ChatGPT

"Kairos" Voice-Controlled Presentation System Documentation

This document is divided into two main parts: a **Functional Requirements Document** outlining what the "Kairos" system must accomplish, and a **Technical Architecture Document** detailing how the system will be designed and implemented. The "Kairos" system is a real-time, voice-activated agent that listens to live audio (speech and music) and controls ProPresenter 7 presentation software accordingly.

Functional Requirements Document

1. Mission and Overview

The mission of the "Kairos" initiative is to create an intelligent, autonomous system that acts as a bridge between a live audio environment (e.g. a stage or church service) and ProPresenter 7 (presentation software). **Kairos** will continuously monitor a live audio feed, understand spoken commands or identify music being played, and trigger the appropriate actions in ProPresenter (such as displaying lyrics or scripture slides) in real time. The system should provide a seamless, voice-driven experience for controlling media and lyrics during live events.

Scope: The system covers end-to-end voice control of presentation content. This includes audio capture, speech recognition, intent parsing, music recognition, and sending commands to ProPresenter. The scope also includes tools for managing content (like songs or scripture slides) and ensuring the system can improve over time via custom model training and human feedback.

Key Stakeholders and Users: The primary users are live production operators (e.g., church service media teams) who will interact with the system by speaking commands or overseeing its automated actions. A human operator also maintains the content library and verifies system suggestions when needed. The system itself (AI agent) carries out most tasks autonomously after initial setup.

2. Actors and Responsibilities

The **Kairos system** involves collaboration between a human operator and the AI agent. The division of responsibilities is as follows:

- **Human Operator Roles:**
- *Content Curator:* Upload and manage media content (song audio files, associated lyric slides, scripture slides) via a web portal. Ensure that for each song or scripture to be voice-accessed, a corresponding presentation exists in ProPresenter's library.
- *System Trainer:* Provide initial training data (audio recordings and transcripts of typical speech commands, unique terms, etc.) to help fine-tune the speech recognition model for the specific environment (accent, acoustic conditions, terminology).

- *Live Supervisor (Verifier):* Monitor the system during operation. When **Kairos** flags uncertain interpretations (low-confidence transcriptions or ambiguous intent), the operator reviews and corrects them through a verification UI. These corrections feed back into improving the system over time.

- **AI System (Kairos) Roles:**

- *Real-Time Orchestrator:* Autonomously handle live audio input, distinguish speech from music, transcribe speech, understand the intent, and execute the appropriate ProPresenter command without human intervention (for high-confidence events).
- *Content Processor:* When new media is uploaded, automatically process it (e.g. extract audio from videos, generate audio fingerprints for music recognition) and update internal databases (custom song fingerprint library, etc.).
- *Model Trainer:* Continuously learn from new data. Fine-tune speech recognition models using the operator-provided corpus and update vocabulary (e.g. new song titles or names) so voice commands remain accurate.
- *Presentation Controller:* Interface with ProPresenter through its APIs to carry out commands (like navigating slides or searching for specific content by title).

3. Functional Requirements

The **Kairos** system must fulfill the following functional requirements (FR):

- **FR-1.1 – Audio Stream Triage:** The system **must** continuously listen to a live audio stream and **detect when meaningful audio occurs**. It should differentiate between spoken words (speech) and music in real time. This ensures that music passages don't get sent to the speech recognizer and spoken commands don't go to the music identifier by mistake.

- **FR-1.2 – Speech Transcription:** When speech is detected, the system **must accurately transcribe spoken commands** into text. It should leverage custom-trained speech-to-text models tailored to the environment (incorporating domain-specific vocabulary like song names, scripture references, or terms like "ProPresenter" and others).

- **FR-1.3 – Intent Recognition:** The system **must interpret the transcribed speech to determine the user's intent**. For example, if someone says "Show John 3 16", the system should recognize this as a `find_scripture` intent with parameters `{book: John, chapter: 3, verse: 16}`. It should support a set of predefined voice commands/intents, such as finding a scripture slide, finding a song lyric slide by title, advancing to the next slide, going back, clearing the screen, etc.

- **FR-1.4 – Music Identification:** When music is playing (e.g. a background track or a band starts a song), the system **must identify the song in real time**. This involves generating an audio fingerprint of the music and checking it against two databases: a **custom content database** of songs provided by the operator (e.g. worship tracks uploaded in advance), and a **global music database** (via a service like ACRCloud) for commercially available songs. Upon identifying the song, the system could use this info (e.g., to automatically display the lyrics of the recognized song if available).

- **FR-1.5 – ProPresenter Control:** For each recognized intent (from speech or music), the system **must execute the corresponding action in ProPresenter 7 via its APIs**. This means translating intents such as "find scripture John 3:16" or "lyrics for Amazing Grace" into actual API calls to ProPresenter. The system should handle: triggering specific slides or presentations (by name or index), moving to the next/previous slide on cue, clearing content layers (like clearing lyrics or backgrounds), and any other relevant presentation controls required by voice commands.

- **FR-1.6 – Content Management Portal:** The system **must provide a secure web interface** for operators to upload and manage content. This includes uploading new music tracks or videos (which the system will process for recognition), and uploading speech audio/transcript pairs for improving the speech recognition model. The portal should accept files (audio, video, or a batch of training data) and trigger the system to process them (for example, extracting audio from videos or adding new songs to the recognition database).

