Dynamic Text Solution - Work Document

Author: Carlos Andres Monserrat Rojas Rojas
Date: November 2025
Project: Dynamic Text Challenge

Challenge

Build a web application that displays dynamic text while satisfying two non-negotiable constraints:

RULE 1: Dynamic string can be set to whatever is requested without having to re-deploy
RULE 2: The URL should be the same no matter what the dynamic string is

---

Requirements Analysis

Explicit Requirements (What was stated)

- Content must be updatable without code deployment
- URL structure must remain constant regardless of content
- Working web application with user interface

Implicit Requirements (What was assumed)

The explicit rules drive several unstated but critical requirements:

Technical Constraints:

- External state management required (eliminates build-time configuration)
- Server-side content resolution needed (eliminates URL-based routing)
- Runtime configuration mechanism necessary

Quality Expectations:

- **Usability**: Users expect intuitive interfaces and quick responses
- **Reliability**: System should work consistently without frequent failures
- **Performance**: Pages should load quickly, updates should be reasonably fast
- **Security**: Admin functions should be appropriately protected in a real case scenario
- **Cost**: Solution should be economically viable for intended use

---

Solution Approaches Implemented

Two distinct approaches were implemented, plus one additional approach analyzed as a possible solution:

Approach 1: Static Files (S3 + CloudFront)

Philosophy: Minimize complexity and cost

```
User → CloudFront CDN → S3 Static Website
                                    ↑
Admin → AWS CLI ────────────────────┘
```

Key Characteristics:

- **Cost**: $0.10-$0.60/month*
  - S3 storage (1MB): $0.023/month
  - S3 requests (1K/month): $0.004/month
  - CloudFront data transfer (1GB): $0.085/month
  - CloudFront requests (10K): $0.0075/month
- **Update Method**: File upload to S3
- **Update Speed**: 1-5 minutes (cache dependent)
- **Complexity**: Minimal - just static files and CDN

*Based on AWS us-west-2 pricing, low traffic volume (1K pageviews/month)

Rule Compliance:

- ✅ RULE 1: Update `config.json` without redeploy
- ✅ RULE 2: Same HTML page serves all content variations

Approach 2: Server-Side Rendering (ECS + SSE)

Philosophy: Optimize for real-time updates and user experience

```
User → CloudFront CDN → Load Balancer → ECS Fargate Container
                                              ↓
                                      Real-time SSE Updates
                                              ↓
                                     All Connected Clients
```

Key Characteristics:

- **Cost**: $25-$35/month*
  - ECS Fargate (0.25 vCPU, 0.5GB): $7.00/month
  - Application Load Balancer: $16.20/month
  - CloudFront data transfer: $0.085/month
  - VPC NAT Gateway: $32.40/month (shared cost)
- **Update Method**: Admin interface with instant broadcast
- **Update Speed**: <100ms to all connected clients
- **Complexity**: Multi-service architecture with real-time features

*Based on AWS us-west-2 pricing, 24/7 uptime, moderate traffic

Rule Compliance:

- ✅ RULE 1: Update via API without container redeploy

- ✅ RULE 2: Server-side routing keeps URL constant

Approach 3: Serverless API (Lambda + API Gateway) - *Theoretical*

Philosophy: Balance cost and functionality **Status**: Not implemented, but analyzed as viable alternative

Estimated Characteristics:

- **Cost**: $0.50-$2.00/month*
  - Lambda execution (10K requests): $0.20/month
  - API Gateway (10K requests): $0.035/month
  - S3 static hosting: $0.023/month
  - CloudFront distribution: $0.085/month
- **Update Method**: Environment variable updates via AWS CLI
- **Update Speed**: 1-3 seconds (cold start dependent)
- **Complexity**: Moderate - serverless functions with static frontend

*Estimated based on AWS us-west-2 pricing, 10K requests/month, minimal compute time

