ChatGPT

Kairos Voice-Activated Media Control – Functional Requirements Document

Mission, Scope, and Assumptions

Mission: The Kairos agentic AI system's mission is to autonomously **design, build, and operate** a voice-activated media control application ("Kairos") that allows hands-free control of live presentation media (e.g. slides, lyrics, audio) via voice commands and audio cues. This AI will serve as an autonomous developer and orchestrator, creating a system that listens to audio in real-time and triggers the appropriate media actions with minimal human intervention.

Scope of Control: The AI agent has **full control over the Kairos system architecture and components**, including software modules and their interactions. Its scope is limited to the media control domain: - It can ingest live audio (microphone or line-in feeds) and analyze it. - It can invoke external AI services (speech-to-text, music recognition, NLU) as needed. - It can issue control commands to the presentation software (ProPresenter) over the network. - **Out-of-Scope:** The AI will **not** control anything outside the media application (e.g., no direct internet browsing or system admin tasks beyond its deployment needs). It will not override human operators on critical decisions – instead, it defers to humans via a verification UI when uncertain or when outside predefined intents. All actions are logged for audit by governance reviewers.

Key Assumptions: - **Environment:** Kairos will be deployed in a live production environment (e.g. a church or live event venue) with a continuous audio feed (from microphones or mixers) available. A stable internet connection is available for cloud APIs (Google STT, Dialogflow CX, ACRCloud). - **External Services Access:** The AI is provisioned with API credentials and network access to Google Cloud Speech-to-Text V2, Dialogflow CX, and ACRCloud services. These services are assumed to be reliable and meet their stated performance (latency and accuracy) in the deployment region. - **ProPresenter Control:** The presentation software (ProPresenter 7.9 or later) is running on the same local network and has its API interface enabled (via WebSocket/TCP). The AI has the needed authentication token or credentials to send control commands to ProPresenter's remote control channel. - **Content Preparedness:** All media content (slide decks, lyric presentations, etc.) are pre-loaded in ProPresenter or accessible to Kairos. A **Data Ingestion Portal** exists for human operators to upload new content (e.g. new song lyrics and associated audio fingerprints) prior to events. It's assumed operators will use this portal to keep the content library updated. - **User Interaction:** End-users (e.g. speakers, musicians) will issue voice commands or musical cues in a reasonably clear manner (e.g. speaking close to a mic, minimal background noise when giving commands). The AI assumes a single primary audio source at a time (it will handle overlapping music/speech by triaging, but extremely noisy or simultaneous command scenarios are not guaranteed to be understood without human help). - **Autonomy and Oversight:** The AI will operate autonomously under normal conditions, but a human tech operator will be present with a **Human-In-The-Loop (HITL) Verification UI**. The AI assumes that if it requests human confirmation (for ambiguous cases or low-confidence decisions), a human will respond via the UI promptly. The AI will refrain from executing potentially disruptive actions without either high confidence or human approval.

With these foundations, the following sections break down the Kairos system into functional modules, detailed requirements, design principles, external interfaces, and non-functional targets.

System Overview and Module Breakdown

Kairos is structured into several functional modules, each responsible for a stage of processing or interaction. The agentic AI will design these modules as an integrated pipeline (with parallel branches for speech and music input). The major modules are:

- **Audio Triage Module:** Front-end audio processing that listens continuously and classifies incoming audio to decide how to route it (speech command vs music vs other).
- **Speech Transcription Module:** Converts spoken commands into text using streaming Speech-to-Text (Google STT V2).
- **Music Identification Module:** Identifies recognized music (e.g. a song being played or hummed) via audio fingerprinting (ACRCloud service).
- **Natural Language Understanding (NLU) Module:** Interprets transcribed speech to determine the user's intent (using Dialogflow CX), and extracts any parameters (e.g. song titles, slide identifiers).
- **Presentation Orchestration Module:** Orchestrates the control of media/presentation software (ProPresenter) based on the identified intent – e.g. advancing slides, displaying a specific song's lyrics, playing or pausing media.
- **Data Ingestion Portal:** A web-based interface for content management. Allows human operators to upload or manage media assets (lyrics slides, audio files) and link them to the system (e.g. adding new songs into the recognition library and mapping them to presentation slides).
- **HITL Verification UI:** A real-time dashboard for human-in-the-loop verification. It displays the AI's recognized commands/cues and intended actions, allowing a human to confirm or correct the AI's decisions in uncertain cases. It also logs feedback for continuous learning.

Each module's functional requirements are detailed below, including their inputs, outputs, triggers, dependencies, success criteria, and suggested implementation technologies. All requirements are labeled **FR-x.y** (where *x* refers to the module and *y* is the requirement number) for traceability and testability.

Functional Modules and Requirements

Audio Triage Module

Description: The Audio Triage module is the always-listening front-end of Kairos. Its role is to monitor the live audio stream and immediately classify the audio content, routing it appropriately: - If the audio contains **speech (a voice command)**, it should be sent to Speech Transcription. - If the audio contains **music (e.g. someone starts singing or a backing track)**, it should be funneled to Music Identification. - It may also detect if the audio is **silence or irrelevant noise**, in which case no further action is taken (to save processing). This module acts as the gatekeeper ensuring the right analyzer handles the audio, thereby optimizing system latency and accuracy.

Functional Requirements (FR-1.x):
- **FR-1.1: Voice/Music Classification** – The system **shall analyze incoming audio in real-time and determine whether it is speech, music, both, or none**. This classification must occur with low latency (within a few hundred milliseconds of audio onset) to promptly direct the audio to the correct pipeline. The

classification should have a high detection accuracy (e.g. >95% accuracy in distinguishing speech vs. music under typical conditions) to avoid misrouting.
- **FR-1.2: Voice Activity Detection** – The Audio Triage module **shall perform voice activity detection (VAD)** to detect the presence of human speech segments. When speech is detected (and dominates the audio), it triggers the Speech Transcription module (FR-2.1). The VAD should be sensitive enough to catch a spoken command even if background music is present, and use buffering if needed to capture the full command utterance.
- **FR-1.3: Music Onset Detection** – The module **shall detect when music or singing starts** (e.g., a song being played or hummed). Upon detecting a music segment (without an overlapping explicit speech command), it triggers the Music Identification module (FR-3.1). The system should continue streaming audio to Music ID for a few seconds to allow the identification to get a fingerprint. If both speech and music are detected concurrently, speech commands take priority (e.g. a spoken command over background music is handled as a voice command).
- **FR-1.4: Noise/Silence Handling** – If the audio is silence or non-relevant noise, **no downstream module shall be triggered**. The system continuously listens but remains idle (does not send empty audio to STT/ACR to save resources). Minor background sounds should not accidentally trigger any module (to avoid false positives). Thresholds for silence and noise levels are configurable.

Inputs & Outputs:
- **Inputs:** A continuous audio stream (PCM or similar) from the venue's microphone or sound system. The audio stream is chunked (e.g. into frames of 20-50ms) for analysis.
- **Processing & Decision:** The module uses algorithms (e.g. an audio classifier or heuristic) to label each time frame or short buffer as "speech", "music", "both", or "none". Likely it uses features like frequency content and energy to differentiate speech vs music (e.g. speech has distinct voice formants, music might have steady rhythms/harmonics). A simple approach is applying a **VAD algorithm** (to detect voiced speech) in parallel with a **music detection algorithm** (possibly via an ML model trained on music vs speech).
- **Outputs:** A **routing signal or event** to other modules: - If classified as speech: output triggers the Speech Transcription module with the buffered speech audio. - If classified as music: output triggers the Music Identification module with the audio sample. - Optionally, output could include a label confidence score (to inform if the classification was uncertain, which could later be used for deciding HITL intervention if needed). - **Trigger Conditions:** Continuous operation. The module is essentially **always on**, analyzing audio frames as they come. A **stateful VAD** may accumulate a trigger only when a valid voice command phrase is detected (e.g. after initial silence -> speech transition). Similarly, music detection might trigger after a few seconds of music confirmed. - **External Dependencies:** This module might use external libraries or models but runs locally for real-time performance. For example: - *WebRTC VAD* library for efficient voice activity detection. - A small TensorFlow or PyTorch model for music vs speech classification (could be a lightweight CNN on audio MFCCs). However, no external cloud API is needed here; classification must be immediate and on-premises. - **Success Criteria:** The triage is successful if it correctly routes audio with minimal delay and error: - **Accuracy:** $\geq$ 95% of spoken commands are correctly identified as speech and not missed or misclassified as music. $\leq$ 5% false positive rate (e.g. music incorrectly flagged as speech or vice versa). - **Latency:** Classification decision made within <100ms of audio onset for speech (to start STT quickly) and within 1–2 seconds for music (since a short window of audio is needed for reliable music ID fingerprint). - **Robustness:** In case of overlapping speech and music, the system still captures the speech for STT (even if music is present). In worst case, the system may pursue both pipelines (speech and music) in parallel if unsure, then discard the irrelevant one upon clearer info. - **Suggested Tech Stack:** C/C++ or Python for implementation. For instance, use the **WebRTC VAD** in Python (via `webrtcvad` module) to get speech boundaries. For music detection, a simple approach is measuring audio spectral entropy or trained

classification (there are open-source models for music/speech detection). This could be integrated in Python. The AI agent might use frameworks like **PyAudio** or **librosa** for audio processing. The module will be packaged as a microservice or thread that constantly processes audio frames and emits events (could be implemented as part of a Python service using async IO or as a separate process communicating via a message queue to downstream modules for decoupling).

Table: Audio Triage Module Summary

Feature	Description
Inputs	Live audio stream (PCM 16kHz, etc.) from microphone or mixer.
Processing	Real-time classification of audio as Speech vs Music vs None (silence/noise). Uses VAD and possibly ML-based audio classification.
Outputs	Route event with audio buffer to Speech Transcription (if speech detected) or Music ID (if music detected). No output (idle) if silence/no irrelevant audio.
Trigger Conditions	Always listening. Triggers on voice activity start or music onset.
Dependencies	Local VAD library (e.g. WebRTC VAD); optional ML model for music detection.
Success Metrics	≥95% correct routing of speech vs music; classification latency <100ms for speech onset.
Tech Stack	Python with audio processing libs (PyAudio, librosa), WebRTC VAD; runs as low-latency service thread.

Speech Transcription Module

Description: Once audio is classified as containing speech (a voice command), the Speech Transcription module handles converting that speech **audio to text**. It uses Google's cloud-based Speech-to-Text API (STT) v2 in **streaming mode** for real-time transcription. The goal is to get highly accurate transcriptions of commands with minimal delay. By using streaming STT, the system can receive partial transcripts as the user is speaking and even perform **early intent detection** before the user finishes speaking, to further reduce end-to-end latency. This module focuses on transcribing short utterances (commands are typically a few seconds long at most).

