Technical Solution Document: Dynamic HTML Service

Challenge: Serverless Dynamic HTML Service with Infrastructure as Code

Author: Hugo Herrera **Date:** August 1, 2025

Repository: https://github.com/HAHGX/dynamic-html-service-aws

---

Document Version: 1.0

Confidentiality: Internal

Table of Contents

1. Executive Summary

This document presents a comprehensive solution for serving dynamic HTML content using an AWS serverless architecture, implemented with Infrastructure as Code (IaC) principles. The solution leverages **AWS Lambda**, **API Gateway v2**, and **Systems Manager Parameter Store**, and is fully automated through **Terraform** and **GitHub Actions**. This design is closely aligned with the **AWS Well-Architected Framework**, ensuring a secure, high-performing, resilient, and cost-effective solution.

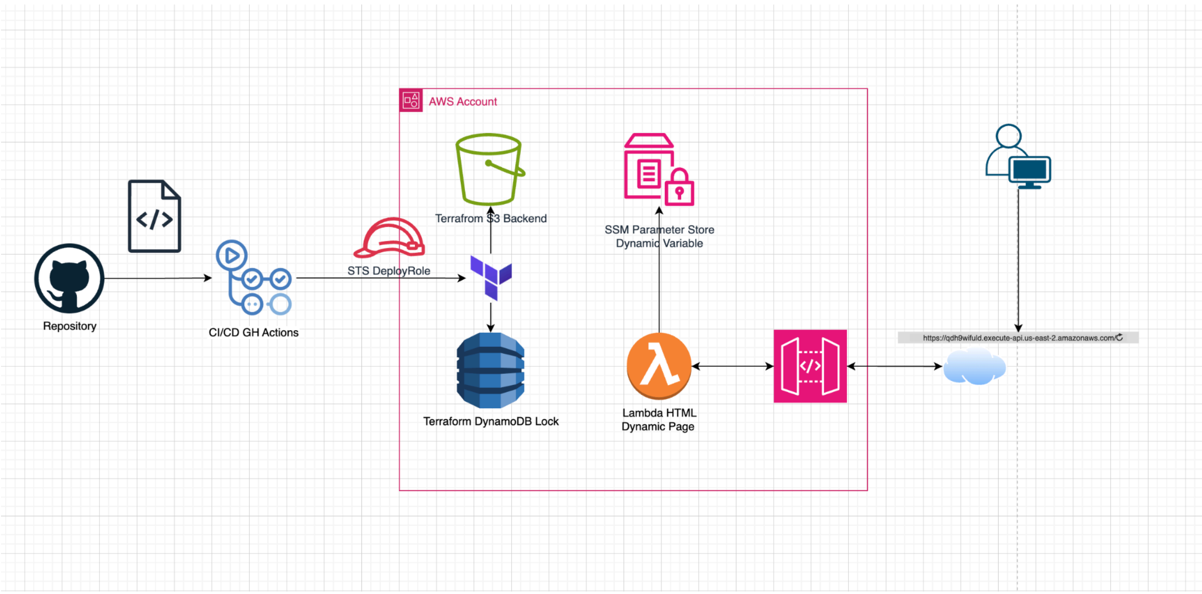
Live Demo URL (environments):

- **Development: https://qdh9wifuld.execute-api.us-east-2.amazonaws.com/**
- **Staging: https://0khbcrqddg.execute-api.us-east-2.amazonaws.com/**
- **Production: https://yzh305gxda.execute-api.us-east-2.amazonaws.com/**

2. Solution Architecture

2.1. High-Level Architecture

The solution implements a serverless architecture on AWS, promoting efficiency and scalability.



AWS Well-Architected Framework Alignment:

- **Performance Efficiency:** The use of managed, serverless services like Lambda and API Gateway allows the infrastructure to scale automatically based on demand, without manual provisioning.

- **Cost Optimization:** The pay-per-request model of the serverless components ensures that you only pay for the resources you consume, eliminating idle costs.

2.2. Core Components

1. **AWS Lambda Function:**
    - **Function:** `dynamic-html-lambda-{environment}`
    - **Purpose:** Serves dynamic HTML content with Bootstrap styling, executing code only when needed.
    - Python V3.11
2. **API Gateway v2 (HTTP API):**
    - **Purpose:** Provides a secure, scalable, and low-cost HTTP entry point for the Lambda function.
3. **AWS Systems Manager (SSM) Parameter Store:**
    - **Purpose:** Stores the dynamic content string, allowing for instant updates without redeploying the application.
4. **Infrastructure as Code (IaC):**
    - **Tool:** Terraform v1.12.0
    - **Purpose:** Automates the provisioning and management of infrastructure, ensuring consistency and repeatability.
5. **CI/CD Pipeline:**
    - **Platform:** GitHub Actions
    - **Purpose:** Automates the deployment of infrastructure changes in a secure and controlled manner.

