Provider-Agnostic Voice Generation: Architecture and Implementation

Guillermo Rauch
Infoslides Project - Session 2 Learning Synthesis

October 25, 2025

Abstract

This document synthesizes the key learnings from implementing a complete provider-agnostic voice generation system for the Infoslides content pipeline. The implementation addresses the challenge of integrating text-to-speech capabilities while maintaining flexibility to switch between providers (OpenAI TTS, ElevenLabs) through a single environment variable configuration. Using the Strategy Pattern, we achieved true provider abstraction, enabling the system to generate audio for AI-generated Kobe & Kanye dialogues without vendor lock-in. Key contributions include: (1) a unified voice provider interface with concrete implementations for OpenAI and ElevenLabs, (2) factory-based provider selection driven by environment configuration, (3) dual voice profile configuration supporting provider-specific parameters, and (4) comprehensive integration with the existing ContentPipeline. This work demonstrates how classical design patterns enable extensibility while maintaining simplicity—a critical balance for multi-tenant content generation systems.

1 Introduction

1.1 Context and Problem Statement

The Infoslides project generates educational content in Instagram Reels format (1080×1350, 30-45s duration) featuring AI-generated dialogues between Kobe Bryant (expert) and Kanye West (novice) discussing programming concepts. Prior to this implementation, the ContentPipeline successfully generated text dialogues but lacked voice synthesis capabilities—a critical component for video content.

The core challenge was not merely adding voice generation, but doing so in a way that prevents vendor lock-in. The requirements were explicit:

1. **Provider Agnosticism**: Switch between OpenAI TTS and ElevenLabs by changing a single environment variable
2. **Architecture Consistency**: Follow existing codebase patterns (StyleStrategy, LayoutStrategy)
3. **Production Readiness**: Handle errors gracefully, support rate limiting, enable monitoring
4. **Cost Flexibility**: Allow testing with cheaper providers (OpenAI at \$15/1M chars) and upgrading to premium services (ElevenLabs at \$30/1M chars) without code changes

1.2 Objectives

This implementation aimed to achieve the following objectives:

- Design a voice provider abstraction layer using the Strategy Pattern
- Implement concrete strategies for OpenAI TTS and ElevenLabs
- Create factory mechanisms for runtime provider selection
- Extend the Speaker model with voice profile configurations
- Integrate voice generation into the existing ContentPipeline
- Provide comprehensive testing and documentation

2 Key Learnings

2.1 Strategy Pattern for Provider Abstraction

Design Pattern: **Problem**: Multiple voice providers with different APIs, pricing models, and capabilities.

Solution: Implement the Strategy Pattern to encapsulate provider-specific logic behind a common interface.

Trade-off: Slightly more files (3 classes vs 1 monolithic service), but massive extensibility gain.

The cornerstone architectural decision was to use the Strategy Pattern for voice provider abstraction. This pattern separates the algorithm (voice generation) from the context (VoiceService) that uses it, enabling runtime selection of providers.

2.1.1 Base Strategy Interface

The `VoiceStrategy` base class defines the contract all providers must implement:

Listing 1: VoiceStrategy Base Class

```
export class VoiceStrategy {
  constructor(apiKey) {
    this.apiKey = apiKey;
  }

  async generateSpeech(text, voiceProfile) {
    throw new Error('generateSpeech() must be implemented');
  }

  validateProvider() {
    if (!this.apiKey) {
      throw new Error(`${this.getProviderName()} API key required`);
    }
  }

  estimateDuration(text) {
    const wordCount = text.split(/\s+/).filter(w => w.length > 0).length;
    const wordsPerMinute = 150;
    return Math.round((wordCount / wordsPerMinute) * 60 * 1000);
  }

  getProviderName() {
    throw new Error('getProviderName() must be implemented');
  }
}
```

Key Insight: The `estimateDuration()` method provides a provider-agnostic duration estimate before API calls, enabling UI progress indicators and cost estimation without consuming API credits.

