



Serverless Todo App with AWS

EXECUTIVE SUMMERY

Project Goal

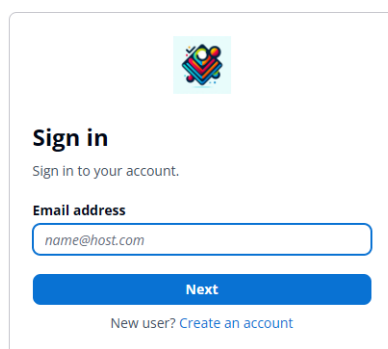
This project demonstrates the development of a fully serverless, secure, and scalable Todo List web application using native AWS services. Although the application is simple by design, it showcases practical use cases and the potential of a serverless approach. It includes authentication, data persistence, API integration, and frontend hosting—entirely without server management—highlighting real-world skills in building modern, cloud-native, scalable applications following AWS best practices.

Technologies Used

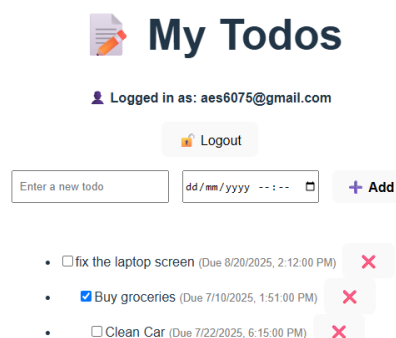
- Frontend: React (Vite) hosted on AWS Amplify
- Authentication: Amazon Cognito (User Pool, Hosted UI, JWT)
- API Layer: Amazon API Gateway (HTTP API)
- Compute: AWS Lambda (Python) for all backend logic
- Database: Amazon DynamoDB (NoSQL key-value store)
- Infrastructure & Logs: AWS IAM (for permissions), Amazon CloudWatch (for monitoring & debugging)

Outcome

- Deployed a fully working Todo App that supports:
 - User sign-in/sign-out with JWT
 - Secure create, read, update, and delete (CRUD) of todos
 - Token-based authorization for all API calls
 - Dynamically updated UI with real-time feedback
- The app is live on a public staging environment and connected to a real AWS backend with IAM-secured components.



The screenshot shows a 'Sign in' form with a logo at the top. Below the logo is the text 'Sign in' and 'Sign in to your account.' There is a text input field for 'Email address' with the placeholder 'name@host.com'. Below the input field is a blue 'Next' button. At the bottom, there is a link that says 'New user? Create an account'.

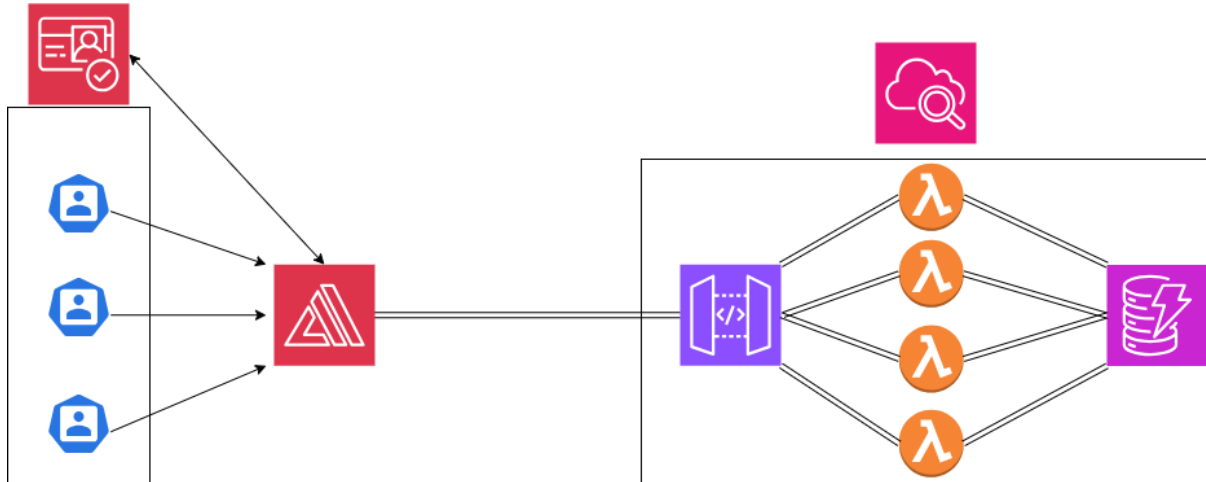


The screenshot shows the 'My Todos' page. At the top, it says 'Logged in as: aes6075@gmail.com' with a 'Logout' button. Below this is a form to 'Enter a new todo' with a date/time picker and an 'Add' button. At the bottom, there is a list of todos: 'fix the laptop screen (Due 8/20/2025, 2:12:00 PM)', 'Buy groceries (Due 7/10/2025, 1:51:00 PM)', and 'Clean Car (Due 7/22/2025, 6:15:00 PM)'. Each todo has a checkbox and a red 'X' icon.