3. Available Options Analysis

Each technology decision was made by comparing alternatives and selecting the most suitable one for the challenge's requirements, based on the pillars of the AWS Well-Architected Framework.

3.1. Compute Options

Option	Pros	Cons	Decision & Well-Architected Rationale
Lambda ✅(Chosed)	Serverless, pay-per-request, auto-scaling.	Cold start latency.	**Selected.** Aligns with **Cost Optimization** (pay-per-use) and **Performance Efficiency** (auto-scaling). Ideal for simple workloads.
ECS Fargate	More control, persistent connections.	More complex, higher cost.	Not selected. Overkill for this use case; would violate the principle of **Cost Optimization**.

3.2. Dynamic Content Storage Options

Option	Pros	Cons	Decision & Well-Architected Rationale

SSM Parameter Store ✅(Chosed)	Native integration, free tier, simple.	4KB limit per parameter.	**Selected.** Aligns with **Operational Excellence** (simplicity of management) and **Cost Optimization** (free tier). Perfect for simple strings.
DynamoDB	Highly scalable, flexible.	More complex, potential overhead.	Not selected. The additional complexity is not justified, which goes against operational simplicity.

3.3. API Gateway Options

Option	Pros	Cons	Decision & Well-Architected Rationale
API Gateway v2 (HTTP) ✅(Chosed)	70% cheaper, faster, simpler.	Fewer features than REST.	**Selected.** Aligns with **Cost Optimization** and **Performance Efficiency** due to its lower latency and reduced cost for simple use cases.
API Gateway (REST)	More features, WAF integration.	Higher cost, more complex.	Not selected. The extra features are unnecessary and would needlessly increase costs.

4. Technical Implementation Details

4.1. Lambda Function Architecture

The function's code is designed to be efficient and resilient, retrieving configurations from environment variables and handling errors gracefully.

Python
```
def lambda_handler(event, context):
  # Retrieve dynamic content from Parameter Store
  # ...
  # Return styled HTML with Bootstrap
  return styled_html_response(dynamic_string, environment_name)
```

4.2. Project and Infrastructure as Code (IaC) Structure

The repository structure is organized into folders and monorespository structure to facilitate maintainability and reusability.

```
├── .github
│   └── workflows
│       ├── auto-plan-on-changes.yaml
│       ├── deploy-development.yaml
│       ├── deploy-production.yaml
│       ├── deploy-staging.yaml
```

```
│       ├── deploy.yaml
│       └── pr-commands.yaml
├── code
│   └── backend
│       └── lambda
│           └── dynamic_html_service
│               └── lambda_function.py
└── infra
    └── aws
        ├── apigateway.tf
        ├── environments.tfvars
        ├── iam.tf
        ├── lambda
        ├── lambda.tf
        ├── outputs.tf
        ├── providers.tf
        ├── ssm.tf
        ├── variables.tf
        ├── versions.tf
        └── waf.tf
├── .gitignore
└── README.md
```

AWS Well-Architected Framework Alignment:

- **Operational Excellence:** IaC enables deployment automation, change management, and disaster recovery by treating infrastructure as code.

4.3. Multi-Environment Strategy

Terraform workspaces are used to isolate environments (development, staging, production), ensuring that changes are tested safely before reaching production.

AWS Well-Architected Framework Alignment:

- **Reliability:** Isolating environments reduces the risk of errors in development affecting production.
- **Operational Excellence:** Facilitates a controlled and predictable Software Development Life Cycle (SDLC).

4.4. Security Implementation

Security is a fundamental pillar of the design, implemented across multiple layers.

1. **Least Privilege (IAM):** The Lambda function only has the permissions strictly necessary to read the SSM parameter.
2. **OIDC Authentication:** The CI/CD pipeline uses OpenID Connect to authenticate with AWS, eliminating the need for long-lived credentials.
3. **State Encryption:** The Terraform state is encrypted at rest in S3.

4. **Rate Limiting (Throttling):** API Gateway is configured to limit requests and prevent denial-of-service (DoS) attacks.

AWS Well-Architected Framework Alignment:

- **Security:** All these measures directly apply recommended security principles, such as identity management, data protection, and infrastructure protection.