Functional Requirements (FR-2.x):
- **FR-2.1: Continuous Streaming STT** – The system **shall initiate a streaming speech recognition session** with Google STT V2 when speech audio is detected (from Audio Triage). The audio from the command is fed to the API in real-time. Google's STT v2 supports infinite streaming with appropriate configuration (using the `long` model to avoid premature cutoff). The stream remains open for the duration of the voice command and can be closed after a silence indicates end-of-utterance.
- **FR-2.2: Accuracy and Language Model** – **Transcription accuracy shall be optimized** by using appropriate language models and hints. The module will configure STT with the relevant language (e.g., English – locale specific to user) and enable automatic punctuation for readability. If there are domain-specific terms (song names, technical phrases), the system shall use STT's **speech adaptation** (custom

phrases/hints) to improve recognition of those terms. For example, common song titles or command words can be given a boost in the STT config. The target is to achieve a **Word Error Rate (WER) < 10%** for typical command sentences.
- **FR-2.3: Low Latency Partial Results** – The module **shall utilize partial transcription results** from the streaming API to achieve early intent detection. As the user speaks, interim results are provided by Google STT (with stability scores). The system should read these partials and, if they sufficiently indicate an intent (e.g., the first words are "Next slide"), it can start preparing the action even before the final transcript is complete. This requires careful logic: e.g., only act on partials that meet a confidence/stability threshold or where an intent can be unambiguously determined early. This feature is aimed to minimize perceived latency – ideally, the system starts executing the command immediately after the user finishes speaking (or even as they are finishing).
- **FR-2.4: End-of-Utterance Detection** – The transcription session **shall automatically end when the user has finished speaking** the command. Google STT v2 can automatically finalize a transcript when a brief silence is detected at the end of speech. The module should either rely on STT's endpointer or use the local VAD to detect end-of-speech, then stop streaming. This prevents unnaturally long open sessions and reduces cost. The transcript (final result) is then passed to the NLU module.
- **FR-2.5: Multi-Utterance Handling** – If multiple voice commands are spoken back-to-back, the system shall handle them sequentially. This means after one utterance is finalized and processed, the stream can remain open (infinite stream mode) to catch subsequent utterances without needing to reconnect to the API. The **long-running recognizer** mode of Google STT v2 will allow continuous recognition across utterances, separated by silence, without losing context (the model knows when a sentence finishes and returns transcription without ending the stream). The requirement is that the system supports at least **one user speaking at a time**, with the assumption that commands are given one after another (not overlapping). (Handling truly simultaneous commands from multiple people is out of scope.)

Inputs & Outputs:
- **Inputs:** Audio stream buffers labeled as speech from Audio Triage. For each detected utterance, the audio is streamed to the STT API. Input audio format should meet Google's spec (16-bit linear PCM, 16kHz or 48kHz).
- **Outputs:** The primary output is the **transcribed text** of the command. Additionally: - Partial (interim) transcripts output in real-time as the user speaks (with a flag indicating they are not final). - Final transcript output (with high confidence and possibly a list of alternative transcriptions if provided by the API). - A confidence score or stability measure if available. Google STT v2 returns a confidence for final results and a stability for interim results. The module can pass these along for NLU or for the HITL UI to display if needed.
- **Trigger/Flow:** Triggered when Audio Triage signals "speech detected". The module opens a gRPC or streaming REST connection to Google Cloud STT. Audio is sent continuously; partial texts are received and final text at end. Once final text is obtained, it triggers the NLU module (FR-4.1). The module then either closes the stream or listens for the next utterance if using an infinite stream approach. - **External Dependencies:** Google Cloud Speech-to-Text API v2 is the core dependency. The system will use Google's **streaming recognition** endpoint (likely via gRPC for efficiency). The AI agent should incorporate Google's SDK or client libraries (e.g., the Python `google-cloud-speech` library or REST calls) to send audio and receive transcripts. Internet connectivity and Google credentials are required. Additionally, the module depends on parameters like the STT model used (`latest_long` vs `latest_short`). We will use `latest_long` or `long` model to allow continuous streaming. - Note: Using cloud STT implies a dependency on network latency. However, Google STT in streaming mode is low-latency and returns partial results quickly (often within a few hundred milliseconds for first words). - **Success Criteria:** - **Transcription Accuracy:** For clearly spoken commands, the text should match the spoken words with high fidelity (goal

WER < 10%). Important command keywords (like “next”, “previous”, “play”, song names) must be recognized correctly nearly all the time. We measure success by testing a set of sample commands; >= 95% of commands should be transcribed without critical errors (i.e., errors that would change the interpreted intent). - **Latency:** Partial transcripts available with <300ms latency per word, final transcript for a 3-second utterance available within <0.5 seconds after speech end. Overall, from end of user speaking to final text output should be <0.5s on average (this allows overall voice command execution within ~1s including NLU and action). - **Robustness:** The streaming session should handle at least 5 minutes of continuous operation without reset (to cover many commands in a sequence). If the stream is dropped or errors out (network glitch), the module should automatically retry or re-establish connection (see fault tolerance design). No memory leaks or buildup should occur over long sessions. - **Cost consideration:** The AI should manage streaming efficiently to minimize cloud usage (e.g., don’t keep it running during long silences unnecessarily – perhaps use VAD to suspend sending audio when truly silent to reduce billed time). - **Suggested Tech Stack:** Use the **Google Cloud Speech-to-Text v2 API** via official client libraries. For example, Python with `google-cloud-speech` library or Node.js with the Google Cloud SDK. gRPC streaming is preferred for real-time. The module can be implemented as a service that accepts audio from the triage module and handles the API calls asynchronously. Techniques like audio chunk queuing, threading or async IO will be used to ensure the audio feed is continuous. Partial results handling logic will be in place to dispatch early intents. The AI might also incorporate **speech adaptation** by dynamically updating the request with hints (e.g., a list of current song titles or speaker names that it expects).

Table: Speech Transcription Module Summary

Feature	Description
Inputs	Audio stream of detected speech (live microphone data in short frames).
Process	Streaming speech-to-text via Google STT v2 (long-form model). Partial and final transcripts generated.
Outputs	Text transcripts of commands (interim & final) with confidences.
Trigger	Activated on voice command detection (speech start). Runs until speech end detected.
External APIs	Google Cloud Speech-to-Text v2 (streaming gRPC).
Success Metrics	WER < 10%; Final result latency <0.5s after utterance; partial results ~<300ms for first word.
Tech Stack	Google Cloud SDK (Python/Node), gRPC streaming, interim results handling for low latency.

Music Identification Module

Description: When the Audio Triage module detects music (e.g., an instrumental or someone singing a song), the Music Identification module’s job is to **identify the song or audio clip**. This is done via audio fingerprinting using the ACRCloud service. By sending a short sample of the audio, ACRCloud can match it against its music database (or a custom database of tracks) and return the track information (song title, artist, etc.). In the Kairos context, identifying a song can allow the system to automatically pull up the corresponding presentation (e.g., lyrics slides) for that song without a spoken command. This module is

crucial for “unspoken” cues – for example, if a band starts playing a song, Kairos can recognize it and display lyrics proactively.

Functional Requirements (FR-3.x):
- **FR-3.1: Audio Fingerprinting Trigger** – The system **shall capture an audio sample for fingerprinting** as soon as music is detected. Typically, ACRCloud requires about ~5-10 seconds of audio to reliably identify a song. The module should record a sample of the music (if continuous streaming is available, it can take a moving window). It shall only trigger identification when a sufficiently clear segment of music is present (to avoid sending noise or very short clips). If the music continues, the module can periodically attempt identification until a result is found or a timeout occurs (to cover cases where the first 10s might not be enough if there’s crowd noise, etc.).
- **FR-3.2: Song Identification via ACRCloud** – The module **shall call the ACRCloud recognition API** with the captured audio sample to identify the content. The request includes the audio fingerprint (or raw audio) and uses either ACRCloud’s global music database or a custom reference database. The response should include metadata like **song title, artist name, album, etc.** If the confidence of match is high (ACRCloud returns a score), the module treats it as a successful identification. If not, it may retry with a longer sample or indicate failure to identify. ACRCloud is capable of identifying both exact recordings and, if configured, cover versions or live performances (through their “cover song identification” feature). The system should enable cover detection to handle live music and humming.
- **FR-3.3: Response Handling and Mapping** – Upon identification, **the module shall map the identified song to the corresponding media content in the system**. This likely involves looking up a mapping of “song title/ID -> ProPresenter presentation” in the system’s content database (populated via the Data Ingestion Portal). For example, if ACRCloud returns “Amazing Grace – Artist XYZ”, Kairos should find the internal entry for “Amazing Grace” lyrics and retrieve its reference (like a presentation name or unique ID in ProPresenter). If multiple versions exist, the system may default to a known preferred version or prompt the operator via the HITL UI to choose. If the song is recognized but not found in the content library mapping, this is a special case (FR-3.4).
- **FR-3.4: Unknown Content Handling** – If the music cannot be identified by ACRCloud (no match with sufficient confidence), or it’s identified as a song that is **not** in the prepared library, the module shall handle it gracefully. It should: - Log an “unidentified song” or “unmapped song” event. - Optionally, display a message on the HITL Verification UI such as “Unknown song detected – no action taken” or even ask the operator if they recognize it and want to manually trigger something. - The system should **not stall** other operations; it can continue listening for other inputs. - This scenario could feed into continuous learning: later, the operator might add this song to the library via the portal (the system could store a sample or the ACRCloud result for reference). - **FR-3.5: Performance** – The identification should be done quickly enough to be actionable. For example, if a song starts and lyrics are needed, identifying within the **first 5–8 seconds** is ideal so lyrics can come up by the first verse. Thus, the module shall be optimized to send the recognition request immediately when a good sample is available and handle the response as soon as it returns (ACRCloud typically responds within a second or two for fingerprint match). The module may operate asynchronously so as not to block the main thread – it can spawn a separate process or thread to handle the fingerprinting and API call, then callback with the result.

Inputs & Outputs:
- **Inputs:** Audio stream or buffered segment identified as “music” by Audio Triage. Likely the module will get a continuous feed of audio frames marked as music. It needs to accumulate enough of these frames (for example, 10 seconds worth) into a buffer for sending to ACRCloud. It might also downmix to mono and use required sample rate (ACRCloud typically uses 8kHz–48kHz audio for fingerprinting). - **Outputs:** - If

identification is successful: **Song metadata and a content lookup result.** E.g., "Song: *Amazing Grace* by XYZ – mapped to Presentation ID 123". This output will trigger the Presentation Orchestration module (FR-5.2) to display the corresponding content. - If identification fails or is inconclusive: an **error or null result** output. Possibly a notification to HITL UI for awareness. The system will then not take automated presentation action (to avoid wrong lyrics). - The module might also output a confidence score or multiple candidate matches if provided. If multiple candidates (e.g., two songs with similar audio), it could either choose the top one if confidence gap is large or ask for human confirmation via HITL. - **Trigger Conditions:** Triggered when Audio Triage flags an onset of music. It continues to monitor/record as music continues. If music stops before identification (e.g., song was too short or cut off), it may cancel the attempt. - **External Dependencies: ACRCloud API/SDK** – The system will use ACRCloud's **Identification API**. The AI agent should integrate the ACRCloud SDK (they have REST API and SDKs for various languages) to send an audio fingerprint. Credentials (access key/secret and host) from an ACRCloud project are required. The project may be set up with: - ACRCloud **Music DB bucket** (for known published songs) and/or a **custom content bucket** for any user-provided tracks (via Data Ingestion Portal). - The service is cloud-based, so it requires internet connectivity. The typical recognition uses ~10s audio and returns a JSON with the match. - Optionally, if extremely low-latency identification were needed, an on-premise fingerprint library could be considered, but here we rely on ACRCloud cloud. - **Success Criteria:** - **Identification Accuracy:** For songs that are in the reference database (either global or custom), the module should identify the correct song at least **90%** of the time within the first 10 seconds of audio. Cover songs or live renditions should be identified provided the cover detection is enabled (ACRCloud claims robust cover/humming identification when configured). Success means the correct title is returned as the top result. - **Identification Speed:** The end-to-end time from music start to receiving identification result should ideally be $\leq$ **5 seconds** for typical cases (this includes ~2-3s of audio capture plus ~1-2s processing). It should not exceed 10 seconds for a result (or else lyrics might appear too late to be useful). - **Low False Positives:** If a wrong song is identified (false match), that can be disruptive (displaying wrong lyrics). The system should favor "no result" over a wrong result if confidence is low. Thus, if ACRCloud confidence < threshold (say 70%), the system treats it as unrecognized to avoid false positives. Our goal is false positive identifications < 5%. - **Resource Use:** The module should not flood the API. It should send at most one recognition request at a time. If songs are back-to-back, it can handle multiple per session, but each with a fresh request. - **Suggested Tech Stack:** Use ACRCloud's **HTTP API or SDK**. For example, a Python implementation might use the `acrcloud` SDK which can generate fingerprint from raw audio and send to their endpoint, or directly use their HTTP endpoint with the audio data base64 encoded. The agent should handle constructing the request in the required format (including signing with the access key/secret). This module could be a separate microservice or a thread that accepts audio chunks and manages the fingerprinting. Technology like **FFmpeg/ACRCloud provided libs** can be used to preprocess audio. For mapping song to content, a simple database (SQL or even a JSON file loaded in memory) can be queried – the Data Ingestion Portal will populate that. The mapping could be cached in memory for speed. The agent should ensure the system has an efficient lookup (e.g., dictionary keyed by song title, or a more robust key if provided like ISRC). ACRCloud often returns standard IDs (like Spotify or ISRC codes), which could be used as keys to map to local content if the portal stores those.

