# Building a Human-in-the-Loop AI Supervisor System

## Introduction and Project Overview

The objective of this project is to develop a human-in-the-loop AI supervisor system designed to enhance the capabilities of Frontdesk’s AI receptionist. While AI receptionists provide efficient and automated customer interactions, they inevitably face challenges such as hallucinations, incomplete answers, or failure to understand complex queries. These limitations impact customer experience and risk miscommunication.

To address these issues, the system will introduce a seamless escalation workflow where unresolved or uncertain queries are promptly handed over to a human supervisor. The supervisor reviews and responds to pending help requests, enabling real-time intervention and maintaining trust. Importantly, the system incorporates learning mechanisms, allowing the AI agent to update its knowledge base from supervisor interactions, thereby improving future autonomous handling of similar queries.

Key features include:

* A simulated AI agent powered by LiveKit, embedded with fundamental business knowledge and call handling logic.
* Automated detection of unanswered queries, with notifications to the caller and logging of help requests in a lightweight database.
* Alerting mechanisms for supervisors through console interfaces or webhook notifications.
* A minimalistic supervisor UI to view, respond to, and track help request lifecycle states like Pending and Resolved.
* Automated AI follow-up messages based on supervisor inputs to close the communication loop.

Deliverables for this project include a fully integrated codebase hosted on GitHub, an accompanying readme detailing setup and design decisions, and a demo video illustrating system workflow. The timeline for completion is one week, emphasizing modular design, quality code, and maintainability. This document will guide the implementation, ensuring clarity and consistency throughout the project.

## AI Agent Setup with LiveKit

Setting up the AI agent with LiveKit marks a foundational step in building the AI receptionist system. Although LiveKit is primarily known as an open-source platform for real-time audio and video communication, it can be repurposed in this context to simulate phone call reception and basic interaction flow. This section outlines the approach for integrating and configuring LiveKit’s SDK to establish the AI agent’s call handling capabilities along with initial prompt design tailored to the salon business scenario.

### Leveraging LiveKit SDK for Simulated Calls

LiveKit provides robust APIs and SDKs for managing WebRTC-based calls. To simulate incoming calls, the system will utilize LiveKit’s server-side and client SDKs (available in JavaScript/TypeScript) to create virtual call sessions. When an incoming call event is triggered in the simulation, the AI agent joins the LiveKit room as a participant, enabling it to "listen" and "respond" via programmatic audio or text interfaces.

While LiveKit manages session lifecycle, signaling, and media streams, this project simplifies actual voice processing by replacing it with text-based simulated interactions logged to the console or sent via webhooks. This allows developers to focus on AI prompt design and escalation logic without implementing real-time speech recognition or synthesis.

### Prompt Engineering with Embedded Business Knowledge

Critical to the AI agent's initial effectiveness is the prompt engineering that seeds it with relevant business information. The prompt structure includes:

* Basic salon details (services, hours, location)
* Example FAQs (pricing, appointment booking policies)
* Instructions on handling common greeting and closing statements
* A directive to flag any queries outside its programmed scope for supervisor escalation

An example prompt snippet to the AI model might be:

"You are the virtual receptionist for ‘Gloss & Glow Salon’. You know our services include haircuts, coloring, and styling. Hours are 9am to 6pm, Monday through Saturday. If a caller asks about appointments, prices, or hours, respond clearly. If unsure about a question or it’s outside this domain, notify the caller and escalate the request to a human supervisor."

This structured prompt guides the AI model toward accurate, business-specific responses while embedding early detection of unanswerable queries.

### Basic AI Response Capabilities and Escalation Criteria

The AI agent is programmed to parse incoming simulated calls and generate appropriate responses based on the prompt context. It uses keyword matching and intent recognition (either via simple rules or a lightweight AI model) to classify queries it can answer versus those it cannot confidently handle.

* **Answerable queries:** Directly respond with predefined or dynamically generated answers.
* **Unanswerable or ambiguous queries:** Inform the caller that a supervisor will assist shortly and automatically create a help request entry.

The escalation logic is essential to ensure the caller is never left without acknowledgment. By combining LiveKit’s event hooks (e.g., on message received) with prompt-engineered AI logic, the system smoothly transitions from AI-only handling to human-in-the-loop intervention when necessary.