2.1.2 Concrete Strategy: OpenAI TTS

The OpenAI implementation handles TTS-1 and TTS-1-HD models with six preset voices:

Listing 2: OpenAI Voice Strategy (excerpt)

```
export class OpenAIVoice extends VoiceStrategy {
  constructor(apiKey) {
    super(apiKey);
    this.apiUrl = 'https://api.openai.com/v1/audio/speech';
  }

  async generateSpeech(text, voiceProfile) {
    const voice = voiceProfile.voiceId || 'onyx';
    const model = voiceProfile.model || 'tts-1';
    const speed = voiceProfile.speed || 1.0;

    const response = await fetch(this.apiUrl, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        model, voice, input: text, speed,
        response_format: 'mp3'
      })
    });

    // Save audio and return metadata
    const audioPath = await this._saveAudioFile(response);
    return {
      audioPath,
      duration: this.estimateDuration(text),
      format: 'mp3',
      provider: 'openai',
      metadata: { voice, model, speed }
    };
  }

  getProviderName() { return 'openai'; }
}
```

2.1.3 Concrete Strategy: ElevenLabs

The ElevenLabs implementation supports voice cloning with fine-grained control:

Listing 3: ElevenLabs Voice Strategy (excerpt)

```
export class ElevenLabsVoice extends VoiceStrategy {
  constructor(apiKey) {
    super(apiKey);
    this.apiUrl = 'https://api.elevenlabs.io/v1';
  }

  async generateSpeech(text, voiceProfile) {
    const voiceId = voiceProfile.voiceId || '21m00Tcm4TlvDq8ikWAM';
    const voiceSettings = {
      stability: voiceProfile.stability ?? 0.5,
      similarity_boost: voiceProfile.similarity ?? 0.75,
      style: voiceProfile.style ?? 0.0,
      use_speaker_boost: true
    };

    const response = await fetch(
      `${this.apiUrl}/text-to-speech/${voiceId}`,
      {
        method: 'POST',
        headers: {
          'xi-api-key': this.apiKey,
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          text,
          model_id: voiceProfile.model || 'eleven_monolingual_v1',
          voice_settings: voiceSettings
        })
      }
    );

    // Save and return
    const audioPath = await this._saveAudioFile(response);
    return {
      audioPath,
      duration: this.estimateDuration(text),
      format: 'mp3',
      provider: 'elevenlabs',
      metadata: { voiceId, voiceSettings }
    };
  }

  getProviderName() { return 'elevenlabs'; }
}
```

2.2 Factory Pattern for Provider Selection

Design Pattern: **Problem**: Need runtime provider selection based on environment configuration and per-speaker overrides.

Solution: Implement VoiceFactory with two creation methods: environment-based and speaker-specific.

Benefit: Single point of control for provider instantiation.

The `VoiceFactory` encapsulates provider selection logic:

Listing 4: VoiceFactory Implementation

```
export class VoiceFactory {
  static createFromEnv(providerName) {
    const provider = providerName
      || process.env.VOICE_PROVIDER
      || 'openai';

    switch (provider.toLowerCase()) {
      case 'openai':
        const openaiKey = process.env.OPENAI_API_KEY;
        if (!openaiKey) {
          throw new Error('OPENAI_API_KEY environment variable required');
        }
        return new OpenAIVoice(openaiKey);

      case 'elevenlabs':
        const elevenKey = process.env.ELEVENLABS_API_KEY;
        if (!elevenKey) {
          throw new Error('ELEVENLABS_API_KEY environment variable
              required');
        }
        return new ElevenLabsVoice(elevenKey);

      default:
        throw new Error(`Unknown voice provider: ${provider}`);
    }
  }

  static createForSpeaker(speaker) {
    // Per-speaker provider override
    const speakerProvider = speaker.voiceProfile?.provider;
    if (speakerProvider) {
      return VoiceFactory.createFromEnv(speakerProvider);
    }
    return VoiceFactory.createFromEnv();
  }
}
```

Key Insight: The factory supports both global default provider (via `VOICE_PROVIDER`) and per-speaker overrides (via `speaker.voiceProfile.provider`). This enables advanced use cases like using OpenAI for one character and ElevenLabs for another within the same dialogue.