---

Code Architecture Analysis

Astro SSR Solution: Feature-Driven Architecture

The Astro implementation demonstrates **feature-oriented design principles** with clear separation of concerns:

```
src/
├── pages/
│   ├── index.astro          # Main display page
│   ├── demo2.astro          # Polling demo page
│   ├── admin.astro          # Content management interface
│   └── api/
│       ├── events.ts        # SSE real-time endpoint
│       ├── text.ts          # JSON data endpoint
│       └── update.ts        # Content update endpoint
├── services/
│   └── sse-service.ts       # Business logic layer
└── data/
    └── text.json           # Data persistence
```

Architecture Principles Applied

SOLID Principles Implementation

Single Responsibility Principle (SRP)

```
// ✅ Each class has one clear responsibility
class SSEService {
```

```
  // Only handles SSE client lifecycle and broadcasting
  addClient(controller: ReadableStreamDefaultController): SSEClient
  async broadcastUpdate(data: string): Promise<void>
  getStats(): ConnectionStats
}

// ✅ API routes have single purpose
export const GET: APIRoute = async () => {
  // Only handles SSE connection setup
};

export const POST: APIRoute = async () => {
  // Only handles content updates
};
```

Open/Closed Principle (OCP)

```
// ✅ SSEService is open for extension, closed for modification
interface SSEClient {
  write: (chunk: Uint8Array) => Promise<void>;
  cleanup: () => void;
  id: string;
  connectedAt: Date;
}

// Can extend with new client types without modifying core service
class PrioritySSEClient implements SSEClient {
  priority: number;
  // ... extends base functionality
}
```

Dependency Inversion Principle (DIP)

```
// ✅ High-level modules depend on abstractions
import { sseService } from '@/services/sse-service.ts';

export const POST: APIRoute = async ({ request }) => {
  // Depends on service abstraction, not concrete implementation
  await sseService.broadcastUpdate(newText);
};
```

YAGNI (You Aren't Gonna Need It) Principle

What was NOT implemented (following YAGNI):

- ❌ User authentication system (not required for demo)
- ❌ Database layer (file storage sufficient)
- ❌ Complex configuration management (simple JSON works)

- ❌ Message queuing system (direct broadcast adequate)
- ❌ Logging framework (console.log sufficient for demo)

What was implemented (actual requirements):

- ✅ Content updating without redeploy (RULE 1)
- ✅ Consistent URL structure (RULE 2)
- ✅ Real-time updates (user experience requirement)
- ✅ Client connection management (technical necessity)

Screaming Architecture (Feature-Focused)

The directory structure "screams" its purpose:

```
// Clear feature boundaries
pages/api/events.ts      → "This handles real-time events"
pages/api/update.ts      → "This updates content"
pages/admin.astro        → "This is the admin interface"
services/sse-service.ts → "This manages SSE connections"
```

Business logic is prominent, framework details are hidden:

```
// Business intent is clear from the interface
interface SSEService {
  addClient(): SSEClient;           // "Add a new client"
  broadcastUpdate(): Promise<void>; // "Send update to everyone"
  getStats(): ConnectionStats;      // "Get connection info"
}
```

Code Complexity Comparison

Astro SSR Solution (Multi-file architecture)

Purpose: Demonstrates scalable patterns for real-world applications

```
// Separation of concerns across multiple files
// services/sse-service.ts - Business logic
export class SSEService {
  private clients = new Set<SSEClient>();

  async broadcastUpdate(data: string): Promise<void> {
    // Complex client management, error handling, cleanup
  }
}

// pages/api/events.ts - API interface
export const GET: APIRoute = async () => {
```

```
  const client = sseService.addClient(controller);
  // Clean integration with service layer
};

// pages/api/update.ts - Update endpoint
export const POST: APIRoute = async ({ request }) => {
  // Input validation, file operations, broadcasting
  await sseService.broadcastUpdate(text);
};
```

Benefits of this approach:

- ✅ **Testable**: Each module can be unit tested independently
- ✅ **Maintainable**: Changes isolated to specific concerns
- ✅ **Extensible**: New features don't require touching existing code
- ✅ **Reusable**: SSEService can be used by other endpoints