Table: Music Identification Module Summary

Feature	Description
Inputs	Buffered audio segment (~5–10 seconds) when music is detected.

Feature	Description
Process	Audio fingerprinting and recognition via ACRCloud. Cover song/humming mode enabled for live performance identification.
Outputs	Song identification result (title/artist) with confidence. Triggers content mapping to presentation.
Trigger	On music onset (no concurrent speech). Possibly continuous monitor if music persists.
External API	ACRCloud Identify API (cloud). Uses project keys and music database.
Success Metrics	≥90% correct song ID within 5-10s; false ID <5%. Unrecognized if confidence < threshold.
Tech Stack	ACRCloud SDK/REST (Python requests or Node fetch); background thread for API call; uses data mapping from system DB.

Natural Language Understanding (NLU) Module

Description: The NLU module takes the transcribed text from the Speech Transcription module and **infers the intended command or action**. It transforms raw text into a structured **intent** with any relevant parameters (entities). Kairos will use Google Dialogflow CX as the NLU engine, leveraging its conversational intent recognition capabilities. Dialogflow CX will be configured with a set of intents corresponding to the possible voice commands (e.g., "Next Slide", "Previous Slide", "Go to Song [Name]", "Pause", etc.). The NLU module may also incorporate additional logic for things that Dialogflow might not cover (but ideally Dialogflow handles all command understanding). Once the intent is determined, the module outputs a standardized intent object that the Presentation Orchestration module can act upon.

Functional Requirements (FR-4.x):
- **FR-4.1: Intent Detection** – The system **shall parse each transcribed command to identify the user's intent**. This is done by sending the text to Dialogflow CX's **Detect Intent** API (or using its client integration). The Dialogflow agent will be pre-built with intents such as: - *"NextSlide"* – triggered by phrases like "next slide", "go to the next slide", etc. - *"PreviousSlide"* – phrases like "previous slide", "go back", etc. - *"GoToSong"* – phrases like "show [song name]", "pull up [song title] lyrics". - *"PlayPauseMedia"* – e.g. "pause video", "play audio". - *"StopBackground"* – e.g. "stop music" (maybe to clear audio). - *"ShowContentByName"* – e.g. "show announcement [name]" or generic content retrieval. - etc., according to the needs of the media control (these are examples; see Intent Mapping table below for details).

Dialogflow CX will return the **matched intent name** and any **parameters** extracted (e.g. the song name in a "GoToSong" intent). The NLU module shall handle this response and ensure we have a valid intent. If the NLU confidence is low or no intent matches, that is handled in FR-4.4.
- **FR-4.2: Entity Extraction and Context** – For intents that require additional info (entities), the NLU must extract them. For example, if the command is "Show *Amazing Grace*", the intent might be "GoToSong" and the entity `song_title = "Amazing Grace"`. The Dialogflow CX agent will have **entity types** defined (like SongName, maybe SectionName for parts of a presentation, etc.). The system shall capture these parameters. The NLU module should also handle simple context or follow-ups. For instance, if a command is ambiguous like "show next verse", the system might use context (which song is currently showing) to interpret that properly. Dialogflow CX can manage context between intents if configured (e.g., have the

concept of a "current song" stored as a parameter context). The requirement is that sequential commands that depend on prior state are understood correctly.
- **FR-4.3: Early Intent Hint (Streaming Integration)** – (Advanced/optional) If the Speech module provided a partial transcript that strongly indicated an intent (via FR-2.3), the NLU module shall be capable of using that early partial to begin processing. This could mean calling Dialogflow CX even before the final transcript if a high-confidence partial command is available. This requirement supports the latency minimization principle: the system can potentially initiate the media action slightly before the user finishes speaking. For example, if partial text = "next slide" while user is still saying "please", the NLU can get intent=NextSlide and trigger the action. This must be done carefully to avoid misfires (only do for simple unambiguous commands). The NLU module should be thread-safe to handle possibly two calls (partial and final); it should ensure that a final transcript can override any action triggered by a partial if there was a discrepancy.
- **FR-4.4: Unrecognized Command Handling** – If the NLU module cannot map the transcript to a known intent (e.g., Dialogflow returns a fallback intent or very low confidence), the system should handle this gracefully: - It should not execute anything (since it doesn't understand). - It will log the event and show an "unrecognized command" alert on the HITL UI. - If possible, it might reprompt or clarify via the operator (the AI itself likely wouldn't ask the user directly in a live setting, but the operator could be alerted to intervene). - The text could be recorded for later analysis – possibly to expand the NLU training data if this turns out to be a legitimate new command phrase. - **FR-4.5: Intent Confirmation (for uncertain cases)** – If the NLU confidence is below a certain threshold or the intent has an entity that is not certain (e.g. song name that was transcribed ambiguously), the module shall flag that the interpretation is uncertain. In such cases, before triggering the action, the system could either: - Consult the HITL Verification UI for confirmation (e.g., "Did you mean to go to song 'Grace' or 'Gracefully'? Please confirm."). The operator can then select the correct intent or correct the parameter. - Alternatively, if no human is available (in fully autonomous mode), the system might use a simple strategy like picking the top guess but being ready to rollback if signaled.

By default in our design, we assume a human operator is present, so the safe approach is to require confirmation if confidence < threshold (say < 0.7). This is a design choice to ensure reliability.

Intent Mapping: The table below outlines key intents the NLU should handle, including example utterances and the expected action. (This is an example mapping that would be implemented in Dialogflow CX.)

Table: Kairos Intent Mapping (Examples)

Intent Name	Example Utterances	Parameters	Action (Outcome)
NextSlide	"Next slide", "Go forward", "Next"	*(none)*	Advance to the next slide in the current presentation.
PreviousSlide	"Previous slide", "Go back", "Back one"	*(none)*	Go to the previous slide in the current presentation.
GoToSong	"Show <*SongName*>", "Go to <*SongName*> lyrics"	SongName (title of song)	Load the specified song's presentation and display the first slide (lyrics) of that song.

Intent Name	Example Utterances	Parameters	Action (Outcome)
GoToSection	“Show <*SectionName*> of the song”	SectionName (e.g. verse 2, chorus) + (implicit current song context)	Jump to the specified section (slide group) of the current song presentation. Requires that a song is already active or specified.
PlayPauseMedia	“Pause video”, “Play clip”, “Resume audio”	*(maybe MediaName)*	Control media playback (if a video or audio track is currently playing, toggle play/pause). If a specific media name is given, target that; otherwise, affect global playback.
ClearScreen	“Clear screen”, “Clear lyrics”	*(none)*	Remove all live output from the screen (e.g., clear slides to black/no content).
ShowAnnouncement	“Show announcement <*Name*>”, “Show slide <*Name*>”	Name (of announcement or slide deck)	Switch to a predefined announcement or slide deck identified by Name. For example, display a specific announcement slide.
StopBackground	“Stop music”, “Mute background”	*(none)*	Stops any background audio playback (could send a clear audio command in ProPresenter, or mute the audio channel).
Fallback	(catch-all for unrecognized)	N/A	No direct action – will trigger a clarification or no-action.

Note: The above intents would be configured in Dialogflow CX with training phrases capturing various ways users might say them. The **GoToSong** and **ShowAnnouncement** intents illustrate use of a parameter; Dialogflow will have an entity type for SongName (perhaps a list of known song titles) and similarly for announcements. The NLU’s ability to recognize those entity values may benefit from providing a lexicon of known titles (which can be updated via the Data Ingestion Portal when new songs are added).

Inputs & Outputs:
- **Inputs:** Final (or partial) transcript text of a spoken command. This comes from the Speech Transcription module. Additionally, context such as the current state (e.g., which song or presentation is currently active, if needed for context-aware commands) can be given to Dialogflow as parameters or session context. - **Outputs:** A structured **Intent object** that includes: - **Intent name** (as defined in the Dialogflow CX agent). - **Parameters/entities** (key-value pairs, e.g. `"song_title": "Amazing Grace"` for a GoToSong intent). - **Confidence or detection score** (if provided by the NLU – Dialogflow CX provides a fulfillment status but not

a direct numeric score in all cases; we may infer confidence from matched intent or some custom logic). - Possibly an indication if this was from a partial utterance.

This output is then fed to the Presentation Orchestration module (FR-5.1) to carry out the action. If the intent is a fallback/unrecognized or requires confirmation, the output would include that status, prompting either a no-op or a request to HITL UI. - **Trigger:** Whenever a transcript is ready. The NLU module is stateless per query but might maintain a Dialogflow session for context. The AI agent will likely maintain a **Dialogflow session ID** across a series of commands in one event so that contexts can be used (Dialogflow CX contexts/pages can handle multi-turn if needed). - **External Dependencies: Dialogflow CX agent** – This requires that the agent (with all intents and entities) is designed and deployed on Google Cloud. The NLU module will use Google's Dialogflow CX API to detect intent. The agent configuration is a deliverable of the AI's design (the AI might set it up via API or require a one-time manual setup based on the FRD specs). The module will use either Dialogflow's REST API or client library (which might internally call gRPC). An internet connection and appropriate authentication (service account or API token) are needed. Additionally, the NLU depends on the content of the Data Ingestion Portal: for example, the list of song names (for entity extraction) should be synchronized. This could be done by periodically updating the Dialogflow entity for SongName via their API when new songs are added. - **Success Criteria:** - **Intent accuracy:** The NLU should correctly identify the intended action from the user's command in $\geq$ 95% of test cases (for well-phrased, in-scope commands). That means the correct intent and parameters are returned. Misclassification rate should be very low, especially for critical commands like "Next/Previous" which are simple. - **Parameter extraction accuracy:** If a user says a known song name, the system should extract it correctly and exactly. The use of Dialogflow entities and perhaps spelling correction in STT should ensure that e.g. "Reckless Love" is captured as such, not "Reck less love". The expectation is that for known phrases and titles, accuracy is high ($\geq$ 90%). Unusual or new names might fail, but the system then should either not guess or ask for confirmation. - **Latency:** The intent detection call to Dialogflow should add minimal delay. Dialogflow CX API calls are typically fast (<300ms). Combined with the streaming STT, the overall pipeline (speech -> text -> intent) should ideally complete within ~0.5–0.7 seconds of speech end. The NLU itself should be on the order of a few hundred milliseconds. - **Robustness:** The NLU module should handle slight transcription errors gracefully. For example, if STT misheard "Go to next slide" as "Go to neck slide", ideally the intent model still maps that to NextSlide. Using lots of training phrases and possibly enabling **fuzzy matching or synonym lists** in Dialogflow can help. The continuous improvement loop will involve adding any common misrecognized phrasing into the training data. - **Fallback handling:** In cases of no match, the system does nothing unintended (no false triggers). It's preferable to not understand (and log it) than to perform the wrong action. Hence, fallback intent should capture all unknown utterances and ensure they don't proceed to orchestration without human oversight. - **Suggested Tech Stack:** Integration with **Dialogflow CX** via REST/gRPC. The AI agent can use Google's Python SDK (`google-cloud-dialogflow-cx`) or make direct REST calls to the `detectIntent` endpoint with the session and text. This module will likely be part of the same process as speech or separate microservice. For simplicity, the AI could design it as a function call in the main application that sends text and gets intent. We will ensure the Dialogflow agent is configured appropriately: - Intents and entities as described. - Perhaps webhooks for complex logic (though likely not needed if intents directly map to actions). - The technology to store context (Dialogflow pages/flows or session parameters) if needed for multi-turn or context-aware commands. - If Dialogflow is unavailable for any reason, as a fallback the system could use a simpler built-in intent matcher or default mappings, but since DF is core, we assume high availability.