- **FR-1.7 – Human-in-the-Loop Verification:** The system **must implement a verification mechanism for low-confidence events**. If the speech recognizer or intent parser is unsure (below a confidence threshold) or the music ID is uncertain, **Kairos** should flag the event and present it to the human operator via an interface. The operator can then confirm or correct what was said or what action was intended. The system will then use these corrections as learning data (improving the speech model or adjusting the vocabulary/phrases for future).

4. Non-Functional Requirements

In addition to the functional behaviors, **Kairos** must meet these performance and quality requirements:

- **NFR-1.1 – Low Latency:** The system should operate in near real-time. From the moment a voice command finishes speaking (end of utterance) to the action being performed in ProPresenter, **at most 3 seconds** should have elapsed for 95% of commands. This end-to-end latency includes all processing steps (audio detection, transcription, intent parsing, and command execution). Lower latency is strongly preferred for a smooth live experience.

- **NFR-1.2 – High Accuracy:** The speech recognition component should achieve **at least 95% transcription accuracy** for typical in-domain phrases (especially scripture references, song titles, and common commands) under expected acoustic conditions. Intent recognition (the NLU agent) should correctly identify the intended command **at least 98% of the time** for clearly spoken inputs. Music identification for known (uploaded) tracks should be **over 99% accurate**, and for general (commercial) music should be industry-standard accurate. High accuracy is critical to trust the system during live events.

- **NFR-1.3 – Scalability & Availability:** The system's backend services (e.g., the orchestration server) should be designed to scale horizontally to handle higher loads or multiple audio streams if needed. Stateless components like the orchestration API should be easily duplicable. Critical real-time components (audio capture and processing clients) should have failover strategies (for example, the audio stream could be picked up by a standby instance if one fails). The system should be robust such that there's no single point of failure that can bring down live operation.

- **NFR-1.4 – Reliability & Resilience:** The system must handle network or service interruptions gracefully. For instance, if the connection to ProPresenter or to an external API (Google STT, Dialogflow, ACRCloud) is lost or times out, **Kairos** should retry operations with exponential backoff and not crash. It should log failures and recover automatically when services come back online. All external integrations should include robust error handling to ensure continuity during a live session.

- **NFR-1.5 – Security:** Operator interfaces (upload portal, verification UI) must be secure. Only authorized users should be able to upload content or intervene in the system. Data (like audio recordings, transcripts, content metadata) should be stored and transmitted securely, respecting privacy and any licensing (for music content). API keys and credentials for services (Google Cloud, ACRCloud, ProPresenter API) must be protected and not exposed.

- **NFR-1.6 – Maintainability:** The system's codebase and configuration should be well-organized and modular. Components like audio processing, speech recognition, NLU, and ProPresenter interface should be decoupled such that they can be upgraded or replaced independently. The system should include monitoring and logging such that issues can be quickly diagnosed. It should also be straightforward to update the speech model with new training data or add new content to the music database without downtime.

(The above requirements serve as the guiding criteria for designing and evaluating the "Kairos" system. Next, we present a detailed technical architecture that meets these requirements.)

Technical Architecture Document

1. High-Level Architecture Overview

The **Kairos** system is composed of several subsystems working together in a pipeline to convert live audio into actions on ProPresenter. At a high level, the flow is:

Live Audio Input → [Audio Ingestion & Classification] → [Speech Processing or Music ID] → [Intent Interpretation] → [Orchestration] → [ProPresenter Command Execution]

To achieve this, the architecture is divided into the following major components:

- **Audio Ingestion & Triage Subsystem (AITS):** Captures the live audio and breaks it into meaningful chunks, determining whether each chunk is speech or music. This prevents unnecessary processing and routes audio to the correct downstream service (speech recognizer or music identifier).

- **Cognitive Verbal Processing Subsystem (CVPS):** Handles speech audio chunks. It uses a Speech-To-Text engine (Google Cloud STT v2) to transcribe speech to text, then uses a Natural Language Understanding agent (Dialogflow CX) to parse the text into a structured intent with parameters.

- **Acoustic Melodic Identification Subsystem (AMIS):** Handles music audio chunks. It uses an audio fingerprinting approach (via ACRCloud services) to identify what song (title/track) is playing, either by matching to a custom database of known tracks or a global database.

- **System Orchestration Core (SOC):** A central controller (implemented as a FastAPI server) that receives events (transcribed text with intent, or identified song info) and decides what action to take. It maps recognized intents to specific ProPresenter API calls. It also manages communication between components via message queues and coordinates the overall sequence.

- **Presentation Control Gateway (PCG):** The module that interfaces directly with ProPresenter 7. This component knows how to send commands to ProPresenter through available APIs (WebSocket, HTTP REST, or TCP) to perform actions like triggering a specific slide, going to next/previous slide, clearing layers, etc. It also maintains a **local cache of the presentation library** to allow searching for items by name (since ProPresenter's APIs do not provide a direct search-by-name for slides or Bible verses).

- **Data Management & Training Pipeline:** Supporting services that run in the background or as one-time setups – including a web portal for content upload, processes to handle uploaded media (like extracting audio from videos, generating fingerprints for songs, etc.), and routines to train or update the speech recognition model and vocabulary using collected data.

