SEVERLESS TEXT TO SPEECH DOCUMENTATION

https://tinyurl.com/ttscapstone

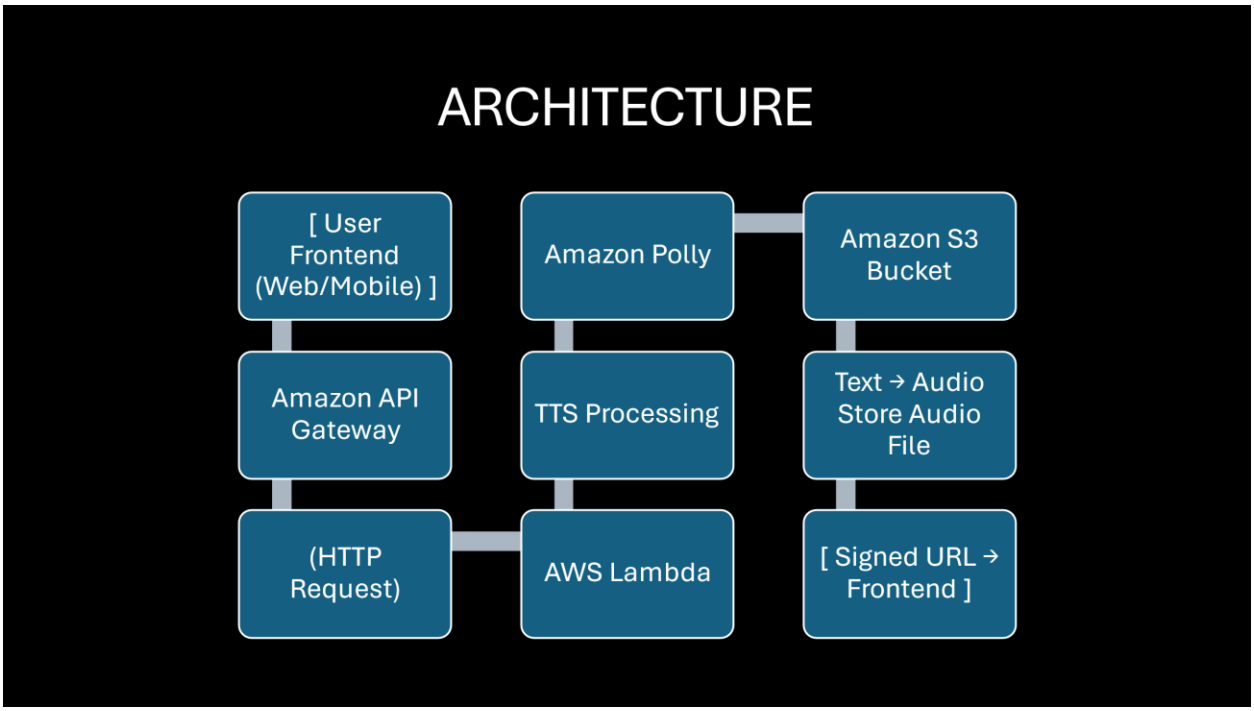
SUMMARY

This is a practical, end-to-end documentation on creating a Text to Speech App. It covers cloud choices for static hosting, serverless compute options, API gateway role, IAM, cloud storage, TTS options, CORS, event/response formats, error handling & logging — and includes a **Lambda function** that calls Amazon Polly, stores audio in S3, and returns a pre-signed URL.

Goals

- Provide a serverless function that accepts text, converts it to audio (TTS), stores audio in cloud storage, and returns a URL to the client.
- Use best practices: least-privilege IAM, CORS for browser clients, structured error responses, robust logging and monitoring, and an IaC template to create the infra.

ARCHITECTURE.



CHOICES & RATIONALE

Static hosting (for frontend)

- Options: Amazon S3 + CloudFront
- Reason: **S3 static hosting + CloudFront** — low cost, highly available, easy to connect with API Gateway, supports HTTPS via ACM.

Serverless compute

- Options: AWS Lambda
- Reason: **AWS Lambda** is fast to integrate with Polly & S3, automatic scaling, pay-per-invocation.

API gateway services & role

- API Gateway (REST API or HTTP API) is the endpoint that receives requests, performs validation, authentication/authorization, attachments of CORS headers, rate-limiting and proxies to Lambda.
- Reason: **API Gateway HTTP API v2** is lower cost and lower latency; for complex features, use REST APIs. Templates below use SAM AWS::Serverless::Function with an API event and CORS config.

Identity & Access Management (IAM)

- Used roles (Lambda execution role) granting only the needed privileges: polly:SynthesizeSpeech, s3:PutObject, s3:GetObject (if Lambda must read), s3:PutObjectAcl only if public objects required (avoid); logs:* is auto-handled for CloudWatch.
- Reason: Principle of least privilege, separate roles for different resources if complexity grows.

Text-to-speech service options

- AWS: **Amazon Polly** — multiple voices, languages, formats, SSML support.
- Reason: If you are on AWS already, **Polly** is easiest; otherwise pick the cloud-aligned TTS.

Cloud storage solutions

- For AWS: **S3** object store — simple, cheap, scalable. Use pre-signed URLs for secure direct client access. Considered lifecycle rules to expire old audio files.