Table: NLU Module Summary

Feature	Description
Inputs	Transcribed text from STT (final or high-confidence partial).
Process	Intent classification via Dialogflow CX agent. Extract intent name and parameters (entities).
Outputs	Structured intent (name + parameters). Possibly confidence or require-confirmation flag.
Trigger	On availability of a transcript (each voice command utterance).
External API	Dialogflow CX DetectIntent API (Google Cloud). Uses predefined agent with Kairos intents.
Success Metrics	≥95% correct intent recognition; ≥90% correct entity extraction for known items; NLU processing <300ms latency.
Tech Stack	Dialogflow CX client (Python/Node), using session context. Sync with content DB for entity lists.

Presentation Orchestration Module

Description: The Presentation Orchestration module is responsible for **executing the desired media control actions** on the actual presentation software (ProPresenter) and related systems, based on the intents from NLU or the results from Music ID. Essentially, this is the module that "presses the buttons" in software that a human operator would normally press: advancing slides, selecting presentations, playing or pausing media, clearing screens, etc. It acts as the **central command dispatcher** that receives an intent and then interacts with ProPresenter's API (and possibly other subsystems) to carry it out.

This module also handles coordinating multiple outputs if needed – e.g., if a "GoToSong" intent comes from either voice command or music detection, it ensures the correct presentation loads and notifies if any manual step is needed.

Functional Requirements (FR-5.x):
- **FR-5.1: Execute Intent Actions** – The system **shall map each recognized intent to one or more ProPresenter control commands** (or internal actions) and execute them. For example: - *NextSlide/ PreviousSlide:* send the ProPresenter command to go to next or previous slide in the active presentation. - *GoToSong:* find the presentation corresponding to the song title (from NLU parameter or Music ID result) and load/display it. This might involve selecting a specific playlist or library item in ProPresenter and triggering it to go live. - *GoToSection:* within the currently displayed song, jump to the specified section (e.g., if sections are labeled in slides, possibly by sending a cue to jump to slide number or using ProPresenter API if it allows selecting slide by index). - *PlayPauseMedia:* if a media (video or audio track) is currently playing or paused, toggle its state. ProPresenter API has commands for triggering media playback or pause. - *ClearScreen/ClearLyrics:* send command to clear certain layers (ProPresenter supports clear commands for various layers – text, media, etc. For lyrics, likely clear slide content). - *ShowAnnouncement:* switch to an announcement presentation by name – likely by selecting it from an announcement playlist and triggering it. - *StopBackground:* mute or stop background audio – possibly this is just another clear audio command or a custom integration if background music is from another source.

For each intent, the orchestration module knows the sequence of API calls or actions to perform in ProPresenter (or the system). These actions should be executed **atomically if possible** (or in a correct order). After sending commands, it can optionally verify success (if the API returns acknowledgments or by querying state – see FR-5.4).

- **FR-5.2: Autonomous Song Display (via Music ID)** – The module shall also handle triggers coming from the Music Identification module. If a song is identified (with high confidence), it effectively creates an implicit "GoToSong" intent. The orchestration module should treat it as such and load the corresponding song lyrics presentation **without a voice command**. This means the system should allow two pathways to trigger presentation: NLU (explicit voice request) and Music ID (implicit cue). The logic may include a check like: if a song is already showing or a manual override is in place, do not override it. But if the screen is clear or showing something else and a new song starts, go ahead and display it. This should all happen seamlessly so that by the time the congregation starts singing, the lyrics appear.
- **FR-5.3: Timing and Smooth Transitions** – The orchestration should aim to not cause jarring jumps. For instance, if the command is NextSlide, it should press it exactly once (no overshoot). If multiple commands come in quickly (like a user says "next, next, next" or music triggers a new item while another is on), the module should queue or prioritize actions in a logical way: - Potentially, if two different actions conflict (unlikely from one user utterance, but consider if an automated music trigger "GoToSong A" comes at same time operator said "NextSlide" for current song), the system may need a priority or to ask confirmation. Likely human voice commands have priority over automatic music triggers. - Transitions: If ProPresenter is in the middle of a transition effect, additional commands should possibly wait until it's done to avoid glitches (depending on ProPresenter's capability – it might queue internally). - The requirement is the module should manage the sequence of operations to ensure the presentation output looks professional (no flashes of wrong content, no double triggers). - **FR-5.4: Acknowledgement and Error Handling** – Whenever a command is sent to ProPresenter (which we'll do via its API), the system should handle the response: - If the API confirms success (some commands may not have explicit ack beyond maybe a WebSocket echo or no error), then great. - If there is an error (connection lost, command not executed), the module should detect this (perhaps via lack of expected response or via an error code) and log it. It should also notify the HITL UI if something fails (e.g. "Failed to advance slide, please check connection"). - Additionally, for certain critical actions, the module can verify state. For example, after a "GoToSong" command, it could query the current presentation name via the API to ensure the correct one loaded. This might be supported by ProPresenter's API (if it provides state messages). - If verification fails or state is not as expected, a retry mechanism (FR-5.5) or human alert should be invoked. - **FR-5.5: Retry Logic for Commands** – The module **shall implement retry logic** for sending commands to ProPresenter in case of transient failures. For instance, if the WebSocket connection momentarily drops or if a command is sent too early (before ProPresenter is ready), the system should attempt it again up to a defined limit. E.g., if "NextSlide" gets no ack, try again in 0.5 seconds. However, it must guard against sending duplicate commands if the first actually succeeded (to not double-advance). This might involve maintaining a simple state of last command and whether confirmation was seen. - In general, network commands could be retried 2-3 times on failure with a short backoff. If still failing, escalate to human (maybe the operator needs to check ProPresenter). - **FR-5.6: Multi-Interface Control (if applicable)** – Currently, we focus on ProPresenter. If in future, Kairos needs to also control other systems (e.g., lighting or sound via MIDI, etc.), the Orchestration module would be the place to integrate that. For completeness: if an intent required controlling something beyond ProPresenter, this module could call another API or send a MIDI command. (For now, we assume all relevant media control is through ProPresenter which itself can send triggers to other systems if configured).

Inputs & Outputs:

- **Inputs:** The **Intent object** from NLU (or a "song identified" event from Music ID). This includes the action to perform and any parameters needed (like which song, which section, etc.). The module might also take inputs from the Data Portal (e.g., the mapping of song name to presentation path, which it needs to find the right item). - **Outputs:** - The direct outcome is the **state change in ProPresenter** (which is not a data output but an effect: e.g., slides actually change). - The module can also output a status message indicating success or failure of the action (which can be logged or displayed on the operator UI). - For example, after executing, it could emit "ActionExecuted(NextSlide)" or "ActionFailed(ShowAnnouncement: name not found)". - If an action required some result (like getting a list of sections, etc.), it could output that back to some context, but mostly it's one-way control. - **Trigger:** Triggered every time an intent or automated trigger comes in. It's stateless in that each command is handled independently, but it may maintain a persistent connection to ProPresenter's API. Typically, the module will start a connection to ProPresenter at system startup (WebSocket handshake) and keep it open to send commands on demand. - **External Dependencies: ProPresenter 7 Control API** – This is the main interface. ProPresenter 7 (v7.9+ as assumed) offers an API via WebSocket (and HTTP in some cases). Specifically: - We will use the **Remote Control WebSocket** channel of ProPresenter [1]. Once connected and authenticated (ProPresenter requires an auth message with a password token if set), we can send control messages. According to documentation, it's a text-based protocol where certain commands (like "presentationTriggerIndex" or similar) are sent to trigger slides [2] [3]. - The AI should follow the ProPresenter API documentation for the exact message formats. For instance, to trigger the next slide, one might send a JSON like `{"action":"presentationTriggerNext","presentationPath":"...", "slideIndex": ...}` or perhaps a simpler shortcut if exists. - The ProPresenter API also allows HTTP GET/POST for some actions (as seen in an OpenAPI snippet), but WebSocket is likely more real-time and bi-directional. - The Orchestration module depends on maintaining the **TCP/IP connection** to ProPresenter. If that connection fails, the module should attempt to reconnect (and alert human if it cannot). - Additionally, it might depend on the **content mapping** from the Data Portal for looking up presentation filenames or indices. If ProPresenter has a predictable structure (like the song library in a playlist), the agent might also use known naming conventions. - **Success Criteria:** - **Timely execution:** Commands should translate to visible results in ProPresenter **immediately (within <200ms)** after issuing (ProPresenter itself is quite responsive locally). For the user perspective, from voice command spoken to slide change visible, target ~1 second or less. The orchestration part of that should be negligible (a few ms to send command). - **Accuracy of action:** The correct slide/presentation is shown. E.g., if the intent was NextSlide, exactly one slide advance occurs. If GoToSong "Amazing Grace" was intended and that song is in library, the correct "Amazing Grace" slides start showing. Success means no wrong content is triggered. Measured by testing each intent end-to-end. - **Reliability:** 99+% of commands result in the action happening as expected on first try (with retries being hidden from user if needed). In rare cases of failure (lost connection), the system recovers quickly (reconnect and perform action) or alerts the operator. - **Non-interference:** If a human is manually controlling ProPresenter in parallel (just in case), the AI's commands should not conflict (we assume normally AI is primary controller, but even if manual actions happen, AI should read current state if possible and act accordingly). - **Safety:** The module should avoid any actions outside the defined scope – it should not trigger anything not explicitly commanded by an intent or automated rule. This is important for governance; we ensure it only does what it's supposed to. - **Suggested Tech Stack:** This module will likely be implemented in the same application process, using a **WebSocket client library** to communicate with ProPresenter. For example, in Python one could use `websockets` or `websocket-client` library. In Node, the `ws` library. The agent will code the handshake (including required headers like `Sec-WebSocket-Key`, etc., as ProPresenter expects a certain handshake format [4] and possibly an auth JSON message once connected). After connection, sending commands might be as simple as sending certain

structured messages (maybe JSON or a specific syntax as per ProPresenter's protocol). The AI could utilize existing wrappers; e.g., the community documentation shows some message types (like `presentationTriggerIndex` etc.). If the OpenAPI is accessible, possibly using HTTP POST endpoints could complement (but likely everything can be done via WebSocket). - The tech stack also includes managing that socket connection in an asynchronous manner (listening for any incoming messages too; the API might send updates, which we might not strictly need, but could use for verification). - For mapping content: possibly maintain a dictionary of "SongName -> ProPresenter Library Path/Identifier" loaded from the Data Portal's database. - Logging and error catching around each command send is important to implement here (to implement retries and error reporting).

Table: Presentation Orchestration Module Summary

Feature	Description
Inputs	Parsed intent (from NLU) or auto-trigger (from Music ID) with action details.
Process	Maps intent to ProPresenter API commands; sends commands via WebSocket/TCP to control slides/media.
Outputs/ Effects	Change in presentation state (slide advanced, new presentation shown, media paused, etc.). Also status feedback (success/fail logs).
Trigger	On every recognized intent or identified song event.
External Interface	ProPresenter 7 Remote Control API (WebSocket) [1] ; requires network connection and auth to ProPresenter instance.
Success Metrics	100% correct action execution for valid intents; command latency ~<200ms; robust reconnection & ≤3 retries on failure.
Tech Stack	WebSocket client (Python `websocket-client` or Node WS). JSON command structures per ProPresenter API. Uses content mapping DB for selecting presentations.

Data Ingestion Portal

Description: The Data Ingestion Portal is a web-based module that allows human operators (or the AI agent in an assisted manner) to **input and manage the media content and reference data** that Kairos uses. This includes uploading new song lyrics, associating songs with audio for recognition, and managing any configuration like Dialogflow training data or entity lists. This portal is crucial for **preparing the system** ahead of an event and for incremental learning – it's where humans ensure the AI has the latest content. It also provides an interface for managing the AI's knowledge base (like adding new commands or adjusting settings) in a controlled way.