- **Human-in-the-Loop Verification Interface:** A small web interface that displays uncertain recognition results (low-confidence transcripts or intents) to the operator for review. This interface ties into a feedback loop where corrections made by the operator are fed back into the system's training data to improve future performance.

Below is an outline of how these components interact in real time:

- The **Audio Ingestion** module captures audio continuously and uses **Voice Activity Detection (VAD)** to find segments of sound (ignoring silence). For each detected sound segment, a **classifier** determines if it's speech or music.
- If it's speech, the audio segment is sent to the **CVPS**: first the STT engine transcribes it, then the text is sent to the NLU agent which returns an intent.
- If it's music, the audio segment is sent to the **AMIS**, which returns the identified song title/metadata.
- The **SOC** receives the final result (intent or song info). It then looks up what action to do (for example, if the intent is "find_lyric_by_title" with song="Amazing Grace", it will prepare a command to load the "Amazing Grace" presentation in ProPresenter). It calls the **PCG** to execute that command.
- **PCG** uses its persistent connection to ProPresenter to trigger the appropriate presentation or action. If "Amazing Grace" lyrics are found in the cache, it triggers that presentation to the screen. If the intent was "next slide", it sends the next-slide command, and so on.
- If any step encounters low confidence or ambiguity (e.g., speech not clearly recognized), the event is logged for human verification rather than acted on immediately.

All these steps are optimized to meet the real-time requirement (under 3 seconds end-to-end). The design choices for each component emphasize low latency, for example by using streaming audio processing and maintaining persistent connections to avoid setup delays.

In the following sections, each component and its technical implementation are described in detail.

2. Audio Ingestion & Triage (Real-Time Audio Processing)

This subsystem continuously listens to the environment and smartly directs audio to the right processing path, forming the first stage of the pipeline.

- **Audio Capture:** We use a Python audio library (such as `sounddevice` or `PyAudio`) to open a live audio stream from the microphone or sound mixer. The audio is captured in a single channel (mono) at a suitable sample rate (e.g. 16 kHz, which is standard for speech recognition). The capture runs in a separate thread or async coroutine, providing a steady stream of audio frames.

- **Voice Activity Detection (VAD):** Raw audio is first processed by a VAD algorithm to detect segments that contain speech or music (as opposed to silence or ambient noise). We plan to use **Silero VAD**, a pre-trained, high-accuracy model for voice activity detection. Silero VAD can analyze short audio frames and output timestamps where speech is present. By applying VAD continuously, the system can isolate chunks of audio that contain any kind of sound event. This reduces unnecessary load on the next stages because we won't send silence or background hum for further processing.

- **Audio Classification (Speech vs Music):** Once the system knows a segment contains sound, it needs to classify the type of sound. For each detected audio segment, we run a lightweight audio classifier to determine if the segment is likely **speech** (spoken words) or **music**. We plan to use the **YAMNet model (YouTube Audio Set classifier)** or Google's MediaPipe Audio Classifier with YAMNet, which is capable of labeling audio segments with categories like "Speech" or "Music". YAMNet is efficient and can run in real time. If the classifier's top category for a segment is "Speech", the segment is tagged for speech processing; if it's "Music", it's tagged for music recognition.

- **Routing to Appropriate Queue:** Based on the classification:

- Speech segments are placed onto a **speech processing queue** (an in-memory async queue or a pub-sub channel).
- Music segments are placed onto a **music identification queue**.

This decoupling allows the speech and music processing pipelines to run in parallel and ensures they receive only relevant audio data.

Latency considerations: The VAD and classifier introduce a small overhead, but they are necessary. To keep latency low, we operate them on small chunks (e.g. 100-300 milliseconds of audio) and in streaming fashion. The system is configured to start processing an audio chunk before the person has even finished speaking (using streaming recognition, detailed later). This way, classification and transcription happen concurrently with the user's speech, effectively hiding some of the processing time and achieving a near real-time response.

3. Speech Processing Pipeline (STT and NLU)

When a speech audio segment is identified, it enters the **Cognitive Verbal Processing Subsystem (CVPS)**, which converts speech into a machine-understandable intent.

- **Streaming Speech-to-Text (STT):** For speech transcription we use **Google Cloud Speech-to-Text (STT) API, V2**. The system establishes a streaming gRPC connection to the STT service. This allows audio to be sent in real time and partial transcripts (interim results) to be received before the speaker has finished. We enable `interim_results=True` so that we don't have to wait until the end of a sentence to start figuring out the intent. The STT is configured with a custom **Recognizer** that links to our tailored models:
- We will fine-tune a **Custom Speech Model** (if available in STT V2) with audio data from our environment (provided by the operator). This custom acoustic model helps the STT better handle the specific voices, microphone characteristics, and ambient noise of our use-case. It requires a substantial training dataset (on the order of 100+ hours of audio and transcripts) to be effective.
- We also use **Model Adaptation** resources for vocabulary. This includes a **PhraseSet** (a list of important phrases like song titles, common scripture names, unique terms) and **CustomClass** definitions (for groups of terms like all Bible book names, all possible numbers for scripture references, etc.). These resources boost the likelihood of the STT recognizing domain-specific words correctly [1] [2]. For example, without adaptation, "ProPresenter" might be transcribed incorrectly; with a PhraseSet boosting it, the accuracy improves. New song titles or terms can be added to these adaptation lists on the fly to keep recognition up-to-date.
- The Recognizer ID encapsulates these settings (language model, custom acoustic model, adaptation data, etc.), so each streaming recognition request uses that ID for consistency.