5. Decision Rationale (AWS Well-Architected Framework Alignment)

5.1. Why AWS Lambda?

It aligns with the **Cost Optimization**, **Performance Efficiency**, and **Operational Excellence** pillars by offering a pay-per-use model, automatic scaling, and zero server management.

5.2. Why API Gateway v2 (HTTP)?

It aligns with **Cost Optimization** and **Performance Efficiency** by being significantly cheaper and faster than the REST alternative for this project's requirements.

5.3. Why Parameter Store over a Database?

It aligns with **Operational Excellence** for its simplicity and **Cost Optimization** for its generous free tier. It eliminates the operational overhead of managing a database.

5.4. Why Terraform over CloudFormation?

It promotes **Operational Excellence** by being cloud-agnostic, having a more readable syntax (HCL), and a larger community, which facilitates infrastructure lifecycle management.

5.5. Why GitHub Actions over other CI/CD Tools?

It supports **Operational Excellence** through its native integration with the code repository and the **Security** pillar thanks to its use of OIDC for credential-less authentication.

6. Deployment and Usage

6.1. Automated Deployment

Deployment is managed via comments in GitHub Pull Requests, which centralizes and controls infrastructure changes.

Commands to deploy via a PR

- /terraform plan development
- /terraform apply development

AWS Well-Architected Framework Alignment:

- **Operational Excellence:** This automated, GitOps-based workflow is a best practice for making small, reversible changes frequently and safely.

6.2. Changing Dynamic Content

To update the content without redeploying, a simple AWS CLI command is used:

```bash
aws ssm put-parameter --name "/dynamic-html-service/dynamic-string-development" --value "New dynamic content" --overwrite
```

7. Future Enhancements

The current design serves as a solid foundation for future improvements, all aligned with the Well-Architected Framework.

Framework Pillar	Proposed Enhancement
Performance Efficiency	Add a **CloudFront (CDN)** distribution to reduce global latency.
Security	Enable **AWS WAF** via CloudFront to protect against common web attacks.
Operational Excellence	Create **CloudWatch dashboards** and **SNS alerts** for proactive monitoring.
Reliability	Implement a **multi-region** deployment with **Route 53** for high availability and disaster recovery.

8. Cost Analysis

The serverless design is extremely cost-effective, especially at a small scale.

Service	**Usage (1 Million Requests)**	**Estimated Monthly Cost**
Lambda	128MB, 100ms	$0.20
API Gateway v2	1M requests	$1.00
Total Estimated		**~$1.22 / month**

AWS Well-Architected Framework Alignment:

- **Cost Optimization:** The analysis and projections demonstrate a clear understanding of costs and how to scale efficiently, a key practice of this pillar.

9. Testing and Validation

Validation is crucial to ensure the quality and reliability of the solution.

AWS Well-Architected Framework Alignment:

- **Reliability:** Testing is fundamental to verify that the system works as expected and can withstand failures.
- **Operational Excellence:** An automated testing strategy allows for deploying with confidence and speed.

11. Conclusion

This solution demonstrates a modern and cost-effective application of AWS serverless services. The design, grounded in the principles of the **AWS Well-Architected Framework**, meets all challenge requirements by providing a secure, scalable, efficient, and code-managed infrastructure. The modular approach facilitates its maintenance and expansion, making it a robust model for both demonstration purposes and production deployment.

Appendix A: Repository Structure

```
dynamic-html-service-aws/
├── .DS_Store
├── .github
│   └── workflows
│       ├── auto-plan-on-changes.yaml
│       ├── deploy-development.yaml
│       ├── deploy-production.yaml
│       ├── deploy-staging.yaml
│       ├── deploy.yaml
│       └── pr-commands.yaml
├── .gitignore
├── README.md
├── code
│   └── backend
│       └── lambda
│           └── dynamic_html_service
│               └── lambda_function.py
└── infra
    └── aws
        ├── apigateway.tf
        ├── environments.tfvars
        ├── iam.tf
        ├── lambda
        ├── lambda.tf
        ├── outputs.tf
        ├── providers.tf
        ├── ssm.tf
        ├── variables.tf
        ├── versions.tf
        └── waf.tf
```

Appendix B: Key Commands

Deploy infrastructure

```
terraform init
terraform plan
terraform apply
```

Update dynamic content

```
aws ssm put-parameter \
  --name "/dynamic-html-service/dynamic-string-development" \
  --value "New content" --overwrite
```

View logs

```
aws logs filter-log-events \
  --log-group-name "/aws/lambda/dynamic-html-lambda-development"
```