CORS

- Configured at API Gateway (or via Lambda response headers) to allow browser clients to call the API. I allowed proper Origin, Content-Type, Authorization and support options preflight.

Monitoring & Logging

- CloudWatch Logs to monitor Lambda output

  he goal of Phase 2 is to design and implement the backend logic for a **serverless text-to-speech (TTS) application**. The system should:

1. Receive text input from clients through a secure API.
2. Process the text using a **serverless function** that calls a **text-to-speech service**.
3. Store the generated audio file in **cloud storage**.
4. Provide the client with a secure, time-limited **download URL** for the audio.
5. Ensure scalability, reliability, and security using **cloud-native services**.

Key Components & Services

1. **Static Hosting (for frontend)**
    - Hosted on **Amazon S3** with **CloudFront** for CDN delivery.
    - Serves HTML/JS frontend that communicates with the backend API.
2. **Serverless Compute (Backend)**
    - **AWS Lambda** executes the application logic on demand.
    - Eliminates server management and scales automatically with requests.
3. **API Gateway**
    - **Amazon API Gateway** provides a REST/HTTP API endpoint.
    - Handles authentication, request validation, and CORS configuration.
    - Directly integrates with Lambda functions.
4. **Text-to-Speech Service**
    - **Amazon Polly** converts text into lifelike speech in multiple languages/voices.
    - Lambda invokes Polly's SynthesizeSpeech API.
5. **Cloud Storage**
    - **Amazon S3** stores generated audio files.
    - Provides **pre-signed URLs** for secure, temporary client downloads.
    - Lifecycle policies manage storage costs by expiring old files.
6. **Identity and Access Management (IAM)**
    - **IAM roles and policies** control access between services.
    - Lambda execution role allows access only to Polly, S3, and CloudWatch.
    - Principle of least privilege ensures tight security.
7. **Monitoring and Logging**
    - **Amazon CloudWatch** captures logs, metrics, and alarms.
    - Provides insights into errors, latency, and usage trends.

BACKEND WORKFLOW

1. **Client Request**
    - The client (browser/app) sends a POST /synthesize request to API Gateway with JSON payload (text, format, optional voice).
2. **API Gateway Processing**
    - Validates request.
    - Applies CORS headers.
    - Forwards request to Lambda function.
3. **Lambda Function Execution**
    - Validates input parameters.
    - Invokes **Amazon Polly** to generate audio.
    - Stores the audio file in **S3** with a unique key.
    - Creates a **pre-signed S3 URL** for retrieval.
4. **Response Delivery**
    - Lambda returns a JSON response with:
        - Status (success/error)
        - S3 object key
        - Pre-signed URL
5. **Logging & Monitoring**
    - Errors, warnings, and execution details are logged in **CloudWatch Logs**.
    - Metrics such as execution time and error count tracked in **CloudWatch Metrics**.

Security Considerations

- **IAM Policies**: Lambda role limited to polly:SynthesizeSpeech and s3:PutObject/GetObject only for its specific bucket.
- **CORS**: API Gateway configured to allow only trusted frontend origins.
- **Pre-Signed URLs**: Time-limited links prevent unauthorized file sharing.
- **Encryption**: S3 bucket encryption enabled (SSE-S3).

- **Throttling**: API Gateway enforces request rate limits to prevent abuse.

Error Handling

- **Client Errors (4xx)**: Invalid inputs (missing text, unsupported format).
- **Server Errors (5xx)**: Polly/S3 failures or Lambda timeouts.
- **Structured Responses**: JSON responses include status, error code, and message.
- **Retries & Backoff**: Transient errors handled gracefully by AWS SDK retries.

Advantages of This Design

- **Scalable**: Serverless model auto-scales with usage.
- **Cost-Efficient**: Pay only for invocations, characters synthesized, and storage used.
- **Secure**: IAM, encryption, and pre-signed URLs protect resources.
- **Maintainable**: Infrastructure managed with Infrastructure as Code
- **Flexible**: Easily extendable to support more formats, languages, or event-driven workflows.

HOW THE FRONTEND & BACKEND WORK TOGETHER

1. User opens the **web app** (served via CloudFront).
2. Enters text, selects voice/format, and clicks **Convert**.
3. Browser sends a POST request to API Gateway.
4. API Gateway routes request to Lambda.
5. Lambda generates audio with Polly, stores it in S3, and returns a **pre-signed URL**.
6. Browser receives the URL and sets it as the source of the <audio> player.
7. User clicks **Play** to hear the generated speech.

BENEFITS OF THIS FRONTEND + BACKEND SETUP

- **Serverless end-to-end**: No servers to manage.
- **Scalable**: CloudFront + Lambda handle global traffic.
- **Low latency**: CloudFront caches static assets close to users.
- **Secure**: IAM restricts backend permissions, pre-signed URLs protect audio files.
- **User-friendly**: Simple web app with audio playback.

10 — Scaling & cost considerations

- Lambda scales automatically up to regional concurrency limit (tune account limits as needed).
- Polly charges per character; S3 charges for storage & GET requests; CloudWatch for logs/metrics. Estimate costs with your expected traffic.