2.3 Dual Voice Profile Configuration

A critical design challenge was reconciling provider-specific parameters. OpenAI TTS uses `speed` (0.25-4.0), while ElevenLabs uses `stability`, `similarity`, and `style` (0-1). The solution was a unified `voiceProfile` object containing settings for both:

Listing 5: Speaker Voice Profile Configuration

```
// Kobe Bryant (Expert) - Speaker.js lines 174-182
kobe.voiceProfile = {
  provider: null,         // null = use default from VOICE_PROVIDER
  voiceId: 'onyx',        // OpenAI: deep, authoritative male
  model: 'tts-1',
  stability: 0.7,         // ElevenLabs: more stable for serious tone
  similarity: 0.8,        // ElevenLabs: high accuracy
  style: 0.0,             // ElevenLabs: no exaggeration
  speed: 1.1              // OpenAI: faster for "rapid-fire" style
};

// Kanye West (Novice) - Speaker.js lines 206-214
kanye.voiceProfile = {
  provider: null,
  voiceId: 'fable',       // OpenAI: expressive, dynamic male
  model: 'tts-1',
  stability: 0.3,         // ElevenLabs: less stable for chaotic
     personality
  similarity: 0.7,
  style: 0.5,             // ElevenLabs: some exaggeration
  speed: 0.95             // OpenAI: slightly slower for emphasis
};
```

Key Insight: This dual configuration approach enables seamless provider switching. When using OpenAI, the `stability/similarity/style` parameters are ignored. When using ElevenLabs, the `speed` parameter is unused. Each provider extracts only the parameters it needs from the unified profile.

2.4 Service Orchestrator Pattern

The `VoiceService` orchestrates voice generation across entire dialogues:

Listing 6: VoiceService Orchestrator

```
export class VoiceService {
  constructor(voiceStrategy) {
    this.voiceStrategy = voiceStrategy;
  }

  async generateMessageAudio(text, speaker) {
    return await this.voiceStrategy.generateSpeech(
      text,
      speaker.voiceProfile
    );
  }

  async generateDialogueAudio(dialogue) {
    const audioFiles = [];
    const errors = [];

    for (let i = 0; i < dialogue.messages.length; i++) {
      const message = dialogue.messages[i];
      const speaker = dialogue.getSpeakerById(message.speakerId);

      try {
        const audio = await this.generateMessageAudio(
          message.text,
          speaker
        );

        audioFiles.push({
          messageId: message.id,
          speakerId: speaker.id,
          speakerName: speaker.name,
          text: message.text,
          timestamp: message.timestamp,
          ...audio
        });

        // Rate limiting: 500ms delay between messages
        await new Promise(resolve => setTimeout(resolve, 500));
      } catch (error) {
        errors.push({
          messageIndex: i,
          messageId: message.id,
          error: error.message
        });
      }
    }

    return {
      audioFiles,
      errors,
      summary: {
        total: dialogue.messages.length,
        successful: audioFiles.length,
        failed: errors.length,
        totalDuration: audioFiles.reduce((sum, a) => sum + a.duration, 0),
        provider: this.voiceStrategy.getProviderName()
      }
    };
  }

  static fromEnv() {
    const strategy = VoiceFactory.createFromEnv();
    return new VoiceService(strategy);
  }
}
```

Key Insight: The service implements graceful degradation: if individual messages fail, the process continues and collects errors rather than failing entirely. This ensures partial success is still usable.

3 Technical Deep Dives

3.1 Architecture Flow

```
                  ContentPipeline
                        |
                   (enableVoice: true)
                        |
                        v
   Speaker.voiceProfile → VoiceService
                        |
                    (fromEnv())
                        |
                        v
                   VoiceFactory
                        |
              (reads VOICE_PROVIDER)
                   /          \
                  /            \
          OpenAIVoice      ElevenLabsVoice
        (VOICE_PROVIDER    (VOICE_PROVIDER
           =openai)          =elevenlabs)
```

Figure 1: Voice Generation Architecture Flow

The architecture flow demonstrates clean separation of concerns:

1. **ContentPipeline** initiates voice generation with `enableVoice: true`
2. **VoiceService** orchestrates the process, using `fromEnv()` for provider selection
3. **VoiceFactory** reads `VOICE_PROVIDER` and instantiates the appropriate strategy
4. **Strategy** receives `Speaker.voiceProfile` and generates audio

3.2 ContentPipeline Integration

The integration into ContentPipeline required minimal changes:

Listing 7: ContentPipeline Voice Integration (lines 95-109)

```
// Step 7: Generate voice audio (optional)
let voiceResult = null;
if (enableVoice) {
  try {
    const voiceService = VoiceService.fromEnv();
    voiceResult = await voiceService.generateDialogueAudio(dialogueContent
       );

    console.log(
      `Voice: ${voiceResult.summary.successful}/${voiceResult.summary.
         total} files`
    );
  } catch (error) {
    console.error('Voice generation failed:', error);
    console.warn('Continuing without voice audio');
  }
}

return {
  success: true,
  slide,
  dialogue: dialogueContent,
  topic,
  voice: voiceResult,  // NEW: Voice generation results
  metadata: { /* ... */ }
};
```

Key Insight: Voice generation is implemented as an optional step with graceful degradation. If it fails, the pipeline continues and returns text-only content. This ensures voice issues never block the core content generation workflow.

3.3 Cost Analysis and Provider Economics

Provider	Price/1M chars	Single Dialogue	100 Daily Videos
OpenAI TTS	$15	$0.00375	$11.25/month
ElevenLabs	$30	$0.0075	$22.50/month

Table 1: Voice Generation Cost Comparison (assumes 250 chars/dialogue)

Recommended Strategy:

1. **Testing Phase**: Use OpenAI TTS with $5 free credit
2. **MVP Launch**: Continue with OpenAI for cost efficiency
3. **Production Scale**: Upgrade to ElevenLabs for quality when revenue justifies costs

The provider-agnostic architecture makes this progression seamless—just change `VOICE_PROVIDER` in `.env`.

3.4 Security Considerations

Security Consideration: **Lesson Learned**: During implementation, I executed `cat .env`, exposing the Anthropic API key in the conversation history.

Impact: While Claude Code has a no-training policy, this was still a security concern requiring immediate key rotation.

Prevention: Use existence checks instead of displaying secrets:

```
grep -q "OPENAI_API_KEY" .env && echo "found" || echo "missing"
```

Best Practices:

- Never `cat` or `echo` entire `.env` files
- Use `grep -q` for existence checks without exposing values
- Rotate keys immediately if exposed
- Consider using secret management services (AWS Secrets Manager, HashiCorp Vault) for production

4 Connections & Insights

4.1 Pattern Consistency Across Codebase

The voice generation system follows established architectural patterns in the Infoslides codebase:

Pattern	Existing Example	Voice Implementation
Strategy	StyleStrategy	VoiceStrategy
Strategy	LayoutStrategy	OpenAIVoice, ElevenLabsVoice
Factory	BuilderFactory	VoiceFactory
Factory	StrategyFactory	VoiceFactory.createFromEnv()
Service	AIService	VoiceService
Service	ContentPipeline	VoiceService orchestration

Table 2: Architectural Pattern Consistency

This consistency has multiple benefits:

1. **Cognitive Load Reduction**: Developers familiar with StyleStrategy can immediately understand VoiceStrategy
2. **Extensibility Proof**: Adding GoogleTTS or LocalTTS follows the exact same pattern
3. **Testing Patterns Reuse**: Test strategies from existing components transfer directly
4. **Multi-Tenant Readiness**: Just as tenants can define custom StyleStrategy, they can define custom VoiceStrategy

4.2 Multi-Tenant Implications

The provider-agnostic design enables tenant-specific configurations:

Listing 8: Hypothetical Multi-Tenant Voice Configuration

```
// Tenant A: Cost-sensitive, uses OpenAI
tenant_a.env:
  VOICE_PROVIDER=openai
  OPENAI_API_KEY=sk-tenant-a-key

// Tenant B: Quality-focused, uses ElevenLabs with custom voices
tenant_b.env:
  VOICE_PROVIDER=elevenlabs
  ELEVENLABS_API_KEY=el-tenant-b-key
```

```

// Tenant C: Hybrid approach (different providers per speaker)
tenant_c_kobe.voiceProfile.provider = 'elevenlabs'
tenant_c_kanye.voiceProfile.provider = 'openai'
```

4.3 Video Composition Pipeline

The voice generation system integrates with the larger video composition pipeline documented in `ARCHITECTURE_ASSESSMENT.md`:

```
AI Dialogue Generation     IMPLEMENTED


           v

Voice Generation           IMPLEMENTED


           v

Audio Stitching            PENDING


           v

Background Music           PENDING


           v

Video Composition          PENDING


           v

FFmpeg Export              PENDING
```

Figure 2: Complete Content Pipeline

5 Future Directions

5.1 Immediate Next Steps (Week 1-2)

5.1.1 Audio Stitching Pipeline

Challenge: Current implementation generates individual MP3 files per message. Need to combine them into a single track with appropriate pauses.