Functional Requirements (FR-6.x):
- **FR-6.1: Content Upload (Songs/Presentations)** – The portal **shall allow users to upload or enter new song content**. For example, when the church plans a new song, the operator can: - Upload the ProPresenter presentation file for that song's lyrics (or create slides within ProPresenter as usual, then ensure the portal knows about it). - Enter the song title and any alternate names (for NLU purposes). -

Optionally, upload an audio sample or MP3 of the song if they want to improve recognition. The portal would then call ACRCloud's API to fingerprint this and add it to a **custom songs database**. Alternatively, if relying on global DB, maybe skip audio upload unless it's a very unique local song. - Once added, the song appears in the system's library list with a mapping of *SongName -> Presentation ID/Name* (and possibly an ACRCloud track ID). - The portal should validate entries (e.g., require a unique name, confirm file formats, etc.).

- **FR-6.2: Content Management & Catalog** – The portal **shall list all existing content items** known to the system (songs, announcement slides, backgrounds, etc.). The operator can browse and search this list. They can edit entries (e.g., correct a song title, or update the mapping if a presentation file changed). They can also remove/deprecate old songs. This ensures the AI's internal mapping and NLU entity lists are up to date. The portal might display status like whether each song has an associated fingerprint in ACRCloud and if not, allow adding one. - **FR-6.3: NLU Configuration Management** – The portal may provide interfaces to manage aspects of the NLU: - Manage the list of known command phrases or synonyms. For instance, if operators find people use a phrase that wasn't understood, they could add it to the relevant intent's training data via the portal (which could then update Dialogflow through its API or at least log it for the AI dev to update offline). - Manage entity lists like SongName. E.g., when a new song is added, the portal should trigger an update to the Dialogflow CX entity for SongName (possibly by calling Dialogflow's API to update the entity with the new entry, so that the NLU instantly knows the new song). If automated update is complex, at minimum the portal exports a file for the AI to use in offline training. - Adjust thresholds or settings for the AI (like confidence threshold for HITL intervention) with governance oversight. Possibly an admin UI to set these values. - **FR-6.4: Operator Accounts & Permissions** – The portal should have authentication (only authorized staff can modify data). Different permission levels might be present (e.g., a content manager vs. a tech admin). This is for governance to ensure changes are intentional and traceable. All changes (adding content, editing, etc.) should be logged with timestamp and user. - **FR-6.5: Integration with ACRCloud and System DB** – When content is added or removed, the portal should **update the relevant external systems**: - If a new song is added and an audio file is provided, the portal will communicate with ACRCloud to add that track's fingerprint to the custom project (using ACRCloud Console API or identification API in a certain mode). - It will update the system's internal database mapping. Possibly, the system uses a simple database (like SQLite or a cloud DB) to store a table of `SongTitle, ProPresenterIdentifier, ACRCloudReference`. The portal writes to that DB. - If integrated, it could also update Dialogflow (as mentioned) by calling the CX API to add new entities or even new intents if that was ever needed (though new intents are less likely to be added often; command set is mostly fixed, but perhaps phrases). - **FR-6.6: Usability & Feedback** – The portal should be user-friendly, as non-technical staff might use it. It should provide guidance on what info is needed for each content item. After adding content, it can give feedback: e.g., "Song added successfully. Recognizable via voice commands and audio detection." If there are issues (like ACRCloud limit reached or API error), it should clearly notify the user so they can address it. - **FR-6.7: System Config & Monitoring (optional)** – The portal could also show some system status: e.g., whether the AI components are online, last time Dialogflow training data was updated, etc. This helps governance and debugging. Not a core requirement but useful for completeness.

Inputs & Outputs (for the Portal):

- **Inputs (from user):** File uploads (e.g. .propresenter bundles, .mp3 audio files), text inputs (song titles, descriptions), selections (categories, etc.), configuration values. - **Outputs (to user):** Web UI pages showing lists of content, confirmation messages, error messages if something fails (like "Failed to fingerprint audio – please try again"). - **Internal Data Outputs:** When user submits data, the portal writes to: - The **Kairos content database** (which could be an internal JSON or SQL DB). This stores metadata like song title, presentation name/path, etc. - The **ACRCloud service** (via API) to add or update the fingerprints as needed.

- The **Dialogflow CX agent** (via API calls) to update NLU data (if within scope to do automatically; otherwise, flag it for manual update). - **Trigger:** Human-initiated via web browser. This is not an always-running pipeline like other modules, but a on-demand interface used outside of live voice sessions (likely used during setup or rehearsals). - **External Dependencies:** - ACRCloud Console API for managing custom content (if using custom bucket). - Dialogflow CX Management API for updating entity values or training phrases (requires proper auth and possibly re-training the agent model after changes – Dialogflow CX might do it automatically when using the API). - A database system (could be a lightweight SQLite or a cloud Firestore etc., depending on deployment) to persist the content info. The portal will use a backend framework to interact with this DB. - **Success Criteria:** - **Data completeness:** All critical content for a service can be entered and verified via the portal ahead of time. Success would be measured by, e.g., if the operator can add 100% of the songs they plan to use and those songs are correctly recognized by the system during the event (this links to the actual runtime success but preparation is key). - **Ease of use:** Non-technical users can navigate and perform tasks without errors. This could be measured via user testing or simply by the absence of user error incidents. Ideally, adding a new song should take only a minute or two. - **Accuracy of sync:** After adding data via the portal, the underlying systems reflect that data. For instance, if a song "How Great Is Our God" is added, then: - The NLU will recognize "How Great Is Our God" as a SongName entity. - The Music ID will catch it (either because it's common and in global DB or because we added an audio reference). - The ProPresenter mapping is correct (when the orchestration tries to load it, it works). - We can test this end-to-end prior to event (the portal might include a "test this song" function). - **Reliability:** The portal must not crash or lose data. Content changes should be saved durably. Also, actions like uploading to ACRCloud or updating Dialogflow either succeed or clearly inform the user to retry if not. - **Suggested Tech Stack:** Likely a simple **web application** – could be a small section in an admin web app for Kairos. For example: - Frontend: HTML/CSS/JS (maybe a framework like React or just plain form-based). - Backend: Python (Flask/Django) or Node (Express) serving REST endpoints for create/read/update of content. The AI agent can set this up or at least scaffold it. - Database: SQLite or PostgreSQL for storing content metadata. Alternatively, even a structured YAML/JSON could suffice if loaded at startup, but DB is safer for concurrent edits. - The portal will call external APIs (ACRCloud, Dialogflow) via its backend when needed. We must secure those API keys on the server side. - Authentication can be done via simple username/password (managed in the system's config) or Google OAuth if integrated with GCP project, etc., depending on environment – details can be decided by governance (security requirement: restrict access). - The UI should run on the local network (perhaps accessible at a URL like http://kairos.local:5000) and be reachable by the operator's laptop.

Table: Data Ingestion Portal Summary

Feature	Description
Purpose	Web interface for human operators to upload and manage content and configuration (songs, slides, etc.) before/during deployment.
Key Functions	Add/edit/remove songs and presentations; upload audio for fingerprinting; update NLU entities; configure system thresholds.
External Hooks	Integrates with ACRCloud (add custom audio); Dialogflow CX (update entities); updates internal content DB.
Security	Authenticated access, logs all changes (audit trail for governance).

Feature	Description
Success Criteria	100% of needed content can be prepared via portal; new items recognized correctly during run; user-friendly and error-free operation.
Tech Stack	Web app (Flask/Django or Node/Express) with a database. REST calls to external APIs for sync. Runs on local server accessible to operator's PC.

Human-In-The-Loop (HITL) Verification UI

Description: The HITL Verification UI is a real-time dashboard for a human operator to **monitor and intervene in the AI's decisions** during live operation. While the goal is for Kairos to operate autonomously, this UI provides transparency and a safety net: the operator can see what the AI is hearing, interpreting, and doing, and they can step in to correct mistakes or approve uncertain actions. This fosters **trust and governance**, ensuring the AI doesn't run away with false commands. Over time, it also serves as a feedback collection tool – corrections made via this UI can be fed back into improving the AI (continuous learning).

Functional Requirements (FR-7.x):
- **FR-7.1: Live Transcription Display** – The UI **shall display the live transcribed text** of any speech the system picks up, in near real-time. For example, as someone speaks a command, the operator should see a text line appear (perhaps partial words updating) and then finalize. This allows the operator to verify if the speech recognition was accurate. It's essentially an on-screen log of "what the AI thinks was said." This helps catch if STT misheard something (operator can then anticipate a wrong intent).
- **FR-7.2: Intent and Action Visualization** – The UI **shall show the interpreted intent and planned action** for each command. For example, after a command is processed, it might show a line: "Intent: GoToSong (*Way Maker*) – Confidence 0.85 – **[Awaiting confirmation]**" or "Action: NextSlide (executed)". If the AI is confident and executes automatically, it can label it as executed. If not confident, it should indicate it's waiting for the operator. This transparent view helps the operator follow along with what the AI is doing.
- **FR-7.3: Confirmation & Override Controls** – For cases where **the AI requires human confirmation or correction (due to low confidence or ambiguous result)**, the UI shall provide interactive controls: - A **"Confirm"** button to approve the AI's suggested action. - An **"Edit/Select"** option to correct a misinterpretation. For example, if the AI heard a song title incorrectly, the operator could choose the correct song from a dropdown or type it. Or if two intents were possible, operator picks the right one. - A **"Cancel"** or "Ignore" button to tell the AI to do nothing for that command (if it was a false trigger or the operator handled it manually).

These controls ensure that before a potentially wrong action happens, a human can intervene. The AI should wait a configurable short period for confirmation if needed (e.g., if a decision is flagged as uncertain, maybe wait up to 2-3 seconds for an operator response; if none, it could either proceed with best guess or do nothing – we'll likely choose do nothing without confirmation in uncertain cases to be safe). - **FR-7.4: Real-Time Alerts & Escalation** – The UI should also highlight **any anomalies or failures**: - If the connection to an API is lost or a module fails (e.g., "STT connection lost" or "ProPresenter not responding"), an alert banner or indicator should show, so the operator is immediately aware and can take manual control if needed. - If an unrecognized command happens (fallback intent), it could show a message like "Unrecognized command: [transcript]. No action taken." Possibly with a prompt, "Should this have been something? (The operator could then decide to manually do something or note to train later)." - If the system identified a song automatically, it could show "Music detected: Song X – auto-displaying lyrics" so the operator knows it switched content by itself. - Essentially any time the AI deviates from normal (uncertain

interpretation, failures, or auto actions), it should be clearly signaled. - **FR-7.5: Logging and History** – The UI should maintain a **scrolling log of recent events** (with timestamps). This log might include entries like: - "Heard: 'next flight' (Transcription confidence 0.6)" - "NLU Intent guess: NextSlide (0.7 confidence) – NEEDS CONFIRMATION" - "Operator confirmed NextSlide – executed." - "Music ID: 'Way Maker' (confidence 0.92) – auto showing lyrics." - "Command executed: Cleared screen."