The output from STT is a text transcript of the spoken command. Thanks to streaming and interim results, we might start parsing the command even before it's fully spoken (e.g., as soon as we hear "Show John three...", we know likely it's a scripture command for John 3).

- **Natural Language Understanding (NLU):** The transcribed text is sent to our NLU agent to extract the intent and any parameters. We use **Google Dialogflow CX** for NLU because it can manage complex flows and multi-step commands if needed. In this project, Dialogflow is configured primarily for single-turn voice commands with a set of defined intents:
- **Intents:** We define intents such as `find_scripture`, `find_lyric_by_title`, `identify_current_song`, `trigger_next_slide`, `trigger_previous_slide`, `clear_screen`, etc., based on the voice commands we need to support.
- **Entities:** We set up custom entities for data we need to pull from commands. For example, a `scripture_reference` entity can be created to parse things like "John 3 16" into parts (book = John, chapter = 3, verse = 16). We might use either a regex or a comprehensive list of book names combined with number parsing for this. Similarly, a `song_title` entity could capture the title of a song (possibly treating everything after the keyword as a title).
- **Dialogflow CX Structure:** We likely use a single flow with a start page that routes based on intent. Each intent will simply output parameters and not attempt its own fulfillment (we handle fulfillment in our system). We are essentially using Dialogflow for NLU parsing only, not for managing dialogue or stateful conversations.

When Dialogflow returns the intent, it provides an **intent name** (e.g., "find_scripture") and a set of **parameters** (e.g., book="John", chapter="3", verse="16"). It also provides a confidence score for the intent

match. If this confidence is high enough, we proceed. If it's below our threshold, we treat the result as uncertain and may involve the human verifier (see section on Human-in-the-Loop).

- **Intermediate Results Handling:** Because we use streaming STT with interim results, our orchestrator can even send partially recognized phrases to Dialogflow to get a head start on intent detection. For instance, as soon as the STT returns something like "show John 3", the orchestrator could query Dialogflow. However, we must balance this with accuracy (waiting until the command is complete vs. getting a early guess). The system will be tuned to use interim results to reduce latency but confirm with the final result before executing an action, to avoid mistakes.

Note on accuracy: By using a custom-trained model and adaptation, the STT should handle even unusual terms (like specific song names or technical terms) well. Dialogflow's custom entities ensure that, for example, it correctly captures Bible references. Both of these approaches are aimed at meeting NFR-1.2 (high accuracy for domain-specific language).

4. Music Identification Pipeline (Audio Fingerprinting)

When the audio classifier labels a segment as music, the segment goes to the **Acoustic Melodic Identification Subsystem (AMIS)**. The goal here is to determine if the music matches a known song and identify which one.

- **Audio Fingerprinting:** The system will generate a fingerprint of the audio segment. A fingerprint is a compact representation of the audio's unique characteristics. We use ACRCloud's fingerprinting technology. The architecture supports two kinds of matches:
- **Custom Content Match:** We maintain a custom fingerprint database for songs that the operator has specifically uploaded (like worship songs, backing tracks, etc. that might not be in any global database). When a new song is uploaded via the content portal, the system uses ACRCloud's tools (which can be run locally) to generate a fingerprint file for that audio. Instead of uploading the entire audio, we upload just the fingerprint to ACRCloud's custom database【user content】. This is efficient and recommended by ACRCloud for large libraries, as the fingerprint files are very small.

- **Global Database Match:** If the song isn't in the custom database, we also query ACRCloud's global music database (which can identify commercial songs by artists). The ACRCloud SDK or API can return details like track title, artist, album, etc., for recognized music.

- **Recognition Service:** We use the **ACRCloud Recognizer API** (with our specific keys and host for the custom bucket) to identify the song. The system will send the audio (or its fingerprint) to ACRCloud's endpoint. This can be done either by sending raw audio (for small segments) or by pre-computing the fingerprint and sending that. ACRCloud will respond with a JSON result. The result will indicate if a match was found in the custom bucket (e.g., `metadata.custom_files` in the response) or in their global database (`metadata.music`). From the response, we extract the song title (and maybe other details like the unique ID if needed).

- **Result Handling:** Once the song is identified, the SOC is informed of the **song title** (or a specific identifier for a presentation). For example, if the system hears a song and recognizes it as "Amazing Grace", it will treat it similarly to a voice command asking for "Amazing Grace" lyrics. The orchestrator can then trigger the lyrics display for that song automatically, assuming that song's

slides are prepared in ProPresenter. If the song is not recognized, the system might do nothing (or possibly prompt the operator if that was unexpected).

Performance: Audio fingerprinting and identification is generally fast (often under a second), but to ensure we meet the latency goal, we might fingerprint only a few seconds of audio. Also, since the audio is continuous, we can perform identification on overlapping chunks to catch a song as soon as possible after it starts. ACRCloud's service is optimized for real-time recognition of music, so we rely on that performance, but we also ensure we don't flood it with requests by only sending when music is steadily detected (not on every minor blip of music).