Simple Static Solution (Single-file approach)

Purpose: Demonstrates minimal viable implementation

```html
<!-- simplest-alternative/index.html - Everything in one file -->
<script>
  // All logic in a single script block
  async function fetchContent() {
    try {
      const response = await fetch(`config.json?v=${Date.now()}`);
      const data = await response.json();
      updateUI(data.dynamicString);
    } catch (error) {
      handleError(error);
    }
  }

  setInterval(fetchContent, 30000);
</script>
```

Benefits of this approach:

- ✅ **Simple**: Everything in one place, easy to understand
- ✅ **Fast to develop**: No build process, no dependencies
- ✅ **Minimal cognitive load**: No abstractions to understand
- ✅ **Self-contained**: Works without any framework

Design Patterns Demonstrated

Observer Pattern (SSE Implementation)

```
// SSEService acts as Subject
class SSEService {
  private clients = new Set<SSEClient>(); // Observers

  async broadcastUpdate(data: string) {
    // Notify all observers
    this.clients.forEach(client => client.write(encodedMessage));
  }
}
```

Factory Pattern (Client Creation)

```
addClient(controller: ReadableStreamDefaultController): SSEClient {
  // Factory method creates configured client instances
  const client: SSEClient = {
    id: this.generateClientId(),
    connectedAt: new Date(),
    write: async (chunk) => { /* configured behavior */ },
    cleanup: () => { /* configured cleanup */ }
  };
  return client;
}
```

Strategy Pattern (Update Methods)

```
// Different update strategies for different solutions
interface UpdateStrategy {
  update(content: string): Promise<void>;
}

class SSEUpdateStrategy implements UpdateStrategy {
  async update(content: string) {
    await sseService.broadcastUpdate(content);
  }
}

class FileUpdateStrategy implements UpdateStrategy {
  async update(content: string) {
    await fs.writeFileSync('config.json', JSON.stringify({content}));
  }
}
```

Trade-offs in Code Design

Complexity vs Maintainability

Astro Solution:

- **Higher complexity**: Multiple files, service abstractions, TypeScript interfaces
- **Better maintainability**: Clear separation allows easy modification and testing

Static Solution:

- **Lower complexity**: Single HTML file with inline JavaScript
- **Limited maintainability**: All changes require editing the main file

Performance vs Flexibility

Astro Solution:

- **More overhead**: Service instantiation, module loading, abstraction layers
- **Greater flexibility**: Easy to add features like authentication, rate limiting, analytics

Static Solution:

- **Minimal overhead**: Direct DOM manipulation, no framework costs
- **Limited flexibility**: Adding features requires architectural changes

Key Code Architecture Insights

1. **Feature-Driven Structure**: The Astro implementation organizes code by business features (events, updates, admin) rather than technical layers

2. **Appropriate Abstraction Level**: The SSEService provides just enough abstraction to be useful without over-engineering

3. **YAGNI Applied Correctly**: No premature optimization or unused features, but proper structure for planned growth

4. **Screaming Architecture**: You can understand what the application does just by looking at the file structure

5. **Single Purpose Principle**: Each file and class has one clear responsibility that maps to a business requirement

The code demonstrates that **good architecture scales appropriately** - using simple approaches where sufficient (static solution) and structured approaches where complexity is justified (SSR solution).