Architecture Overview

This project follows a modern serverless architecture, leveraging fully managed AWS services to create a secure, scalable, and cost-efficient web application. The diagram below illustrates the flow of data and interaction between the key components:



Components Breakdown

- Frontend (React + AWS Amplify Hosting):
The frontend is a single-page application built with React and hosted using AWS Amplify, which provides secure and scalable static site hosting. Code updates can be deployed directly by uploading builds, or integrated with source control tools for automated deployment.
- Authentication (Amazon Cognito):
Amazon Cognito handles user sign-in and sign-out through a hosted UI. After login, users receive an ID token (JWT) that is used for secure access to backend resources.
- API Layer (Amazon API Gateway – HTTP API):
The backend API is exposed via Amazon API Gateway using HTTP API, offering lower latency and simplified setup. Routes are protected using JWT authorizers integrated with Cognito.
- Compute (AWS Lambda – Python):
Business logic is implemented in Python using AWS Lambda. Each function is triggered by an HTTP request and performs a specific action (create, delete, list todos). Lambda ensures automatic scaling and pay-per-use execution.
- Data Store (Amazon DynamoDB):
DynamoDB is used to persist todo items in a NoSQL schema, optimized for fast and predictable performance. The table uses user_id and todo_id to isolate data per user.
- IAM & Security:
API Gateway validates JWT tokens from Cognito before forwarding requests to Lambda. IAM roles are configured with least-privilege policies, ensuring that Lambda functions can only access the specific resources required.



Development Process

The development process followed a logical, layered approach focused on clarity, modularity, and leveraging AWS managed services effectively. At startd from the back to the front/ setting up a DB layer then a processing layer and then a front and user management layer.

The project uses **Amazon DynamoDB** with `user_id` as the partition key, enabling secure, user-specific data isolation. Chosen for its high performance, serverless architecture, and automatic scalability, DynamoDB supports flexible, schema-less data storage—ideal for evolving task attributes. its key-value model ensures fast, efficient access to a per-user data.

Next, a series of **AWS Lambda** functions were implemented in Python, with each function handling a distinct operation: creating, retrieving and deleting todos. These functions were tested individually using the Lambda console's built-in test feature, simulating different payloads and edge cases. All Lambda functions were assigned the same IAM role, configured with scoped permissions for DynamoDB read/write access and CloudWatch logging to simplify policy management and maintain centralized logging.

With the business logic validated, an **HTTP API** was provisioned to expose these Lambda functions via RESTful endpoints. Each route was explicitly mapped to its corresponding Lambda function. Manual testing was performed using curl to verify responses, authorization, and error handling. During this phase, a **CORS (Cross-Origin Resource Sharing)** issue was identified — a common challenge in browser-based applications. The API Gateway was updated to handle preflight OPTIONS requests and return appropriate Access-Control-Allow-Origin headers to allow requests from any where for testing and later on from only the Amplify-hosted frontend.

Authentication was added by integrating **Amazon Cognito**, configuring a user pool and enabling a hosted login UI. The HTTP API was secured using JWT validation via a Cognito authorizer. Token parsing and management were handled on the frontend, ensuring that only valid authenticated requests were sent to the backend.

The **frontend** was built using **React**, featuring a clean and minimalistic UI for managing todos. It was deployed on **AWS Amplify**, benefiting from its built-in hosting, SSL, and integration with static frontends. Token handling, conditional rendering, and fetch requests were implemented to securely interact with the backend.

To support observability and efficient debugging, both AWS Lambda functions and the HTTP API Gateway were integrated with **Amazon CloudWatch Logs**. Each Lambda invocation generated logs capturing input events, processing steps, and errors, enabling quick identification of issues such as invalid payloads or failed DynamoDB operations. Similarly, CloudWatch logging was enabled for API Gateway to trace incoming requests, response statuses, and integration errors—particularly useful during early testing and while resolving CORS and authorization issues. This centralized logging approach provided full visibility into the application's behaviour across layers and served as a valuable tool for both development and ongoing monitoring.