5. System Orchestration Core (Backend Service)

The **System Orchestration Core (SOC)** is the brain that coordinates all subsystems. It's implemented as an asynchronous Python backend (using **FastAPI** for the web framework). FastAPI provides both a web interface (for any HTTP endpoints we need, such as the content upload or verification UI) and a robust event loop for handling background tasks and integration with message queues.

Key responsibilities and design of the SOC:

- **Message Queue Integration:** The SOC will listen to the audio routing queues. For example, it subscribes to the `speech_queue` and `music_queue` where the audio ingestion subsystem publishes events. This could be done with an in-memory queue if everything runs in one process, or a Redis pub/sub or similar if distributed. When a speech audio event comes in, the SOC hands it to the STT service; when a music event comes in, the SOC hands it to the music ID service.

- **Parallel Processing:** Because the tasks (transcription, intent detection, music ID) involve waiting on external services, the SOC uses asynchronous calls or background workers. For instance, it can call the Google STT API in a non-blocking way, so multiple audio segments can be in flight if needed. FastAPI's async support and possibly background task features (or Celery/RQ if needed for heavier tasks) will be utilized.

- **Intent-to-Action Mapping:** Once the SOC receives a result (either an intent from Dialogflow or a song ID from ACRCloud), it decides what to do:

- For a **song identification** result (e.g., "Song X identified"), the SOC could treat it as if the user requested that song's lyrics. It would look for the presentation name for Song X.
- For a **voice command intent**, the SOC has a direct mapping. For example:
    - `find_scripture` intent → the SOC expects parameters like `book`, `chapter`, `verse`. It will format a scripture reference string (e.g. "John 3:16") and instruct the PCG to find that presentation.
    - `find_lyric_by_title` intent → the SOC takes the `song_title` parameter and asks the PCG to load that song's presentation.
    - `trigger_next_slide` intent → the SOC calls PCG to go to next slide.
    - `clear_screen` intent → the SOC calls PCG to clear a certain layer (the intent might specify lyrics, backgrounds, or all).

- `identify_current_song` intent (if implemented as a voice command like "What song is this?") → the SOC would route that to AMIS to get the song info, then possibly speak it out or display it (though speaking it out is beyond scope since we don't mention TTS).

Essentially, the SOC contains the **business logic** that links an input (speech or music event) to an output (ProPresenter action). It can be implemented with a simple if/else or mapping table for intents to functions.

- **Error Handling and Logging:** The SOC will also handle cases where an intent cannot be fulfilled – for example, if the user requests a song or scripture that doesn't exist in the ProPresenter library. In such cases, the SOC might log an error or send a message back (e.g., a TTS response or just an operator alert, depending on desired behavior). For our architecture, likely it will just log it and maybe the operator will notice nothing happened and intervene.

- **Human-in-the-Loop Trigger:** If the confidence from the STT or NLU was below threshold, the SOC will not immediately execute the action. Instead, it will create a record in a verification datastore. For instance, it might store: "Audio at 10:32:15, transcribed as 'show John 3 60', confidence 60%". This gets displayed in the verification UI for the operator to check (perhaps the operator can see that it should have been 'John 3 16', and correct it). The SOC might also have a simple rule: do not execute low-confidence commands without approval. This prevents potentially wrong slides from showing up on screen.

- **Web API Endpoints:** In addition to background processing, the SOC (FastAPI app) will expose some endpoints:

- `POST /upload/music` – for the content upload portal (to receive a music file and process it, see Data Management section).
- `POST /upload/video` – to upload a video or a link for processing.
- `POST /upload/speech_corpus` – to upload training data (audio + transcripts).
- Perhaps endpoints to retrieve the list of pending verification items, or to submit a verification correction.

- These endpoints ensure the system can integrate with external interfaces (like a web frontend for uploads or verification).

- **State Management:** The SOC itself is mostly stateless in terms of processing (each audio event is handled independently), which aids scalability. However, some state is needed like the ProPresenter library cache (which is maintained in the PCG component) and perhaps some session state if we had multi-turn interactions (not planned initially). The design tries to keep things stateless so multiple instances of the SOC could run behind a load balancer if needed (though the audio capture likely runs on one machine near the audio source).

6. ProPresenter Integration and Control (Presentation Control Gateway)

Interfacing with **ProPresenter 7** is a critical part of the system. ProPresenter is presentation software that can be remote-controlled via APIs, but its official APIs have limitations regarding searching and triggering arbitrary slides by name. The **Presentation Control Gateway (PCG)** is our custom module to bridge the

gap between what the system intends to do (e.g., show a certain song or Bible verse) and what the ProPresenter APIs can do.

Key aspects of the PCG design:

- **Connection Methods:** ProPresenter offers a few integration points:
- A **WebSocket API** (which is actually used by ProPresenter's own remote control apps). This requires connecting to a WebSocket server on the ProPresenter machine (usually ws://<ip>:<port>/remote) and sending JSON commands. It's not officially documented widely, but community documentation exists. It supports actions like advancing slides, triggering specific presentations by index, etc.
- An **HTTP REST API** (introduced in ProPresenter 7.9+), which has endpoints for certain actions (like triggering items by ID, clearing screens, etc.). This is officially documented but somewhat limited for our needs of searching by name.
- A **TCP Socket API** using JSON (on a specific port) – similar in functionality to the WebSocket, sometimes easier to interface via a simple socket if needed.