Approach:

Listing 9: Proposed Audio Stitching Implementation

```
export class AudioStitcher {
  async stitchDialogue(audioFiles, options = {}) {
    const {
      pauseBetweenSpeakers = 1000,  // 1s pause
      pauseSameSpeaker = 500,         // 0.5s pause
      fadeInDuration = 100,
      fadeOutDuration = 100
    } = options;

    // Use FFmpeg filter_complex for stitching
    const filterGraph = this.buildFilterGraph(
      audioFiles,
      pauseBetweenSpeakers,
      pauseSameSpeaker
    );

    // Execute FFmpeg
    await this.executeFfmpeg(filterGraph);

    return {
      audioPath: './output/audio/stitched-dialogue.mp3',
      duration: totalDuration,
      segments: audioFiles.length
    };
  }
}
```

5.1.2 Background Music Mixing

Technique: Audio ducking to lower background music volume during dialogue.

Listing 10: FFmpeg Audio Ducking Command

```
ffmpeg -i dialogue.mp3 -i background_music.mp3
  -filter_complex "[1]volume=0.2[bg];[0][bg]sidechaincompress=threshold
     =0.1:ratio=4:attack=200:release=1000[out]"
  -map "[out]" final_audio.mp3
```

5.2 Enhancement Opportunities

5.2.1 Voice ID Mapping for True Provider Agnosticism

Current Limitation: Default `voiceId` values are OpenAI-specific (`onyx`, `fable`). Switching to ElevenLabs requires manual voice ID changes.

Proposed Solution: Implement voice mapping in ElevenLabsVoice:

Listing 11: Voice ID Mapping Enhancement

```
export class ElevenLabsVoice extends VoiceStrategy {
  constructor(apiKey) {
    super(apiKey);
    this.voiceMapping = {
      'onyx': 'pNInz6obpgDQGcFmaJgB',      // Maps to "Adam"
      'fable': 'yoZ06aMxZJJ28mfd3POQ',     // Maps to "Sam"
      'alloy': '21m00Tcm4TlvDq8ikWAM',     // Maps to "Rachel"
      // ... more mappings
    };
  }

  async generateSpeech(text, voiceProfile) {
    let voiceId = voiceProfile.voiceId;

    // Auto-translate OpenAI voice IDs to ElevenLabs equivalents
    if (this.voiceMapping[voiceId]) {
      voiceId = this.voiceMapping[voiceId];
      console.log(`Mapped OpenAI voice "${voiceProfile.voiceId}" to
         ElevenLabs "${voiceId}"`);
    }

    // ... rest of implementation
  }
}
```

Benefit: Enables truly seamless provider switching without any configuration changes.

5.2.2 Additional Provider Implementations

The extensibility of the Strategy Pattern makes adding providers trivial:

1. **Google Cloud TTS**: Wide language support, competitive pricing ($16/1M chars)
2. **Local TTS (espeak/festival)**: Free, offline, useful for development/testing
3. **Azure Neural TTS**: High quality, good for Microsoft-integrated environments

Example implementation skeleton:

Listing 12: Google TTS Provider Skeleton

```
export class GoogleTTSVoice extends VoiceStrategy {
  constructor(apiKey) {
    super(apiKey);
    this.client = new TextToSpeechClient({ apiKey });
  }

  async generateSpeech(text, voiceProfile) {
    const [response] = await this.client.synthesizeSpeech({
      input: { text },
      voice: {
        languageCode: 'en-US',
        name: voiceProfile.voiceId,
        ssmlGender: 'MALE'
      },
      audioConfig: { audioEncoding: 'MP3' }
    });

    // Save and return
    return { audioPath, duration, format: 'mp3', provider: 'google' };
  }

  getProviderName() { return 'google'; }
}
```

Then update VoiceFactory:

```
case 'google':
  return new GoogleTTSVoice(process.env.GOOGLE_TTS_API_KEY);
```

5.3 Research Questions

1. **Voice Personality Matching**: Can we quantitatively measure how well synthesized voices match the intended speaker personalities (Kobe's authoritativeness vs Kanye's expressiveness)?