This history is useful for after-action review (governance can review what happened). It might also be saved to a file or database for audit. - **FR-7.6: Feedback Loop Integration** – When the operator makes a correction (through override or by confirming something different than the AI guessed), the system shall capture that as **feedback data**. For instance, if AI thought the song was "God is Here" but operator chose "God is Near", that information (audio snippet + correct label) can be logged for retraining STT or adjusting the language model. Similarly, if an intent was wrong, that could become a new training phrase example. This requirement ties into continuous learning: the UI is the touchpoint for humans to teach the AI in real time. The mechanism could be: - Log the incorrect transcript vs correct text (which can inform whether a custom word needs to be added to STT hints). - Log the mis-classified intent and what it should have been (to refine NLU). - Possibly provide an interface for operator to flag "This was wrong" explicitly if something goes awry that the AI didn't notice. - **FR-7.7: Non-Obtrusive Operation** – The UI should be designed to be **non-blocking** in the workflow. That is, if the AI is confident and everything is working, the operator shouldn't have to click "OK" for every command – it should just show them passively. Only when needed (uncertainty or error) should it demand attention with a prompt or highlight. This keeps the human workload low in normal operation (they can just monitor). The UI could have color-coded statuses (green for auto-executed actions, yellow for awaiting confirmation, red for errors) so the operator can at a glance see if anything needs action. - **FR-7.8: Performance** – The UI must update in real-time (with minimal lag). From the moment a word is transcribed or an intent is decided, the UI update should be within a few hundred milliseconds so the operator is seeing things as they happen. It should handle rapid events (though typically commands are sequential). The UI should be robust – if it disconnects or refreshes, it should be able to reload the latest state from the system so ongoing operation isn't hidden.

Inputs & Outputs:
- **Inputs (to UI):** Streams of data from the backend system: - Transcription text (perhaps via WebSocket from the backend to the web UI to achieve low-latency updates). - Intent results and confidence. - Triggers when an action is executed or needs confirmation. - System status signals (like error flags). - Essentially the UI is subscribed to various event topics from the Kairos backend. - **Outputs (from UI):** User actions: - Confirmations or corrections (which the UI sends back to the backend, e.g. via an API call or socket message, telling the Orchestration module it can proceed or telling NLU the corrected intent). - Possibly manual trigger buttons (the UI might also double as a mini controller – e.g., if operator wants to manually trigger next slide via UI, which could be convenient if voice fails. This is not primary but could be a nice fallback: a button that directly calls the orchestration action). - Feedback submissions (like a "report issue" button). - **Trigger:** The UI is essentially active throughout the live session. It's triggered by events from other modules; it also triggers back actions when operator interacts. - **External Dependencies:** Runs on a local network, likely served by the same backend as the Data Portal (or a related service). It could be part of the same web app but a different page/mode specialized for live monitoring. It likely uses **WebSockets or Server-Sent Events** from the backend to push real-time updates to the browser. No third-party external APIs needed at runtime except those already used by other modules. Possibly uses a library like Socket.IO if using Node, or Flask-SocketIO in Python, etc., to push events. - **Success Criteria:** - **Effective human oversight:** The operator is able to catch 100% of AI mistakes *before* they have negative impact (e.g., wrong lyrics displayed or wrong slide). This is subjective, but essentially if the AI makes an error and it's flagged for

HITL, the operator should catch and correct it in time. That implies the UI gave them clear notice and an easy control to fix it. - **Low intervention needed:** Over time, as AI improves, >90% of actions are auto-confirmed (green) and do not require user click. If currently a lot need confirmation, that's okay initially, but success means that ratio improves. This is more of a long-term metric. - **User satisfaction:** The operator feels in control and trusts the system because they can see everything. This can be measured via qualitative feedback. If something goes wrong, the UI should have logged it and given them some heads-up. - **Performance:** UI updates are real-time enough that operator never perceives a delay. They should see words appear as they're spoken (maybe slight delay but not seconds later). If partials are too fast to be useful to show all, maybe show final only or highlight partials less. - **Reliability:** The UI should not crash or freeze. It should handle network issues gracefully (if connection to backend lost, it should indicate "Disconnected" and try to reconnect). - **Security/Audit:** Only authorized personnel access it (maybe it's on a secure network anyway). All operator inputs are logged (so we know who overrode what). - **Suggested Tech Stack:** Likely integrated with the Data Portal's web server: - Could be a single-page web app (React or Angular) for dynamic updates, or a simple page with a bit of JavaScript to append logs. - Use WebSocket for pushing real-time events from backend to front-end. The backend (Kairos core) can emit events on certain channels (like transcription events, intent events). - E.g., if backend is Python Flask, use Flask-SocketIO; if Node, use Socket.IO; or use plain WebSocket protocol. - The UI will have interactive elements (buttons) which when clicked, send back to an endpoint (the backend then handles, e.g., if operator confirms an intent, the backend will proceed to call orchestration or if operator corrects a parameter, backend will override the NLU result with that parameter and then proceed). - The design should emphasize clarity: large text maybe for current command, color-coded highlights, etc. Possibly a table or list for events, with the most recent one at top. - It might also incorporate a "manual mode" toggle – if the operator turns off the AI (like a big "Kill Switch" if needed), that could be part of UI (for governance safety, one could include an emergency "disable AI" button that immediately stops Kairos from sending any commands, in case something is going really wrong).

Table: HITL Verification UI Summary

Feature	Description
Purpose	Real-time dashboard for human oversight, showing AI's hearing (STT), thinking (NLU), and actions (or proposed actions), allowing intervention.
Display	Live transcript text, detected intents with confidence, actions taken or awaiting confirmation, system alerts.
Interactivity	Operator can confirm or override intents, correct mis-heard entities, cancel actions, and possibly manually trigger actions if needed.
Feedback Loop	Operator corrections are logged and fed into system improvement (e.g., update training data later).
Performance	Updates in real-time (<200ms latency for UI updates). Non-blocking for operator unless needed.
Tech Stack	Web interface (could be part of Data Portal app). Uses WebSocket/SSE for live updates. Buttons send commands to backend (via REST or socket).

Guiding Design Principles for the AI Agent

In architecting and building Kairos, the agentic AI should adhere to several key design principles to ensure the system meets its performance goals and is robust and safe. These principles govern how the AI designs the pipelines and logic:

- **Latency Minimization:** The AI should design the system for **ultra-low latency** response to voice commands. Techniques include using streaming processing wherever possible and parallelizing tasks. For instance, the Speech Transcription is streaming, yielding partial results that allow early intent inference. The agent should ensure that modules are connected in a pipeline with minimal buffering. Early intent detection logic (acting on partial transcripts) should be implemented to shave off response time when possible (e.g., start loading the next slide as soon as "next" is recognized, without waiting for "slide please"). Additionally, using local processing for audio triage avoids sending unnecessary audio to cloud, saving time. The AI might also consider **audio framing and overlapping** techniques to not miss syllables at chunk boundaries. The overall design goal is that from a user saying a command to the action on screen, it feels instantaneous (aim ~1 second or less). Any opportunities to preprocess or predict user needs (e.g., preloading likely next songs if known from setlist) should be utilized to reduce on-demand latency.

- **Fault Tolerance & Retry Logic:** The AI must incorporate robust error handling in every module. This means detecting failures and recovering gracefully:

- External API calls (STT, NLU, ACRCloud) might fail or timeout; the agent should design retry mechanisms with exponential backoff. For example, if ACRCloud doesn't respond within 2s, try once more; if still fails, log the failure and maybe try a different strategy (like switch to global DB if custom DB failed, or vice versa).
- The system should not crash on exceptions – use try/catch around network calls and use fallback defaults when necessary.
- If a module becomes unresponsive (e.g., STT stream stalls), the agent should have watchdog timers to restart that component or reinitialize the connection.
- For the ProPresenter control, as specified, if a command fails, try again up to a couple times (FR-5.5). If the connection is lost, attempt reconnect every few seconds until restored. During that time, perhaps queue up any commands (or alert operator to do manual control).
- **Isolation:** Each module should be as independent as possible (microservice mindset) so that a failure in one doesn't cascade. For instance, if Music ID fails (doesn't get a result), the rest of the system still runs; it just won't auto-display that song, but voice commands can still work.
- Logging of errors is vital for post-mortem. The agent should ensure all errors/exceptions generate log entries with timestamps and context.

- Also consider thread safety and memory: ensure streaming threads don't deadlock. Use non-blocking IO for network calls if possible. This keeps the system responsive even under partial failures.

- **Human Operator Handoff Protocols:** As an agentic system in a critical live setting, the AI design must always account for a **human-in-the-loop fallback**. Concretely:

- When confidence in an interpretation or action falls below a threshold, the system defers to human: e.g., flagging for confirmation on the HITL UI (FR-7.3). The AI should define those thresholds (perhaps initial threshold 0.8 confidence; can be tuned).
- The AI should make it easy for the human to take over complete control if needed. For example, if the operator hits a "Pause AI" button, Kairos should stop issuing automated commands immediately. This requires the agent to implement a global state or kill-switch that the UI can toggle.
- The design should emphasize **graceful degradation**: if one of the AI subsystems fails or is turned off, the human can run things manually. Kairos should not lock out manual control. (In practice, if AI stops sending commands, the operator can use ProPresenter normally – so the AI should not monopolize resources.)
- The system also needs to clearly communicate when it needs input. For instance, a flashing highlight on the UI when awaiting confirmation so the operator notices.
- The protocols should also include escalation: e.g., if the AI repeatedly fails to understand over, say, 3 attempts in a row, it might automatically yield control (maybe suggest the operator to take over until it stabilizes). Or if something out-of-ordinary is detected (like perhaps someone giving a complex command outside its scope), it doesn't try something random – it asks for help.

- Essentially, **human override is always respected**: any time the operator provides an input (confirmation, correction, or manual trigger), the AI should immediately adapt to that and not fight it (e.g., if operator manually changes a slide, AI should know current slide changed and not immediately try to "correct" it back).

- **Continuous Learning via Feedback Loops:** The AI agent's design should incorporate mechanisms for learning from both real-time feedback and offline data:

- As described, the HITL UI will gather examples of errors (transcription errors, intent errors). The agent should store these examples (with audio and what the correct outcome was) in a database of training data.
- The system can periodically (perhaps offline, not during the live session) retrain or fine-tune models using this data. For example, fine-tune an STT custom model or update the speech adaptation phrase hints if it consistently mishears a certain word (e.g., if "Corinthians" is always wrong, add it to hints).
- Dialogflow CX can be improved by adding new training phrases for intents when the AI missed them, or adjusting the entity lists. The agent might automate some of this: e.g., after an event, collate all "unrecognized commands" and present them to a developer or directly add them to an "improve" queue.
- **User feedback integration:** If operators have a way to provide explicit feedback (like pressing a "That was wrong" button on the UI for something that wasn't caught), the system should log that too.
- Potentially, the agent could also learn from patterns in usage: e.g., if in practice, certain intents are rarely used or certain phrasing is common, the agent can optimize or refocus resources accordingly.

- Over time, continuous learning should reduce the reliance on HITL. A governance note: any model updates should be vetted (perhaps tested in a sandbox) before going live, to avoid regressions – the AI agent should follow MLOps best practices if it autonomously retrains (likely, in initial phases, a human data scientist will oversee updates; full autonomous model retraining would need careful governance approval).

- **Transparency and Explainability:** (Governance principle) The AI should be designed to be as transparent as possible in its decision-making. This is facilitated by the HITL UI showing what's happening, but also internally:

- The agent should produce logs or even a trace of reasoning for complex tasks (like if it had to choose between two intents, it could log the scores).
- If possible, incorporate explainability: e.g., Dialogflow doesn't exactly explain why it chose an intent, but the agent can log the confidence. For music ID, log the matching song and score.

- This principle ensures that if something goes wrong, it's not a black box – developers or operators can diagnose it. It's more of a methodology: the agent should not sacrifice clarity for slight performance gains in a safety-critical environment.

- **Scalability and Modularity:** The design should be modular (as we've done by splitting into services) to allow scaling and future expansion:

- If, for example, a future requirement is to handle multiple independent sessions (like two venues simultaneously), the architecture can scale horizontally by deploying another instance of the pipeline.
- If the audio input grows to multiple channels, the agent could replicate the triage->STT->NLU pipelines per channel.
- The cloud components (Google STT, Dialogflow, ACRCloud) are all scalable by nature (they handle concurrency on their side). The AI should be mindful of rate limits though and possibly throttle if needed (e.g., don't open too many STT streams at once beyond account quota).
- Modular design also allows swapping components (if one day they choose a different STT provider or a local ASR model, the agent design should allow that by abstracting the interface).

- The AI should document interfaces between modules clearly (like what data format passes between), to ease maintenance or replacement.