Technology Stack Decisions

Framework Choice: Astro.js

Selected for its **server-side rendering simplicity** and **minimal client-side JavaScript**:

```
---
// Server-side execution (satisfies RULE 1)
const data = JSON.parse(fs.readFileSync(dataPath, 'utf-8'));
```

```
const dynamicString = data.dynamicString || "default";
---
<!-- Client gets rendered content (satisfies RULE 2) -->
<h1>The saved string is {dynamicString}</h1>
```

Framework considerations:

Any framework that interacts with a server could solve this challenge.:

Javascript Frameworks:

- **Angular**: Excellent for complex data flows with RxJS observables. Overkill for this simple use case
- **React**: Great ecosystem and state management (Redux, Zustand). Client-side hydration adds unnecessary complexity here
- **Vue**: Clean syntax with Vue 3 + Pinia. Good for component-driven architectures
- **Next.js**: Built-in optimizations handle complexity well. More than needed for basic SSR
- **Express + EJS**: Straightforward server-side templating. Less overhead, more control

Backend Frameworks (MVC pattern):

- **Java**: Spring Boot with Thymeleaf templating
- **Python**: Django with template rendering
- **Scala**: Play Framework with Twirl templates

Astro chosen for personal preference - recently discovered this fast tool, component-based SSR felt natural for this challenge.

Real-Time Updates: Server-Sent Events

Chosen over WebSockets for simplicity:

- One-way communication (server → client) sufficient
- Automatic reconnection built-in
- HTTP-compatible (works through firewalls/proxies)
- Simpler server implementation

Infrastructure Implementation

CDK Stack Architecture

The project demonstrates **Infrastructure as Code** with multiple deployment targets:

```
// Multi-stack deployment approach
switch (deploymentTarget) {
  case 'ssr':    new SSRStack(app, 'SSRStack', { env });
  case 'simple': new SimpleStaticStack(app, 'SimpleStack', { env });
  case 'both':   /* Deploy both for comparison */
}
```

Service Architecture (SSR Solution)

Clean separation of concerns:

```
// Service Layer - Business Logic
export class SSEService {
  private clients = new Set<SSEClient>();

  addClient(controller: ReadableStreamDefaultController): SSEClient {
    // Manages client lifecycle and cleanup
  }

  async broadcastUpdate(data: string): Promise<void> {
    // Reliable delivery to all connected clients
  }
}

// API Layer - HTTP Interface
export const GET: APIRoute = async () => {
  const client = sseService.addClient(controller);
  // Handles SSE connection setup
};

// Storage Layer - Data Persistence
const data = JSON.parse(fs.readFileSync(dataPath, 'utf-8'));
```

Live Demonstrations

Two approaches are deployed and functional:

🏗 Full SSR Solution (Real-time)
URL: https://d1jk0h2l40omp5.cloudfront.net
Status: ✅ Implemented and deployed **Features**: Sub-second updates, admin interface, automatic reconnection

🪣 Simple Static Solution (Cost-optimized)
URL: https://simplest-alternative-944473419677-us-west-2.s3.us-west-2.amazonaws.com/index.html
Status: ✅ Implemented and deployed **Features**: Identical UI, file-based updates, minimal infrastructure

⚡ Serverless Solution (Theoretical)
Status: 📋 Analyzed but not implemented **Purpose**: Demonstrates middle-ground approach between static and full SSR

Quality Attributes Demonstrated

Performance Trade-offs

Approach	Initial Load	Update Speed	Concurrent Users	Monthly Cost	Status

Approach	Initial Load	Update Speed	Concurrent Users	Monthly Cost	Status
Static	<200ms	1-5 min	Unlimited	$0.50	✅ Deployed
SSR	<300ms	<100ms	500+	$35	✅ Deployed
Serverless	<400ms	1-3 sec	1000+	$2	🏷️ Theoretical

Cost vs Performance Analysis

50x cost difference between solutions meeting identical functional requirements:

```
Cost Comparison:
Static S3:        $0.50/month  [▏                    ]  1.4%
Serverless:       $2.00/month  [██                   ]  5.7%
SSR Container:   $35.00/month  [████████████████████] 100.0%
```

This demonstrates how **quality attribute priorities** dramatically impact implementation costs.

