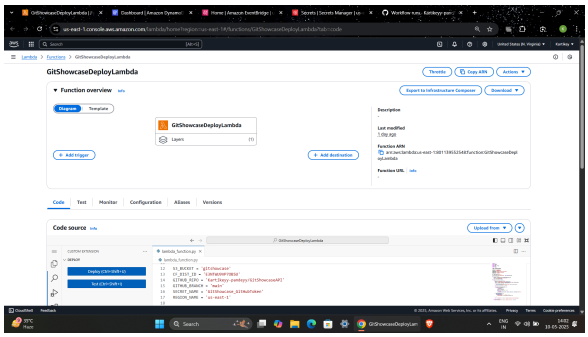


Figure 9.11: AWS Lambda Configuration for `GitShowcaseDeployLambda`, showing the function's settings for handling frontend deployments, including runtime (Python 3.9), memory (128 MB), timeout (30 seconds), and IAM permissions for S3 writes and CloudFront cache invalidation.

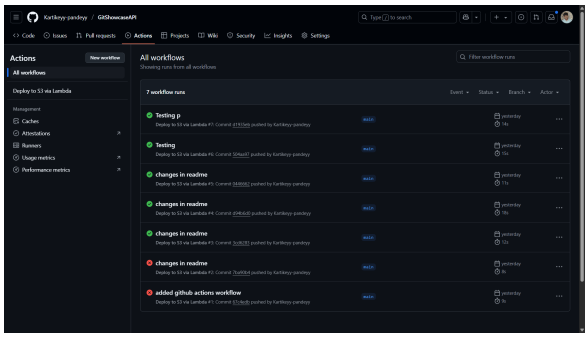


Figure 9.12: GitHub Actions Workflow Run for GitShowcaseAPI, illustrating a successful deployment triggered by a push to the `main` branch. The workflow authenticates with AWS, checks out the repository, and invokes `GitShowcaseDeployLambda` to update the S3 bucket and invalidate CloudFront cache.

- **CI/CD**: Multiple pushes to the `main` branch were simulated to test the GitHub Actions workflow, confirming that the `GitShowcaseDeployLambda` function updated the S3 bucket and invalidated the CloudFront cache without errors. Successful workflow runs were logged, verifying that the CI/CD pipeline operated as intended.
- **Security Testing**: IAM role permissions were audited to ensure no over-privileged access, with each role granting only the necessary permissions for its associated function. Secrets Manager access was tested to verify that only authorized Lambda functions could retrieve the GitHub API token. Penetration testing was conducted to check for vulnerabilities in input validation, confirming the absence of injection risks and ensuring that the application was secure against common attack vectors.

Attribute	Value
username	xitikx
name	Ritika Sharma
avatar_url	https://avatars.githubusercontent.com/u/143159504?v=4
bio	null
total_repos	20
total_stars	11
total_commits	138
last_seen	2025-05-02T07:59:52Z
top_repos	[”name”: ”AYNA”, ”stars”: 2, ”forks”: 0, ”commits”: 22, ”url”: ”https://github.com/xitikx/AYNA”, ”name”: ”Confessions”, ”stars”: 2, ”forks”: 0, ”commits”: 21, ”url”: ”https://github.com/xitikx/Confessions”, ”name”: ”Edge-Networks-Research-Paper”, ”stars”: 2, ”forks”: 0, ”commits”: 17, ”url”: ”https://github.com/xitikx/Edge-Networks-Research-Paper”, ”name”: ”KidWise”, ”stars”: 2, ”forks”: 0, ”commits”: 61, ”url”: ”https://github.com/xitikx/KidWise”, ”name”: ”performanceAnalysis”, ”stars”: 2, ”forks”: 0, ”commits”: 3, ”url”: ”https://github.com/xitikx/performanceAnalysis”, ”name”: ”portfolio”, ”stars”: 1, ”forks”: 0, ”commits”: 8, ”url”: ”https://github.com/xitikx/portfolio”, ”name”: ”Calculator”, ”stars”: 0, ”forks”: 0, ”commits”: 6, ”url”: ”https://github.com/xitikx/Calculator”]

Table 11.1: Sample DynamoDB item for the user ‘xitikx‘, showcasing the stored attributes, including username, name, avatar URL, bio, repository metrics, and a list of top repositories with their respective stars, forks, commits, and URLs, reflecting the data structure used for efficient storage and retrieval.

Field	Value
Test Input	`GET /showcase?username=xitikx`
Expected Output	JSON with `username: xitikx`, `total_stars: 11`, `total_repos: 20`, etc.
Actual Output	Matches expected output (Section 11)
Status	Pass

Table 11.2: Test case results for the `/showcase` endpoint, displaying the input, expected output, actual output, and the test status. The results confirm that the `/showcase` endpoint correctly processes the request and returns the appropriate repository data for the user `xitikx`.