The PCG will likely use a combination, but primarily the **WebSocket API** for real-time control because it is persistent (no HTTP setup overhead each time) and supports the library queries we need.

- **Authentication:** ProPresenter's remote API requires an authentication step (sending a password in an auth message once connected). The PCG will handle this by reading credentials from config and authenticating the moment it connects to the ProPresenter WebSocket.

- **Library Cache Building:** One of the first things the PCG will do on startup is retrieve the list of available presentations (song slides, scripture slides, etc.) from ProPresenter:

- It can send a `libraryRequest` action via WebSocket which prompts ProPresenter to send the entire library listing (each item with an identifier, name, etc.).
- It can also send `playlistRequestAll` to get all playlist structures (which include the ordering of items, though for search-by-name, the library listing is more important).
- The PCG will store this information in memory (as a Python dictionary or similar, mapping presentation names to their identifiers or indices in the library). For example, it might store that "Amazing Grace" corresponds to library item ID `xyz123` or is at index 5 in the library list.

- It likely also retrieves the Bible verse slides if they are stored as presentations (some versions of ProPresenter allow generating scripture slides on the fly, but if not, the operator might have to pre-generate them). For our purposes, we assume scriptures we need are also in the library as presentations named like "John 3:16" or "John Chapter 3".

- **Triggering Presentations by Name:** Since the voice commands refer to songs or scriptures by name/reference, and since we have the library cached:

- For a `find_lyric_by_title` intent with a song title, the PCG will look up the exact (or nearest match) in the library cache. If found, it obtains the necessary info (depending on API: could be a numeric index or a unique ID).
- Using WebSocket, we can send a `presentationTriggerIndex` command that requires the index of the item in the library or playlist. If we know it's the 5th item in the library and belongs to a certain

playlist, we include those indices in the JSON payload. ProPresenter will then load that presentation to the screen (usually at the first slide).
- Alternatively, with the REST API, if we had the item's unique identifier or name, we could use an endpoint (if available) to trigger it. The documentation suggests endpoints to trigger by ID, but since we already have the WebSocket open and library indices, using WebSocket is simpler and faster.

- For a `find_scripture` intent, the approach is similar: the PCG looks for a presentation whose name matches the requested reference (like "John 3:16"). If found, it triggers it. **Important:** This requires that the operator has pre-added that scripture passage as a slide or presentation in ProPresenter (via ProPresenter's Bible tool or manually). The system itself does not fetch Bible verses; it relies on what's in ProPresenter's library.

- **Simple Commands:** For intents like `trigger_next_slide` or `trigger_previous_slide`, the PCG doesn't need the cache. It can directly send the WebSocket action `presentationTriggerNext` or `presentationTriggerPrevious`, which instructs ProPresenter to advance or go back in the currently live presentation. Similarly, a `clear_screen` intent might translate to a specific API call like an HTTP GET to `/v1/clear?Layer=...` or a WebSocket `clearAction` message. We will implement these using whichever API call is documented. (ProPresenter allows clearing different layers: e.g., clear only lyrics, or backgrounds, etc., and those are accessible via API.)

- **Fallbacks and Coverage:** If for some reason the WebSocket connection fails or doesn't support a needed feature:

- The PCG might use the HTTP API as a fallback for certain operations (for example, sending a message to a stage display or triggering a macro, if those were needed).

- The TCP JSON interface is very similar to WebSocket and might not be needed if WebSocket works, but we keep it as an option in case direct socket usage is simpler for some environments.

- **Performance:** The PCG maintains a persistent connection to avoid the latency of reconnecting or HTTP handshakes on each command. This means once **Kairos** starts, it stays connected to ProPresenter's remote API at all times. Commands are then just a matter of sending a small JSON message and ProPresenter responding, which is typically very fast (tens of milliseconds on a local network).

- **Limitations and Dependencies:** We note that the ProPresenter integration requires some **setup by the operator**:

- The ProPresenter machine must have its remote API enabled and accessible to our system.
- Content (songs, scriptures) must exist in the library. The system can't display what isn't there. For scriptures, one approach is the operator uses ProPresenter's scripture tool to pre-fetch and save slides for all commonly referenced verses or at least ensure internet connectivity for ProPresenter to pull them live. For songs, the operator must have imported the lyrics as a presentation.
- If a voice command comes in for something not found, we might log it or possibly, in future, we could add logic to instruct the operator to add it.

7. Data Management and Model Training

While the above sections describe the live, real-time components, **Kairos** also includes supporting services for preparing and updating the system's intelligence:

- **Content Upload Portal:** Implemented as part of the FastAPI application (or a separate web service), this portal allows the operator to upload:
- **Music files (audio)** – e.g., new worship songs or background tracks. When a file is uploaded (via an HTTP POST request with the file), the system saves it (likely in cloud storage or local storage) and then triggers a fingerprint generation process (using ACRCloud's local fingerprint tool). The resulting fingerprint is then uploaded to our ACRCloud custom bucket so that the song becomes recognizable in the future.
- **Video files or links** – if the operator has a video (maybe a YouTube link of a song or a local video file), the system will download or accept the file, then run a process (via `ffmpeg`) to extract the audio track from the video. That audio track is then fingerprinted as above for music identification purposes.
- **Speech training data** – a .zip or batch of audio recordings with their transcripts. This data is used to improve the STT model. For instance, if the operator recorded the pastor saying various scripture references or typical phrases, those can be uploaded and later used to custom-train the speech model.