- **Security and Privacy:** (Though not explicitly requested, as a principle for governance) The agent should ensure sensitive data (like audio of people speaking, which could be considered personal data) is handled with care:

- Maybe implement an option to not store raw audio unless needed for training (and if stored, secure it).
- API keys for Google and ACRCloud must be stored securely (not exposed in client-side code or logs).
- The system should be designed to run on a secure network (which likely it is, within a local church LAN for ProPresenter; still, encryption of comms where possible is good – Google APIs are HTTPS, the WebSocket to ProPresenter is presumably within LAN, maybe not encrypted but can be since it's local).
- Ensure the web portal has authentication and uses HTTPS if accessed over a network.
- The AI agent should incorporate these security steps by default to satisfy governance review.

External Interface Definitions

Kairos interacts with several external systems and services. The following table summarizes each external interface, including its purpose and how the AI system uses it:

Table: External Interfaces and Integration Details

External Interface	Purpose / Function in Kairos	Integration Method & Protocol	Notes / Dependencies
Google Cloud Speech-to-Text v2	Speech recognition – converts live audio to text.	**gRPC Streaming API** (preferred) or REST. The system opens a streaming recognition session and sends audio chunks, receiving interim and final transcripts.	Requires Google Cloud credentials. Using `long` model for continuous streaming (no need to reopen per utterance). Latency depends on internet (usually low). Pricing per 15s of audio applies.
Google Dialogflow CX	Natural Language Understanding – maps text to intents.	**REST API** (`detectIntent` endpoint) or client SDK over HTTPS. Each transcript (or significant partial) is sent with a session ID, returns intent JSON.	Dialogflow agent pre-configured with Kairos intents/entities. Requires OAuth credentials. The system may also use Dialogflow Management API to update entities (song list) when content changes.
ACRCloud Music Recognition Service	Audio fingerprinting for music identification.	**HTTP REST API** for Identify. Kairos sends audio (PCM or fingerprint) via POST, receives JSON result with song metadata. Alternatively, use ACRCloud SDK (which internally calls the API).	API access key, secret, and host needed. The project is set up with ACRCloud's music DB and possibly custom content bucket for non-standard songs. Typical usage: 10s audio per ID request.
ProPresenter 7 (Remote Control API)	Control of presentation slides and media.	**WebSocket (TCP)** connection to ProPresenter's remote control port [1]. After connecting (with proper headers and auth token if set), send control commands as text frames (JSON commands as per ProPresenter's protocol).	ProPresenter must have Network API enabled (with a set port and optional password). The AI sends commands like triggering specific slide indices, clearing media, etc. Connection is on local network (e.g., ws://192.168.1.10:port/remote). The protocol is documented but proprietary [2]. Low-level TCP reliability considerations apply.

External Interface	Purpose / Function in Kairos	Integration Method & Protocol	Notes / Dependencies
ProPresenter Stage Display (if used)	(Optional) Reading current slide info or timers.	**WebSocket** on the "stagedisplay" channel [5]. The AI could subscribe to get current slide text or stage messages (not essential, but could be used for verification or context, e.g., knowing what song is currently live).	Same host/port as remote, but different channel ("stagedisplay"). Needs separate auth message. Not required for minimal operation, but useful for context (e.g., knowing current song to interpret "next verse").
Kairos Data Ingestion Portal (Web UI)	Human interface for content management (internal to system).	**HTTP/HTTPS (REST)** – accessed via web browser. Operators log in and use web forms. The portal's backend (part of Kairos) communicates with DB and external APIs (ACRCloud, Dialogflow) as needed.	Runs on local server. Not exposed to public internet. The portal itself doesn't interface with external systems except through backend logic described above.
Kairos HITL Verification UI	Human interface for live monitoring/ control (internal).	**WebSocket or SSE** from Kairos backend to browser for real-time updates; **HTTPS** for sending operator inputs back. Essentially a web app served by Kairos.	Not external in the sense of third-party, but an interface for users. Should be optimized for low latency (websocket updates). Protected via internal network access.
ACRCloud Console API (optional)	Management of ACRCloud project (adding custom fingerprints).	**REST API** calls for uploading custom content or managing buckets.	Used by Data Portal when adding new audio. Requires separate credentials (same key/ secret). Usually done offline (not time-critical).
Google Cloud (other) – e.g., Google Storage or Secret Manager	Storing recordings or secrets if needed.	*If used*, via respective APIs (HTTPS).	For example, if storing audio feedback data for training, could use cloud storage. Secrets for API keys could be stored in Secret Manager. These are auxiliary and not part of main flow.

Each interface integration will be thoroughly tested. All cloud APIs (Google STT, Dialogflow, ACRCloud) have usage quotas; the AI will include monitoring of API usage and fallbacks if limits are hit (e.g., default to

manual control or offline mode if cloud STT is not available). The ProPresenter API integration is critical – it will be tested on the exact version in use and any version updates of ProPresenter must be evaluated for compatibility (e.g., changes in protocol as noted in their docs).

Non-Functional Requirements (NFRs)

Beyond the functional behaviors, Kairos must meet several non-functional criteria to be considered successful. These cover performance, reliability, scalability, and overall autonomy/safety. Each NFR is labeled for reference:

- **NFR-1: Low Latency Interaction** – The system shall operate in real-time, with minimal delay from user input to system action. Specifically, **voice command latency** (from end of speech to action executed) should be *under 1.5 seconds*, with a target of ~1.0 second or less for common commands. In many cases (simple commands), latency could be as low as 500ms. For **music recognition trigger**, the time from song start to lyrics on screen should ideally be *under 5 seconds*, maximum 10 seconds. These latency targets ensure the AI feels responsive and doesn't lag the live event. Measurements will be done during testing (e.g., using timestamps in logs to compute pipeline delays). If any part exceeds its budget (say STT taking too long), optimizations or alternate approaches (like using a faster model or partial results) must be applied.

- **NFR-2: Accuracy and Precision** – The system must be highly accurate in both recognizing inputs and executing the correct outputs:

- **Speech Recognition Accuracy:** Target a Word Error Rate (WER) of < 10% in the acoustic environment it's deployed (taking into account accent of speakers and moderate background noise). Critical command words should be nearly 100% correct (with adaptation helping achieve this).
- **Intent Recognition Accuracy:** ≥ 95% of voice commands should result in the correct intent and parameters identified (when spoken as trained).
- **Music Identification Accuracy:** ≥ 90% success in identifying songs that are in the reference database within the first 10 seconds. False identification should be < 5% (prefer non-identification over incorrect identification).
- **Action Execution Accuracy:** The correct media action should occur *exactly as intended* for every confirmed command. Essentially zero tolerance for executing a wrong action (hence the HITL safeguard for low confidence). This ties to system accuracy in a holistic way – e.g., it's acceptable if STT misheard something as long as it doesn't result in a wrong slide shown (because either NLU fails to match or asks for confirmation).

- **Metrics & Testing:** We will test with a set of sample voice commands and songs. Success criteria might be: in a test of 50 voice commands, at least 47 are handled correctly without human help; in 20 song play tests, at least 18 are auto-recognized correctly, etc. Continuous monitoring of accuracy in production will be done through logs (comparing how often operator had to correct something).

- **NFR-3: Reliability and Uptime** – Kairos should be reliable throughout an event:

- **System Uptime:** The system should run stably for at least 4 hours continuously (typical length of event with rehearsal), ideally 24+ hours, without needing a restart. Uptime requirement: >= 99%

during intended operation times. (This implies no crashes; memory leaks or unhandled exceptions must be avoided).
- **Failover Mechanism:** In case a component fails, the system should degrade gracefully rather than totally fail. For instance, if STT service is down, perhaps the system could still identify songs via audio (or vice versa), and alert that voice commands won't work. Or the operator can use manual control – the system should allow that switch with minimal fuss.
- **Data Persistence:** The content DB (songs, etc.) and any important configs should be persisted so a restart doesn't lose data.
- **Transactions:** Actions like updating content should be transactional (don't corrupt data if something fails mid-way).
- The AI agent should incorporate testing like simulating network outages, high latency, etc., to ensure reliability under less-than-ideal conditions.

- **Maintenance:** If an update is needed (e.g., new model or content), it should be possible to do that without significant downtime (maybe a quick restart or a hot-swap of config). But presumably, maintenance happens outside live usage.

- **NFR-4: Scalability** – While the initial deployment might be for one venue and one audio stream, the architecture should support scaling:

- The system could potentially handle **multiple audio input streams** or multiple instances (for different rooms). The AI's modular design (with distinct pipelines for each source) should allow instantiating another pipeline easily. Cloud services like STT and Dialogflow can scale with multiple sessions, but we must ensure the code can handle concurrency (thread-safe, non-blocking design).
- If the content library grows large (say hundreds of songs), the NLU and search should still perform well. Dialogflow CX can handle many entity entries, but we should ensure lookup times remain low.
- The system should be able to handle a higher rate of commands if needed – e.g., if in a busy scenario commands come in rapid succession, the pipeline should queue/process them without dropping. A rough expectation: maybe up to **10 voice commands per minute** sustained, and the system should keep up.
- **Resource usage:** The AI should optimize CPU/GPU usage to be able to run on the available hardware (perhaps just a standard PC or server at the venue). We assume the host machine is the same running ProPresenter (which typically is a well-spec'd PC). The AI's background processing (like VAD, WebSockets, etc.) should not interfere with ProPresenter's performance. Memory footprint should be reasonable (e.g., under a few hundred MB).

- If needed, components like STT can be offloaded to separate machines or cloud (which they are in cloud). So horizontal scalability is possible by distributing modules (for example, run the audio processing on an edge device, and cloud for recognition).

- **NFR-5: Autonomy and User Dependence** – A key goal is to maximize the autonomy of the AI while maintaining safety:

- The system should be able to handle the **vast majority of routine operations without human intervention**. We target that after initial roll-out and tuning, >90% of voice commands and song cues are handled autonomously (i.e., the AI doesn't ask the operator for help). Initially, during pilot, this might be lower, but it should improve with learning.

- The AI should never require more effort from the operator than doing the task manually would. If the HITL UI confirmation becomes too frequent, that's a signal to improve the AI.
- **Autonomy boundaries:** The AI will not attempt tasks outside its defined scope. For example, it will not try to modify content layouts or control devices it's not supposed to. This is important for governance – it sticks to its functional requirements and doesn't get creative in unintended ways.
- The agent will abide by any **policy constraints** set by governance. For instance, if certain media should not be shown unless expressly approved, the AI will not override that. (E.g., maybe it should never display copyrighted lyrics from a song that's not pre-approved – the AI should then refuse and ask human).

- Autonomy also implies the AI can recover from minor issues by itself (like reconnecting to APIs) rather than always asking a human to fix it.

- **NFR-6: Auditability and Logging** – The system must maintain logs for all actions and decisions, to satisfy governance review and troubleshooting:

- Every voice command and the resulting intent/action should be logged (with timestamp, transcription, confidence, intent, action taken, whether auto or confirmed by human).
- Similarly, music detections and content loads should be logged.
- Any errors or exceptions should be clearly logged with stack traces (for developers) and user-friendly messages (for operators).
- Logs should be timestamped and preferably in UTC (or local with TZ). They should be stored at least for the duration of the event, and archived if needed for later analysis.

- These logs allow governance to audit if the AI did something unexpected and why. They also feed the improvement process.

- **NFR-7: Security and Privacy** – (Adding as it's often a concern) The system will handle possibly sensitive audio (like anything said on stage) and control a critical system (presentation). It must be secure:

- Only authorized individuals can access the Data Portal and HITL UI. Implement authentication and perhaps network-level security (e.g., only accessible within certain VLAN).
- API keys for cloud services are kept secret. The agent should use secure storage (not plaintext in code). If using environment variables or config files, those are protected.
- The communication with cloud APIs is encrypted (HTTPS). The WebSocket to ProPresenter might not be encrypted, but since it's local, that's acceptable; if used over a wider network, a VPN or secure tunnel might be considered.
- Ensure that if the system records audio for analysis, this data is protected and managed per privacy guidelines (e.g., maybe don't record whole sermons, only short snippets for debugging, and purge them after use).
- The AI itself should be protected from external tampering – since it's an autonomous agent with control, one must ensure only the intended user inputs (voice or operator UI) influence it. This means securing the microphone input (someone from outside shouldn't hijack it with commands unless they have the mic) – this is more physical security/policy.