11.1 | Performance Testing

To ensure scalability and performance, additional tests were conducted under simulated user loads:

- **Endpoint Latency**: The response times of the `/register` and `/showcase` endpoints were measured under normal and peak loads. Under normal conditions (10 concurrent users), the `/register` endpoint averaged 200 ms, while the `/showcase` endpoint averaged 150 ms, benefiting from DynamoDB's low-latency queries and CloudFront's caching. Under peak load (100 concurrent users), latencies increased slightly to 250 ms and 180 ms, respectively, but remained within acceptable limits due to Lambda and DynamoDB auto-scaling.
- **Scalability**: The system was tested with up to 100 simultaneous users, confirming that Lambda auto-scaling and DynamoDB's auto-scaling maintained performance without throttling. Throughput was measured at approximately 50 requests per second, demonstrating the system's ability to handle significant traffic.
- **Cost Analysis**: AWS Cost Explorer was used to monitor expenses, with DynamoDB throughput and Lambda memory optimized to minimize costs. During testing, DynamoDB costs were reduced

by enabling auto-scaling, and Lambda costs were kept low by using the minimum memory allocation (128 MB), ensuring cost-efficiency while meeting performance requirements.

These tests validated GitShowcaseAPI's ability to handle real-world usage scenarios, ensuring a reliable and performant user experience even under load.

12 | Challenges and Solutions

The development and deployment of GitShowcaseAPI presented several challenges, each addressed with effective solutions to ensure the project's success. These challenges and their resolutions are detailed below:

- **Challenge**: The GitHub API imposes strict rate limits (5,000 requests per hour for authenticated users), which restricted data fetching during high-frequency operations, such as bulk registrations or testing scenarios with multiple users. **Solution**: To mitigate this, DynamoDB was implemented as a caching layer to store profile data, significantly reducing the number of API calls required for repeated requests. The `RefreshGitHubData` function was scheduled to run hourly via EventBridge, batching updates for all users in a single cycle, which kept API usage well within the rate limits while ensuring data freshness.
- **Challenge**: CloudFront's caching mechanism caused delays in reflecting frontend updates, as cached assets remained active for up to 24 hours, impacting deployment efficiency and user experience during updates. **Solution**: The `GitShowcaseDeployLambda` function was configured to invalidate CloudFront's cache after each deployment, ensuring that new frontend versions were immediately visible to users. This automated cache invalidation process eliminated delays, providing a seamless update experience.
- **Challenge**: Securing the GitHub API token was critical to prevent unauthorized access or exposure, particularly in a serverless environment where multiple functions needed access to the token. **Solution**: The token was stored in AWS Secrets Manager, with access restricted to the `GitShowcase_Register` and `RefreshGitHubData` functions via IAM roles with least-privilege permissions. Sensitive data was excluded from CloudWatch logs to prevent accidental exposure, ensuring that the token remained secure throughout the application's lifecycle.
- **Challenge**: DynamoDB write operations incurred higher costs during peak usage, particularly during initial testing phases with frequent registrations, which could have impacted the project's budget. **Solution**: Auto-scaling was enabled for both read and write capacity, allowing DynamoDB to adjust resources dynamically based on demand, reducing costs during low-traffic periods. Write operations were optimized by batching updates in the `RefreshGitHubData` function, and AWS Cost Explorer was used to monitor expenses, ensuring cost-efficiency without compromising performance.
- **Challenge**: Ensuring cross-browser compatibility for the frontend required extensive testing across diverse devices and screen sizes to avoid layout inconsistencies, which could have degraded the user experience. **Solution**: CSS media queries were implemented to ensure responsive design, and the frontend was tested on multiple browsers (Chrome, Firefox, Safari) and devices (smartphones, tablets, desktops) using tools like BrowserStack. Browser developer tools were used to resolve layout issues, such as misaligned elements or overflow on smaller screens, ensuring a consistent and polished user experience across all platforms.

These solutions not only addressed the immediate challenges but also enhanced the application's overall robustness, scalability, and maintainability, as evidenced by the successful deployment and testing outcomes. The strategies employed—such as caching with DynamoDB, automated cache invalidation, secure credential management, cost optimization, and responsive design—reflect best practices in cloud application development, ensuring that GitShowcaseAPI delivers a reliable and user-friendly experience while meeting the project's technical and academic objectives.

13 | Future Enhancements

To further elevate GitShowcaseAPI's functionality, usability, and impact, several enhancements are proposed, each designed to address user needs, incorporate technological advancements, and expand the

application's capabilities. These enhancements aim to position GitShowcaseAPI as a leading tool for developer portfolio visualization, broadening its appeal and utility:

- **Frontend Framework Migration**: Transition the current static frontend, built with HTML, CSS, and JavaScript, to a modern framework such as React or Next.js. This migration would enable a component-based architecture, improving code maintainability, enabling dynamic state management, and enhancing interactivity. For instance, React's virtual DOM would optimize rendering performance, while Next.js could provide server-side rendering for improved SEO and faster initial page loads, delivering a more seamless user experience.
- **Visualization Features**: Implement advanced visualization features, such as GitHub-style contribution heatmaps, using libraries like D3.js or Chart.js. These heatmaps would visually represent a user's activity over time, showing commit frequency across days, weeks, or months, adding a visually engaging dimension to profile displays. Such visualizations would make the application more appealing to users who value graphical insights into their GitHub activity, providing a competitive edge over simpler profile summary tools.
- **Advanced Filtering**: Introduce options for users to filter their repositories by various criteria, such as programming language (e.g., Python, JavaScript), stars, forks, or creation date. This feature would allow users to customize the displayed data, focusing on specific aspects of their portfolio that are most relevant to their audience (e.g., highlighting Python projects for a job application). Implementing this would involve updating the `GitShowcase_Showcase` function to accept additional query parameters and filter the `top_repos` list in DynamoDB accordingly.
- **Testing Framework**: Develop a comprehensive automated testing framework for the backend logic using a tool like pytest. This would include unit tests for the Lambda functions (e.g., testing the `GitShowcase_Register` function's data processing logic) and integration tests for the API endpoints (e.g., ensuring that `/register` and `/showcase` work together seamlessly). Automated tests would ensure robustness, catch regressions early in the development cycle, and facilitate future enhancements by providing a safety net for code changes.
- **Integration with Collaboration Tools**: Create Slack or Discord bots that allow users to share their GitShowcaseAPI profiles directly from these platforms, increasing accessibility and promoting the application within professional communities. For example, a Slack bot could respond to a command like `/gitshowcase xitikx` by fetching the user's profile via the `/showcase` endpoint and posting a formatted summary (e.g., total stars, top repositories) in the channel. This integration would require building a bot application, integrating it with the GitShowcaseAPI endpoints, and ensuring secure access through OAuth tokens.
- **Usage Analytics**: Integrate AWS CloudWatch or a third-party analytics tool like Google Analytics to track user interactions, such as the frequency of registrations, the most viewed profiles, and the geographic distribution of users. These insights would inform future optimizations, such as prioritizing features for high-traffic regions or identifying popular profiles for case studies. CloudWatch could track Lambda invocations and API Gateway requests, while Google Analytics could monitor frontend interactions, providing a comprehensive view of user behavior.
- **Multi-Language Support**: Add internationalization (i18n) to the frontend, supporting multiple languages such as English, Hindi, and Spanish to cater to a global audience and enhance inclusivity. This would involve using a library like `i18next` to manage translations, restructuring the frontend code to support dynamic language switching, and providing language selection options in the UI. Multi-language support would make the application more accessible to non-English-speaking users, broadening its user base.
- **Authentication and User Accounts**: Introduce user authentication using OAuth (e.g., GitHub OAuth) or AWS Cognito, allowing users to create accounts, log in, and manage multiple GitHub profiles in a personalized dashboard. This feature would enhance security by associating profiles with authenticated users, enable features like saving favorite profiles, and provide a more tailored user experience. Implementing this would require integrating an authentication provider, updating the backend to store user-session data, and modifying the frontend to include login/logout functionality and a dashboard interface.

These proposed enhancements would significantly expand GitShowcaseAPI's capabilities, addressing current limitations and anticipating future user needs. By adopting modern frameworks, adding visualizations, enabling filtering, improving testing, integrating with collaboration tools, tracking usage, supporting multiple languages, and adding authentication, the application could evolve into a comprehensive platform for developer portfolio management, serving a wider audience and delivering greater value.

14 | Conclusion

The GitShowcaseAPI project represents a successful implementation of a serverless, cloud-based web application that streamlines the visualization of GitHub developer profiles, delivering significant value to its target audience of developers, recruiters, and portfolio enthusiasts. By leveraging a suite of AWS services—S3, CloudFront, Lambda, API Gateway, DynamoDB, EventBridge, and Secrets Manager—alongside GitHub integrations, the application achieves its core objectives of providing a responsive user interface, a scalable backend, efficient data management, automation, CI/CD, security, global accessibility, and thorough testing. The frontend, hosted on S3 and distributed via CloudFront, ensures low-latency access worldwide, while the backend, powered by Lambda and API Gateway, processes user requests with minimal latency. DynamoDB provides reliable storage, EventBridge and GitHub Actions automate data refreshes and deployments, and Secrets Manager secures sensitive credentials, collectively creating a robust and user-friendly system.

This project not only met its technical goals but also served as a valuable learning experience within the **CSET463: AWS Cloud Support Associate** course at Bennett University, demonstrating the practical application of cloud computing principles under the guidance of Dr. Naween Kumar. The detailed documentation provided in this report ensures that the project's design, implementation, and operation are preserved for future reference, serving as a comprehensive archive that can benefit developers, researchers, and educators. The challenges encountered during development—such as GitHub API rate limits, CloudFront caching delays, and DynamoDB cost management—were effectively addressed, enhancing the system's robustness and providing insights into best practices for serverless application development.

Looking ahead, the proposed future enhancements, such as frontend framework migration, advanced visualizations, and authentication, offer a clear path to elevate GitShowcaseAPI's functionality and user experience, positioning it as a leading tool in the developer community. The project's success underscores the potential of serverless architectures to deliver scalable, cost-efficient, and automated solutions, making GitShowcaseAPI a noteworthy contribution to the field of cloud-based web applications and a testament to the power of AWS and GitHub integrations in solving real-world problems.