The portal ensures these tasks happen asynchronously (the upload request returns quickly, and processing continues in background). Tools like Celery or background tasks within FastAPI can handle this.

- **Speech Model Customization:** After collecting enough domain-specific audio data (ideally many hours of recordings from the actual venue, voices, and terminology), the AI can fine-tune a custom STT model. Google's STT v2 allows training a custom model by providing a set of audio files and transcripts in Cloud Storage. We plan the following:
- Use the Google Cloud SDK or client library to start a custom model training job, referencing our dataset. This might be done once initially (Phase II in planning) and possibly updated occasionally if new data is added.
- Once trained, the custom model is deployed and referenced in our Recognizer resource that the streaming STT client uses.
- In parallel, maintain and update the adaptation PhraseSets for new vocabulary. For example, if a new song titled "Lord of All Hope" is added, we update the PhraseSet so that the STT will better recognize that exact phrase in the future.

- The combination of an acoustically tuned model and up-to-date vocabulary boosting ensures that the system stays accurate as new content is introduced.

- **Continuous Learning Loop:** The verification feedback (from FR-1.7) is fed into this training pipeline. Concretely, any time the human operator corrects a mistake (like the system misheard something), that piece of audio and the correct transcript can be added to the training corpus. The system can periodically (say, weekly or monthly) retrain or at least adjust its adaptation lists based on these corrections. This way, the more the system is used and corrected in a specific environment, the more accurate it becomes.

- **Database/Storage:** For storing metadata, we might have a database (like PostgreSQL or a simple file storage) to keep track of uploaded content metadata, mapping them to ProPresenter library items, and storing verification logs. The audio/video files might live in a storage bucket or local disk. Fingerprints might only live on ACRCloud, but we could keep copies if needed.

8. Human-in-the-Loop Verification Mechanism

To ensure reliability and continuous improvement, **Kairos** integrates a human verification step for handling uncertain recognition outcomes:

- **Detection of Uncertainty:** Each major step provides a confidence score (Google STT can provide word-level and utterance-level confidence; Dialogflow provides intent detection confidence; ACRCloud might provide a score for the match). The SOC will check these scores. For example, if speech was transcribed with low confidence or the NLU returned a fallback intent or low confidence, we mark the event as uncertain.

- **Queue for Verification:** Instead of executing a low-confidence command, the SOC will create an entry in a **Verification Queue/Log**. This entry contains:

- A timestamp and perhaps an audio clip of the segment in question.
- The system's transcribed text and identified intent (if any).
- The confidence score and which step was low (STT or NLU or both).

- An empty field for "verified correction."

- **Verification UI:** A separate simple web interface (which could be a page served by FastAPI or a separate small app) allows the operator to view these entries. The UI might list items like:

- *"Audio at 10:32:15 -> heard 'John 3 60' (low confidence). Intent guessed: find_scripture. [Play Audio] [Text Box for correction]"*.
- The operator can play back the recorded audio snippet, see what the system thought, and then type the correct transcript or select the correct intent if the system got intent wrong.
- For a music ID uncertainty, it might show: *"Music at 10:35:00 -> heard unknown (low confidence match). [Play Audio] [Dropdown of possible songs or text]"*.

- Essentially, it gives the human a chance to fix the mistake.

- **Feedback Integration:** When the operator submits a correction, the system can immediately take action if it's still relevant (for instance, if the operator corrects "John 3:16", the system might then trigger that scripture slide). More importantly, the correction is logged:

- The audio and corrected transcript can be moved into the training data pool for STT.
- If it was an intent misunderstanding, perhaps the phrasing can be added as a training phrase in Dialogflow for the correct intent.
- If it was a new song the system didn't know, that can trigger adding that song to the recognition database or updating PhraseSets.

This closes the loop such that every error makes the system smarter. Over time the need for human intervention should drop as the models adapt to the specific usage.

- **Thresholds:** The system will be tuned to only flag when necessary, to avoid burdening the operator. For example, if confidence > 90%, it might always trust the automation. If 70-90%, it might execute but also log for review later. If < 70%, it might pause and wait for confirmation. These values will be adjusted based on testing to find a good balance of autonomy vs. safety.

9. Performance and Reliability Strategies

Designing for low latency and high reliability is a core part of the technical architecture:

- **Streaming and Parallelism:** As noted, using streaming STT and overlapping the processes (classification, transcription, intent parsing) helps reduce wait times. The system processes audio as it comes without waiting for an entire command to finish in silence (which could add a noticeable delay).

- **Persistent Connections:** The PCG's persistent WebSocket connection to ProPresenter avoids repeated HTTP overhead. Similarly, maintaining a live gRPC stream to Google STT (with appropriate timeouts) means we don't start a new connection for every utterance – the stream can be kept open for a session of continuous audio, sending audio as it comes and receiving transcriptions.