Key Challenges Encountered:

- **CORS Configuration:** Required correct setup of OPTIONS handling and headers in API Gateway for frontend compatibility.



- **JWT Session Management:** Ensured expired or missing tokens were handled gracefully in the frontend to avoid bad calls of the api gateway.
- **Path Parameter Mapping:** Correctly configured path parameters in API Gateway to match the expected Lambda event structure.

AWS Services Used

AWS Lambda

Used to implement the backend logic for all operations (Create, Read, Delete) on the Todo items. Lambda was chosen for its serverless nature, eliminating the need to manage infrastructure, while providing automatic scaling and built-in integration with other AWS services such as API Gateway and DynamoDB.

Amazon DynamoDB

Served as the NoSQL database to persist todo items. Chosen for its scalability, performance, and native integration with Lambda. The table schema was designed with a partition key based on user_id to ensure efficient queries and user data separation in a multi-tenant environment.

Amazon API Gateway (HTTP API)

Used to expose RESTful endpoints for the frontend to communicate with the backend Lambda functions. HTTP API was chosen over REST API for its lower latency, simpler configuration, and built-in support for JWT authorizers. Route-to-function mapping and CORS were configured directly within API Gateway.

Amazon Cognito

Provided secure, managed user authentication. Cognito User Pools handled user registration, login, and token issuance. The HTTP API used Cognito as a JWT authorizer to restrict backend access to authenticated users only. The hosted UI simplified the login flow and token redirection for the frontend.

AWS Amplify (Hosting)

Used to deploy and host the React frontend. Amplify provides a fast, serverless hosting solution with HTTPS support. Chosen for its seamless deployment flow and compatibility with SPAs (single-page applications).

Amazon CloudWatch

Used to monitor Lambda, API gateway and dynamo DB, executions and capture logs. Provided insight into function behavior, helped diagnose issues (e.g., missing claims in JWTs), and confirmed whether a Lambda was invoked or failed before entry.

AWS IAM (Identity and Access Management)

Used to manage permissions for Lambda functions. A single IAM role was attached to all functions, scoped with policies for DynamoDB access and CloudWatch logging. IAM ensured least-privilege access principles were followed.



Security

Security was a core design principle throughout the development of this serverless application. The architecture applies AWS best practices to ensure secure access control, proper data segregation, and minimal surface area for potential vulnerabilities.

Authentication with Amazon Cognito

User authentication is managed using Amazon Cognito User Pools. Users log in through the hosted Cognito UI and receive an ID token (JWT), which is stored on the client and used for all subsequent API requests. No authentication logic or credential handling occurs on the frontend application.

Authorization with JWT-Based API Gateway Authorizer

The HTTP API is secured using a JWT authorizer configured to validate tokens against the Cognito User Pool. API Gateway enforces that only valid, authenticated requests reach the Lambda functions. Each Lambda function extracts the sub (subject) claim from the JWT to identify the user and enforce data-level isolation in the DynamoDB table.

IAM Roles and Least Privilege

All Lambda functions are assigned a dedicated IAM role that grants only the minimal required permissions:

- Read and write access to the specific DynamoDB table
- Write access to Amazon CloudWatch Logs

No administrative or wildcard permissions are used. All access is scoped using explicit resource ARNs.

CORS Configuration

Cross-Origin Resource Sharing (CORS) is enforced at the API Gateway level. The configuration permits requests only from the Amplify-hosted frontend domain. Preflight OPTIONS requests are automatically handled by API Gateway, and appropriate Access-Control-Allow-Origin headers are included in all relevant responses.

Secure Token Storage and Session Validation

Tokens are stored in browser localStorage. Before initiating API requests, the frontend validates token expiration and structure. If no valid token is present, the user is redirected to the Cognito login flow, ensuring that unauthorized or expired sessions do not result in backend invocation attempts.



Challenges & Solutions

Throughout the implementation of this serverless application, several real-world challenges arose that required careful troubleshooting and a deeper understanding of AWS service interactions. The following outlines the key obstacles encountered and how they were resolved.

CORS Configuration and Preflight Failures

Challenge:

Initial requests from the React frontend were blocked by the browser due to missing Access-Control-Allow-Origin headers. In some cases, OPTIONS preflight requests failed before reaching the Lambda backend.