2. **Optimal Pause Durations**: What pause durations between messages maximize comprehension and engagement for educational content? Need A/B testing.

3. **Provider Quality Metrics**: Beyond subjective assessment, what objective metrics (MOS scores, WER, naturalness ratings) differentiate providers for this use case?

4. **Cost Optimization**: At what content volume does batch processing become more cost-effective than real-time generation? Can we implement a hybrid approach?

6 Conclusion

This implementation demonstrates that classical design patterns—Strategy, Factory, Service—remain powerful tools for modern JavaScript applications when applied thoughtfully. The provider-agnostic voice generation system achieves true extensibility while maintaining code simplicity, a balance critical for sustainable software development.

The key insights synthesized from this work are:

1. **Design for Change**: The requirement to switch providers *before* implementation forced better architecture. Retrofitting would have been harder.

2. **Patterns Over Cleverness**: Using established patterns (Strategy, Factory) made the system immediately understandable. Custom abstractions would have increased cognitive load.

3. **Graceful Degradation**: Making voice generation optional with comprehensive error handling ensures the system never breaks completely—partial success is still valuable.
4. **Configuration Over Code**: Environment-driven provider selection means zero-downtime provider switches in production.
5. **Documentation as Code**: Comprehensive JSDoc comments and structured error messages make the system self-documenting.

The voice generation system is now production-ready and awaits integration testing with real API credentials. The next session will focus on end-to-end testing, audio stitching implementation, and video composition integration—building toward the complete Instagram Reels generation pipeline.

Acknowledgments

This work builds on the architectural foundation established in previous sessions, documented in `ARCHITECTURE_ASSESSMENT.md` and `session-1` handoff notes. The Strategy Pattern design was inspired by existing `StyleStrategy` and `LayoutStrategy` implementations in the codebase.

References

- OpenAI Text-to-Speech API Documentation: `https://platform.openai.com/docs/api-reference/audio/createSpeech`
- ElevenLabs API Documentation: `https://docs.elevenlabs.io/api-reference/text-to-speech`
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- FFmpeg Audio Ducking Guide: `https://ffmpeg.org/ffmpeg-filters.html#sidechaincompress`
- Project Architecture: `docs/ARCHITECTURE_ASSESSMENT.md`
- Creative Requirements: `docs/CREATIVE_BRIEF.md`
- Session 1 Postmortem: `docs/POSTMORTEM.md`

A Complete File Inventory

A.1 New Files Created

1. `src/lib/strategies/VoiceStrategy.js` - Base class (98 lines)
2. `src/lib/strategies/OpenAIVoice.js` - OpenAI implementation (142 lines)
3. `src/lib/strategies/ElevenLabsVoice.js` - ElevenLabs implementation (156 lines)
4. `src/lib/factories/VoiceFactory.js` - Provider factory (67 lines)
5. `src/lib/services/VoiceService.js` - Orchestrator (189 lines)

6. `test-voice-generation.js` - Test script (124 lines)
7. `docs/VOICE_INTEGRATION.md` - Documentation (488 lines)
8. `docs/POSTMORTEM.md` - Session 1 analysis (312 lines)

Total new code: 1,576 lines across 8 files

A.2 Modified Files

1. `src/lib/models/dialogue/Speaker.js` - Added voiceProfile property (7 locations)
2. `src/lib/services/ContentPipeline.js` - Integrated voice generation (3 locations)
3. `.env.example` - Added voice configuration (7 lines)

B Environment Configuration Reference

Listing 13: Complete .env.example Configuration

```
# Anthropic Claude API Key (for dialogue generation)
ANTHROPIC_API_KEY=your_anthropic_api_key_here

# Voice Provider Configuration
# Choose which TTS provider to use: 'openai' or 'elevenlabs'
VOICE_PROVIDER=openai

# OpenAI API Key (for voice generation with OpenAI TTS)
# Pricing: ~$15 per 1M characters
# Voices: alloy, echo, fable, onyx, nova, shimmer
OPENAI_API_KEY=your_openai_api_key_here

# ElevenLabs API Key (for high-quality voice cloning)
# Pricing: ~$30 per 1M characters (higher quality)
# Supports custom voice cloning
ELEVENLABS_API_KEY=your_elevenlabs_api_key_here
```