Below is a summary table highlighting some of the key performance and success criteria (combining several NFR targets for clarity):

Table: Key Success Criteria and Performance Targets

Aspect	Metric	Target Value
Voice Command Latency	Time from end of spoken command to action execution (slide change)	$\leq$ *1.5 s* (typical $\leq$1.0 s)
Music Recognition Latency	Time from music start to lyrics displayed	$\leq$ *5 s* (max 10 s)
STT Accuracy	Word Error Rate (WER) on commands	< *10%* WER ($\approx$90%+ accuracy)
NLU Intent Accuracy	Correct intent detection rate	$\geq$ *95%* of commands correctly understood
Song ID Accuracy	Song correctly identified (if in library)	$\geq$ *90%* within first 10s
False Trigger Rate	Unintended actions due to AI error	$\approx$ *0%* (no action without proper intent; use HITL to catch uncertainties)
Uptime Reliability	System uptime during event	$\geq$ *99%* (no unplanned downtime)
Autonomy Rate	Commands handled without human intervention	$\geq$ *90%* (after training phase)
Human Confirmation Frequency	Operator interventions per service	*"As few as possible"* – aim for <1 per 10 commands (improve over time)
Error Recovery	Time to recover from a fault (e.g., reconnect STT)	< *5 s* to auto-recover or failover
Scalability	Concurrent sessions or channels	Design for modular scaling (e.g., handle 2+ mics if needed) without performance loss
Security	Unauthorized access or data leak incidents	*0* incidents – follow best practices (authentication, encryption)

These NFRs collectively ensure that Kairos will perform effectively in a live environment and maintain the trust of its human operators and stakeholders.

Agentic Reasoning and Sequencing Recommendations

Given that an autonomous AI agent will be orchestrating the development and possibly the operation of this system, it's important to outline how the agent should reason through certain scenarios and decisions.

The agent should follow these guidelines on *when to fine-tune models, when to consult humans, and when to retry or adjust its approach*:

- **When to Fine-Tune or Adapt Models:** The AI agent should consider fine-tuning or adjusting a model when it observes a **systematic pattern of errors** that could be alleviated by learning. For example:
- If the STT frequently misrecognizes specific words or names (perhaps unique to the organization, like uncommon names or jargon), the agent should update the STT's **adaptation phrases** with those words. This can be done dynamically by maintaining a list of important terms (e.g., song titles, speaker names) – updating this list does not require model retraining but is a form of on-the-fly customization.
- If after an event, analysis of logs shows certain phrases or accents caused mis-transcriptions, the agent might recommend or initiate a process to fine-tune an ASR model (if using a custom trainable model or switching to a more specialized model). Since Google STT is cloud and not directly fine-tunable by us, adaptation is our main tool; however, the agent could also explore alternative STT providers (like if English accent was an issue, maybe try a model specifically for that accent).
- For NLU (Dialogflow CX), fine-tuning means updating the agent's training. The agent should add new training phrases for intents when novel phrasing was encountered, and adjust entity lists. The agent can automate this by collecting confirmed corrections from the HITL loop. For instance, if users often say "lyrics for [song]" and we hadn't included "lyrics for" in training, the agent adds it. This can be done via Dialogflow's API or flagged for a developer, depending on autonomy allowed.
- **Schedule of tuning:** The agent shouldn't tweak models during a live session (to avoid unpredictable changes). Instead, it can schedule these updates in a **maintenance window** (e.g., between events). Possibly at the end of each event, it writes out a "learning report" with suggestions or automatically calls APIs to update. These changes then take effect in the next run (after verifying nothing broken).
- The agent should also evaluate **when not to fine-tune**: if an error is one-off or due to a bizarre edge case, it might not be worth retraining. Focus on recurring issues.
- If using any ML model locally (not currently, but suppose a VAD or classifier model), the agent could retrain that with new data if performance was lacking (though VAD is probably fine out-of-box).

- In summary, fine-tune when there's a clear benefit and enough data to support the adjustment, and do so in a controlled, testable manner (prefer incremental changes and test on past data to ensure improvement).

- **When to Consult the Human Operator:** The AI's default mode is autonomous, but it should *automatically consult the human* in scenarios including:

- **Low Confidence Intents:** If Dialogflow returns an intent with low confidence, or multiple intents are close in score (like the system isn't sure), immediately flag for human confirmation (FR-7.3). For example, if confidence < 0.7 or if STT had to fallback on alternatives. The agent should design a threshold logic and possibly adapt it as it learns false positives/negatives. The human will then confirm or correct, as per HITL UI design.
- **Ambiguous Entity Resolution:** E.g., if user says "Show Grace", and there are two songs "Grace" and "Graceful" in the library, the AI might not be certain which one. It should ask the operator rather than guess. Alternatively, it might try a clarification dialogue with the user, but in live use that's not ideal, so operator is better.

- **Potentially Destructive Actions:** If an intent could disrupt the program feed significantly (like "clear all screens" or "stop streaming"), the AI might always ask confirmation unless it's absolutely sure the command was given intentionally. This is a safety check.
- **Repeated Misunderstandings:** If the AI fails to understand a couple of attempts from the user (like the user rephrases because AI didn't act), the agent should involve the human. For instance, after two "fallback" intents in a row, flash a help request to operator ("User might be asking for something not understood, please assist").
- **System Uncertainty or Novel Situations:** If a user says something completely out of scope (not in any intent), the AI shouldn't just ignore silently; it should at least log it. Possibly the operator can intervene if needed ("He's asking for lights off – not my domain"). But mainly, consult human when not confident about what to do.
- The agent's design should ensure that this handoff is smooth and quick. The UI should clearly show "Needs your input" so the operator can act within seconds. If the operator doesn't respond in, say, 3-5 seconds (maybe they are busy), the agent needs a policy – likely do nothing rather than risk it. Or if it's a simple thing like NextSlide and it was low confidence but likely, maybe it can do it anyway after a timeout – but defaulting to caution is better for now (we can state: if no response and the risk of action is low, maybe proceed; e.g., showing wrong next slide is low risk because operator can always go back, but showing entirely wrong song lyrics could be more noticeable).

- **Human override commands:** If the operator explicitly presses a manual control (like hits a next slide on ProPresenter or a control in UI), the AI should treat that as gospel and perhaps temporarily suspend its own actions if there's any conflict. For example, if operator manually jumps to a song, the AI should update its context to that song and not fight it.

- **When to Trigger Retries and Alternative Strategies:** The AI agent should implement retry logic (as discussed in fault tolerance) and know when to switch tactics:

- **Transient Failures (Network/API errors):** On a failure to reach an API (timeout, HTTP 500, etc.), the agent should retry after a brief delay. Typically:
    - 1st failure: wait 1 second, try again.
    - 2nd failure: wait 2 seconds, try again.
    - If still failing, assume the service is down. At that point, the agent should either failover or alert. For STT: if down, maybe it can try a backup STT service (if pre-integrated) or as last resort, tell operator "Voice recognition unavailable". For ACRCloud: maybe retry a third time or switch to an alternate service if any (maybe none, so just give up and log).
- **Recognition Uncertainty:** If STT returns a very low confidence transcript or blanks, it might retry processing that audio with a different approach (e.g., if using Google with auto model, maybe try a specific model or re-send with different parameters or smaller segments). Or if time allows, re-prompt user (though in a live scenario you can't easily ask the speaker to repeat unless it's obvious to them nothing happened).
- **Music ID fallback:** If a first attempt with 5s audio returns nothing, agent should automatically send a longer sample (maybe 10s or 15s) for a second try. If still nothing, fallback to not found.
- **ProPresenter command retry:** As specified, if no acknowledgement or effect from a command in a short time, resend (perhaps the agent can listen to stage display data to see if slide index changed – if not changed in 0.5s, try again).
- **Alternate pathways:** The agent can maintain an arsenal of methods. For instance, if internet is lost entirely, the agent might not have cloud STT. Perhaps as a backup, the system could switch to an offline model (if one was embedded). This isn't specified originally, but agentic design might

consider it. For now, likely not implemented, but in reasoning, the agent would at least alert and suggest a manual mode in such a case.

- **Continuous Monitoring:** The agent should also periodically verify that key connections are alive (ping Dialogflow maybe by a dummy intent periodically, or check WebSocket to ProPresenter hasn't dropped). If something is off, a preemptive retry or reconnect can happen before a command is actually missed.

- **Limit retries:** The agent should avoid infinite loops. E.g., if a network is down, after a few tries it should stop spamming and just report error. This is to avoid cascading issues or flooding logs.

- **Sequence of Decision Making:** In real-time operation for each input, the agent will follow a sequence something like:

- **Audio Triage**: Decide route (speech vs music).
- **If speech**: Stream to STT, get text.
    - As text is coming, maybe form an early hypothesis.
- **NLU**: Send to Dialogflow, get intent.
    - If confidence high and intent clearly mapped: proceed.
    - If confidence low/ambiguous: go to HITL wait state.
- **If in HITL wait**: Display info, wait up to X sec for operator.
    - If operator confirms/corrects: use that.
    - If operator does nothing within time:
        - Option 1: drop the action (preferable for safety), possibly log "timed out".
        - Option 2: if it's something harmless like NextSlide and the guess was likely right, maybe proceed. (This could be configurable.)
- **If music**: Record sample, call ACRCloud.
    - If identified confidently: treat as an implicit GoToSong intent with known song.
    - If low confidence: either ask operator "Music detected, unsure of song" or just do nothing (probably do nothing or just flash "unknown song").
    - If not identified: do nothing (operator will manually handle in that case usually).
- **Execute orchestration**: Send commands to ProPresenter.
    - If fails, retry as per logic.
    - Confirm action done (if possible).
- **Log outcome**: to history and potentially for learning.

The agent's design should ensure these steps are synchronized properly – e.g., no overlap that causes confusion (if two commands come in at once, queue them in order; maybe handle one at a time unless we truly expect overlapping voice commands which is rare).

- **Proactive Sequencing:** The agent can also sequence proactive actions:
- For example, if a song is identified via music, it might be proactive. Or if the setlist of songs is known (maybe via an input to Data Portal for the day's plan), the AI could pre-load those presentations in a cache or even pre-open them in ProPresenter (if that speeds up display) and pre-fetch any needed resources. It wasn't explicitly described, but an agent might reason to do that to optimize performance.

- Similarly, if after one song finishes (maybe it detects silence after music), the agent might anticipate that next song might start or that the pastor might speak – it could adjust by focusing STT vs ACR

priority accordingly. This is more speculative, but an intelligent agent can incorporate simple context rules (like after music stops, listen for speech commands like “next song” etc.).

- **Governance and Safety Checks:** The agent should always sequence tasks with a **safety-first approach**:

- If ever in doubt, prefer asking a human or doing nothing over taking a risky action.
- The agent should not fine-tune or change configurations in the middle of a live session in a way that could degrade performance without thorough testing (likely it won’t, as we said, schedule improvements outside live).
- Keep the human informed of what’s going on (transparency principle).
- Provide fail-safes (like the big red button metaphor – operator can kill the AI output instantly if needed, and agent should incorporate that possibility into its state machine).

Following these reasoning guidelines will help the agent not only build the system effectively but also manage it in real-time in a way that is aligned with human operators and governance policies. The end result should be an AI that behaves as a reliable assistant: fast and autonomous when it’s sure, and quick to get human guidance when it’s not – all while continuously getting better through learning.

---

1 2 3 4 5 ProPresenter API 7.6 documentation
https://jeffmikels.github.io/ProPresenter-API/Pro7/