Solution:

CORS was correctly configured in API Gateway, explicitly allowing the Amplify domain. Preflight responses were managed by enabling OPTIONS method integration with default passthrough behavior. After ensuring all routes had consistent CORS headers and that Lambda responses returned application/json, the issue was resolved.

JWT Session Expiry and Authorization Errors

Challenge:

Expired or missing Cognito tokens led to 500 or 401 errors from API Gateway. In some cases, Lambda functions assumed the sub claim was always present and failed with KeyError.

Solution:

Token parsing and expiration validation were implemented in the frontend. Before any backend interaction, the app checks token validity and redirects unauthenticated users to Cognito log in UI. Backend logic was hardened by validating token claims and returning structured error messages for invalid sessions.

Parameter Mapping in HTTP API Gateway

Challenge:

Lambda functions expected parameters via event ['path Parameters']['todold'], but requests failed due to incorrect route configurations. And even led to a CORS errors duo API gateway 404 and 500 response without the right headers.

Solution:

HTTP API routes were updated to include explicit path parameters (/delete-todo/{todold}), and parameter mapping was configured manually in the API Gateway console. The issue was tested and verified using curl before frontend integration.

Local Debugging and Lambda Test Events

Challenge:

Without an established CI/CD pipeline or centralized logging initially, validating individual Lambda functions was cumbersome. Diagnosing issues across API Gateway, Lambda, and DynamoDB interactions required manual event crafting and lacked observability.



Solution:

Each Lambda function was rigorously tested using the AWS Console's "Test" feature, simulating various payloads and error scenarios. Standard test events were saved and reused for consistency. Simultaneously, structured logging was implemented throughout the code to trace execution paths, inspect input/output, and surface failures. Logs were then monitored and filtered via Amazon CloudWatch Logs, enabling fast identification of logic errors, CORS issues, and permission failures.

This approach ensured high confidence in each function's behavior before integrating it into the full serverless workflow, and it laid the groundwork for production-grade observability practices.

Certainly — here's the updated version of the **Future Improvements** section, now with an added paragraph addressing **product-level enhancements** such as notifications and UX:

Future Improvements

While the current solution effectively demonstrates the power of a serverless architecture, several enhancements can further improve scalability, security, and maintainability.

Monitoring and Observability

Integrating centralized logging and tracing will significantly improve troubleshooting in production. AWS CloudWatch Logs Insights can help analyze logs across Lambda functions, while AWS X-Ray can provide full request tracing across API Gateway, Lambda, and DynamoDB. CloudWatch Alarms should also be configured to detect spikes in errors or latency.

Secrets and Configuration Management

Hardcoded values like table names and Cognito client IDs should be replaced with secure storage in AWS Systems Manager Parameter Store or AWS Secrets Manager. This enhances security, supports multi-environment setups, and enables cleaner deployment automation.

Infrastructure as Code (IaC)

Migrating manual AWS resource creation to AWS CloudFormation or the AWS CDK will allow version-controlled, repeatable deployments. This also lays the foundation for CI/CD integration and easier environment replication for staging or testing.

Scalability and Performance:

While the app already benefits from the inherent scalability of Lambda, API Gateway, and DynamoDB, production scenarios may benefit from DynamoDB auto-scaling, global tables for multi-region access, and concurrency limits on Lambda to control cost and throughput.

Product-Level Enhancements:

To increase real-world usability, several product features can be added. For example, implementing an email or push notification system using Amazon SES or SNS can alert users when a due date has passed. The UI/UX can be improved with more responsive design, loading indicators, accessibility support, and form validations. Additional functionality such as tagging, filtering, or recurring tasks could also enhance user engagement.



SUMMARY

This project successfully demonstrates the design and deployment of a fully serverless, secure, and scalable web application on AWS. Using native AWS services—such as Lambda, DynamoDB, API Gateway, Cognito, and Amplify—it showcases an end-to-end cloud-native architecture without the need to manage any underlying servers.

The application implements real-world patterns including authentication via JWT, RESTful API integration, fine-grained IAM permissions, and frontend hosting—all while adhering to principles of modularity, scalability, and pay-per-use cost efficiency. The process included iterative development, debugging with CloudWatch, and direct API testing via curl and the AWS Console.

Although the application is intentionally simple, it reflects architectural practices applicable to production-grade solutions and highlights the flexibility of serverless systems in AWS. The project also surfaces valuable experience with common challenges such as CORS management, token validation, and event parameter mapping.

Overall, this solution serves as a strong foundation for more advanced architectures and illustrates proficiency in AWS serverless development workflows.