Security Considerations

Current Security Posture

Appropriate for a technical demonstration:

- ✅ **Public content model** eliminates most data protection concerns
- ✅ **HTTPS enforcement** via CloudFront
- ✅ **Infrastructure security** through VPC and security groups
- ⚠️ **Admin interface** has no authentication (acceptable for demo)

Production Security Requirements

What would need attention in real-world usage:

- Authentication for admin functions
- Input validation and sanitization
- Rate limiting on update endpoints
- Audit logging for content changes

Security implementation would depend on use case:

- **Internal tool**: Basic auth or SSO integration sufficient
- **Public service**: Multi-factor auth and comprehensive monitoring
- **Enterprise**: Full compliance framework (SOX, GDPR, etc.)

Improvements Discussion

"What could be improved with more time?"

The answer **depends entirely on the intended use case** for applications with these requirements:

Use Case 1: Digital Signage / Display Boards

Priority: Reliability and simple content management **Improvements**:

- Content scheduling (show different messages at different times)
- Multi-zone support (different content for different displays)
- Offline fallback content
- Simple CMS interface for non-technical users

Use Case 2: Emergency Notification System

Priority: Speed and reliability under stress **Improvements**:

- Multi-region deployment for disaster resilience
- Priority message queuing
- Multiple notification channels (SMS, email integration)
- Automatic failover mechanisms

Use Case 3: Marketing Campaign Landing Pages

Priority: Analytics and optimization **Improvements**:

- A/B testing framework for content variants
- Analytics integration (conversion tracking)
- Dynamic content based on user segments
- Performance optimization for mobile devices

Use Case 4: Internal Corporate Communication

Priority: Integration and workflow **Improvements**:

- Slack/Teams integration for content updates
- Approval workflow for content changes
- Integration with corporate SSO
- Audit trail for compliance

Use Case 5: IoT Device Configuration

Priority: Scale and device management **Improvements**:

- Device-specific content targeting
- Bulk update capabilities
- Device status monitoring
- Configuration versioning and rollback

Technical Improvements (Universal)

Regardless of use case, these would improve any implementation:

Code Quality:

- Comprehensive test suite (unit, integration, end-to-end)
- Input validation and error handling
- Structured logging and monitoring
- Documentation and API specifications

AWS Well-Architected Framework Improvements:

Operational Excellence:

- Automated deployment pipelines
- Health checks and alerting
- Backup and recovery procedures
- Performance monitoring dashboards

Security:

- Authentication for admin functions
- Input validation and sanitization
- Rate limiting on update endpoints
- Audit logging for content changes

Reliability:

- Multi-region deployment for disaster resilience
- Automatic failover mechanisms
- Circuit breakers for external dependencies
- Graceful degradation strategies

Performance Efficiency:

- CDN optimization for global delivery
- Caching strategies for static content
- Connection pooling for database access
- Resource right-sizing for cost efficiency

Cost Optimization:

- Reserved instance planning for predictable workloads
- Auto-scaling based on demand patterns
- Lifecycle policies for data retention
- Regular cost analysis and optimization reviews

Sustainability:

- Energy-efficient instance types
- Regional deployment in renewable energy zones
- Optimized resource utilization

Developer Experience:

- Local development environment setup
- Hot reload for faster iteration
- Clear contribution guidelines
- Automated code quality checks

Key Technical Insights

1. Simple Requirements, Complex Decisions

Two basic rules eliminated numerous architectural patterns and forced specific technical choices, demonstrating how **constraints drive innovation**.

2. Quality Attributes Matter More Than Features

The **50x cost difference** between functionally equivalent solutions shows that **non-functional requirements** often determine project success more than feature completeness.

3. Use Case Determines Optimization Strategy

There's no universally "best" solution - the optimal approach depends on:

- **Update frequency requirements** (real-time vs eventual consistency)
- **Budget constraints** (operational cost vs development time)
- **Operational complexity tolerance** (managed services vs custom solutions)
- **Scale expectations** (dozens vs thousands of concurrent users)

4. Evolution Paths Matter

Starting with the simplest solution that works provides **options for growth**:

- Static files → Serverless API → Container-based → Event-driven
- Each step adds capability at increased complexity and cost