- **Local Processing vs Cloud:** Some tasks are done locally (VAD, classification, fingerprint generation) to offload work from external APIs and reduce data transfer. For example, using local VAD prevents sending silence to Google; using local fingerprinting avoids uploading large files to ACRCloud. Only the essential data (like a short audio clip or fingerprint) is sent out for recognition.

- **Concurrency and Scaling:** The architecture can be deployed such that the audio processing (AITS) runs on a machine physically receiving audio (e.g. an on-premise PC connected to the sound system), while the orchestrator (SOC) could run in the cloud. They would communicate perhaps over a lightweight message broker. If needed, multiple audio sources (say multiple venues) could each have an ingestion pipeline feeding a central cloud orchestrator that scales to handle multiple streams. The FastAPI core being stateless (except for PCG connection) means multiple instances could serve different events or operate in active-active failover.

- **Error Recovery:** Each integration has a retry logic. For instance:

- If the WebSocket to ProPresenter disconnects (maybe ProPresenter was restarted), the PCG will detect this and continually attempt reconnection (with exponential backoff to not spam).
- If the Google STT streaming call fails or returns an error, the system might reset the stream and start a new one automatically.
- If ACRCloud recognition fails (network issue), it could retry the request once or twice.

- The system logs errors and continues running even if one part fails momentarily. This is vital for a live system (it should never crash outright; at worst it might temporarily not respond to commands, but should recover).

- **Testing for Performance:** We will simulate various scenarios to ensure the 3-second latency is met. For example, measure time from end of spoken command to slide change. If needed, further optimizations like processing shorter audio chunks or increasing local processing power can be used. We will also monitor CPU/memory to ensure the system can run on the target hardware without issues (especially since audio processing and multiple threads can be resource-intensive).

10. Testing and Deployment Considerations

Finally, the technical plan includes thorough testing and a strategy for deployment:

- **Unit and Integration Testing:** Each component will be unit-tested with simulated inputs:
- The VAD and classifier logic will be tested with sample audio files to ensure they correctly identify speech vs music segments.
- The STT and NLU pipeline will be tested by mocking Google STT and Dialogflow responses (to avoid using the actual APIs every test). For example, feeding a recorded audio of "Next slide" and verifying the system interprets the intent and calls the PCG with a next-slide command.
- The PCG will be tested against a **mock ProPresenter server**. We can create a dummy WebSocket server that expects certain messages (like an auth, then a trigger command) and verify that our PCG client sends exactly those when a certain intent is fed in. This ensures our integration code is correct without needing a real ProPresenter instance.

- Edge cases such as network failures, unrecognized commands, etc., will be tested by simulating those conditions in the test environment.

- **End-to-End Testing:** Before deployment, an end-to-end test will be conducted:

- Provide a known audio input (for example, a recording of someone saying "Show John 3 16") and see if the correct slide is triggered in ProPresenter (which could be automated by checking ProPresenter's state via API or a visual cue).
- Similarly, play a known song's clip and verify the system triggers the right lyrics.

- These tests ensure that all components together function as intended and meet the timing requirements. We will script these tests possibly with an automated tool or as part of a CI pipeline.

- **Deployment Architecture:** We plan to containerize the components using Docker. For instance:

- A container for the FastAPI app (which includes SOC, PCG, and possibly the audio processing if on the same machine).
- Alternatively, a separate container for the audio ingestion if it runs on a different host near the audio source.
- A container for any background workers (if using Celery for file processing).

These containers can be deployed on a cloud service (like Google Cloud Run or Kubernetes on GKE) for the core logic. However, since audio input might require local capture, the audio capture component might run on a local machine (which could still run Docker, or just Python directly) and then connect to the cloud services.

Security-wise, we will ensure the connection between the local system and cloud is secure (VPN or at least authenticated messaging), and all credentials (Google API keys, ACRCloud keys, ProPresenter password) are stored in secure config (not hard-coded).

- **Monitoring and Logging:** Deployed system will have logging of key events (e.g., every command recognized and action taken) so that the operators or developers can audit what happened if something goes wrong. We might integrate with a logging service or at least keep logs on disk. Additionally, monitoring can be set up for resource usage and API quotas (e.g., Google STT quota) to avoid hitting limits. Alerts can be sent if, say, the STT usage is nearing a daily limit or if any component goes down.

- **Future Expansion:** The architecture is built to be extensible. New voice commands can be added by updating the Dialogflow agent and adding handling in SOC and maybe new API calls in PCG. The system could also integrate other inputs (like maybe a text chat control or schedule triggers) relatively easily because of the modular design. But those are beyond the initial scope.

Conclusion: The above technical architecture is a comprehensive design to fulfill the functional requirements of the **Kairos voice-controlled presentation system**. By carefully segmenting the problem (audio triage, speech vs music processing, orchestration, and integration with ProPresenter) and addressing performance and reliability at each step, the system aims to deliver a seamless and trustworthy tool for live-event media control. All components, from custom speech models to the ProPresenter gateway, work in concert to allow an operator to simply **speak** (or play a song) and have the correct visuals appear, with minimal latency and high accuracy. This architecture also ensures that as the system is used over time, it **learns and improves**, bridging the gap between human flexibility and AI automation in a live production setting.

---

1 2 Improve transcription results with model adaptation | Cloud Speech-to-Text V2 documentation | Google Cloud Documentation
https://docs.cloud.google.com/speech-to-text/v2/docs/adaptation-model