Architecture Decision Records: Cloud Cost Calculator

This document contains a series of decisions made during the architecture and development of the Automated AWS Cost Calculator project. Each record outlines a specific technical choice, the context, the alternatives considered, and the final decision with its justification.

ADR-001: Choice of Infrastructure as Code (IaC) Tool

- **Status:** Accepted
- **Date:** 2025-10-15

Context:
The entire cloud infrastructure for this project must be managed as code to ensure repeatability, version control, and automated deployment. The primary choices for managing AWS infrastructure are AWS CloudFormation (and its abstractions like CDK/SAM) and HashiCorp Terraform.

Decision:
We have decided to use **Terraform** as the exclusive IaC tool for this project.

Alternatives Considered:

1. **AWS CloudFormation:**
    - *Pros:* Native AWS service, tight integration with other AWS tools, no state file to manage (managed by AWS).
    - *Cons:* Verbose syntax (YAML/JSON), can be difficult to read and write complex logic, less intuitive for managing multi-cloud or on-prem resources (though not a factor here).
2. **AWS CDK (Cloud Development Kit):**
    - *Pros:* Allows defining infrastructure in familiar programming languages (Python, TypeScript), providing high-level abstractions and the power of loops and conditionals.
    - *Cons:* Transpiles to CloudFormation, which can obscure the final state and make debugging difficult. Introduces a software development lifecycle for infrastructure which can be more complex than a declarative approach.

Justification:

- **Declarative & Readable:** Terraform's HCL (HashiCorp Configuration Language) is highly readable and declarative. It describes the desired *end state*, and Terraform figures out the "how," which simplifies the mental model for engineers.
- **Strong Modularity:** Terraform has a robust and easy-to-understand module system, which was critical for creating the reusable components (Lambda, KMS, S3, etc.) that form the basis of this project's clean architecture.
- **Ecosystem & Community:** Terraform is the industry standard for IaC with a massive community, extensive documentation, and a rich ecosystem of pre-built modules and providers. This makes it easier to find solutions and onboard new engineers.
- **Plan/Apply Workflow:** The terraform plan command provides a clear, human-readable execution plan before any changes are made. This is a critical safety feature that is more intuitive and powerful than CloudFormation's "change sets."

ADR-002: Frontend Hosting Architecture

- **Status:** Accepted
- **Date:** 2025-10-15

Context:
The project requires a public-facing, scalable, and secure way to host the static frontend application (HTML, CSS, JavaScript). The primary goal is to provide a reliable user interface for the dashboard while minimizing cost and operational overhead, and adhering to security best practices.

Decision:
We have decided to host the frontend using a **private S3 bucket** with a **public-facing CloudFront distribution** as the origin. Access from CloudFront to S3 will be restricted via an **Origin Access Identity (OAI)**.

Alternatives Considered:

1. **S3 Static Website Hosting:**
    - *Pros:* Extremely simple to set up, very low cost.
    - *Cons:* **Requires the S3 bucket to be publicly readable**, which is a major security concern flagged by tools like Checkov. Lacks native HTTPS support

without a custom domain. Does not provide the performance benefits of a CDN.

2. **Hosting on a Compute Service (e.g., EC2, Fargate):**
    - *Pros:* Full control over the web server (e.g., Nginx, Apache).
    - *Cons:* Massive overkill for a static site. Introduces significant operational overhead (patching, scaling, security groups, load balancers) and is far more expensive than a serverless object storage solution.

Justification:

- **Security:** This is the primary driver. By making the S3 bucket private and using an OAI, we completely eliminate the risk of direct, unauthenticated access to the bucket's contents. This architecture resolves an entire class of security findings.
- **Performance:** CloudFront acts as a Content Delivery Network (CDN), caching our frontend assets at edge locations around the world. This provides significantly lower latency for end-users compared to accessing the S3 bucket directly from a single region.
- **HTTPS by Default:** CloudFront provides a free, default SSL/TLS certificate, allowing us to serve our dashboard over HTTPS out of the box, which is a modern web standard.
- **Cost-Effectiveness:** While slightly more complex than direct S3 hosting, the cost difference for a low-traffic application is negligible, and the security and performance benefits far outweigh it.

ADR-003: CI/CD Authentication Method

- **Status:** Accepted
- **Date:** 2025-10-15

Context:
Our CI/CD pipeline in GitHub Actions requires programmatic access to our AWS account to provision infrastructure. This requires a secure authentication mechanism that adheres to the principle of least privilege and avoids long-lived credentials.

Decision:
We have decided to use **OpenID Connect (OIDC)** to establish a trust relationship between GitHub Actions and an AWS IAM Role.

Alternatives Considered:

1. **Long-Lived IAM User Access Keys:**
    - *Pros:* Simple to set up.
    - *Cons:* **Major security risk.** The access key and secret key must be stored as GitHub secrets. If these secrets are ever exposed, an attacker has permanent access to the AWS account. Rotating these keys is a manual, error-prone process. This is considered an anti-pattern for modern CI/CD.

Justification:

- **Keyless Authentication:** OIDC is a "keyless" method. The GitHub Actions runner requests a temporary, short-lived token from GitHub's OIDC provider. It then presents this token to AWS, which verifies its authenticity and allows the runner to assume an IAM Role. No permanent secrets are ever stored in GitHub.
- **Short-Lived Credentials:** The credentials obtained by assuming the role are temporary (typically valid for one hour), drastically reducing the window of opportunity for an attacker if they were ever compromised.
- **Granular Control:** The trust policy on the IAM Role can be scoped to a specific GitHub organization, repository, or even a specific branch (e.g., only allow deployments from the main branch). This provides fine-grained security control.
- **Industry Best Practice:** OIDC is the industry-standard, recommended best practice for authenticating CI/CD systems with cloud providers.

ADR-004: API Backend Architecture

- **Status:** Accepted
- **Date:** 2025-10-15

Context:
The frontend dashboard requires a secure, scalable, and cost-effective way to fetch cost data from AWS. The backend must not expose any AWS credentials to the client-side browser.

Decision:
We have decided to build a **serverless API using Amazon API Gateway (HTTP API) with a Lambda Proxy Integration.**

Alternatives Considered:

1. **Server-Based API (e.g., on EC2 or Fargate):**
   - *Pros:* Full control over the application runtime and environment.
   - *Cons:* Not cost-effective for an API with potentially infrequent traffic (pay for idle). Requires managing servers, scaling, patching, and a load balancer. High operational overhead.
2. **Direct S3 Data Fetch:**
   - *Pros:* Simple concept.
   - *Cons:* Requires a separate process to periodically write a JSON file to a public S3 bucket, which means the data would not be real-time. It also complicates the security model.

Justification:

- **Serverless & Cost-Effective:** We only pay when the API is actually called. There are no idle costs, which is perfect for this application.
- **Scalability:** Both API Gateway and AWS Lambda scale automatically and massively in response to traffic, requiring zero manual intervention.
- **Security:** This pattern provides a secure "front door" for our backend logic. The Lambda function's IAM role, which has permission to access Cost Explorer, is never exposed to the internet. The frontend only ever communicates with the public API Gateway endpoint.
- **Managed Service:** AWS manages the underlying infrastructure, availability, and security of both API Gateway and Lambda, dramatically reducing our operational burden.