### Role of LiveKit and Focus on Prompt Design

LiveKit serves as the communication backbone, orchestrating the simulated call environment and enabling event-driven workflows. However, since LiveKit is not inherently designed for AI conversational systems, significant emphasis is placed on prompt design to direct AI behavior explicitly. This means the accuracy and reliability of the AI agent largely depend on carefully crafted prompts and classification logic rather than complex conversational AI models.

### Scope and Limitations

It is important to note that full conversational handling with natural language dialog management is beyond this project scope. The AI agent's role centers on:

* Receiving simulated calls through LiveKit sessions
* Responding with straightforward, scripted answers
* Detecting unknown queries and triggering help requests for supervisor follow-up

This minimal scope focuses efforts on proving the human-in-the-loop escalation architecture, ensuring modularity and ease of integration rather than full AI conversational sophistication.

## Design and Implementation of Human Request Handling

The core functionality of the human-in-the-loop system revolves around efficiently escalating unresolved or uncertain queries from the AI receptionist to a human supervisor. This section details the designed lifecycle of help requests, the supporting data structures, notification mechanisms, and architectural considerations for managing this escalation workflow.

### Escalation Workflow and Request Lifecycle

When the AI agent encounters a question it cannot confidently answer, it initiates an escalation protocol that preserves customer experience and enables effective supervisor intervention. The escalation process follows these steps:

1. **Notify the Caller:** The AI immediately informs the caller that their question requires supervisor attention and that they will follow up shortly. For example, the AI might respond, *“I’m going to check with my supervisor to help answer your question. Please hold on.”*
2. **Create a Help Request:** The system logs a new help request entry in the database with all pertinent details.
3. **Alert the Supervisor:** A notification simulates alerting the human supervisor—for instance, a console log message or a webhook POST to an endpoint representing the supervisor’s UI—containing the request context.
4. **Supervisor Intervention:** The supervisor reviews the help request, either resolving it by providing an answer or marking it unresolved if no suitable resolution is found.
5. **AI Follow-up:** Upon resolution, the AI sends a follow-up message to the caller to close the interaction loop. If unresolved, the AI may inform the caller accordingly or escalate further.

The help request lifecycle is formally modeled with three core statuses:

| Status | Description |
| --- | --- |
| Pending | Request logged, awaiting supervisor review |
| Resolved | Supervisor provided a satisfactory answer |
| Unresolved | Supervisor could not resolve the request |

Each request transitions sequentially, ensuring clear tracking and reporting.

### Database Schema and Storage Design

To store and manage help requests elegantly and efficiently, the system utilizes a lightweight, NoSQL database such as **DynamoDB** or **Firebase Realtime Database**. These choices enable low-latency reads/writes, built-in scalability, and straightforward integration with serverless or backend APIs.

A proposed schema for help requests includes the following fields:

| Field | Type | Description |
| --- | --- | --- |
| requestId | string (PK) | Unique identifier for each help request |
| customerId | string | Reference to the caller or customer identifier |
| question | string | The original, unresolved question text |
| status | string | Current request state: Pending / Resolved / Unresolved |
| createdAt | timestamp | When the request was created |
| updatedAt | timestamp | Timestamp of last status update or supervisor action |
| supervisorId | string | ID of the supervisor who handled the request (nullable) |
| resolution | string | Supervisor’s response or notes regarding resolution (nullable) |

This schema supports indexing or querying by status and customerId, enabling quick lookup for UI display and follow-up logic.

### Notification and Supervisor Alert Simulation

Upon request creation, the system simulates supervisor notification through:

* **Console Logs:** Simple output displaying new requests and details for immediate development visibility.
* **Webhook Callbacks:** An HTTP POST request sent to a configured endpoint representing the supervisor UI backend, including structured JSON payload with the request context.

Example console log for a new help request:

[Supervisor Alert] New Help Request Created:  
Request ID: req\_1234  
Customer ID: cust\_5678  
Question: "Do you offer vegan hair products?"  
Status: Pending

Webhook payload sample:

{  
 "requestId": "req\_1234",  
 "customerId": "cust\_5678",  
 "question": "Do you offer vegan hair products?",  
 "status": "Pending",  
 "createdAt": "2024-06-01T12:00:00Z"  
}

These notifications ensure immediate supervisor awareness while decoupling the AI request handling logic from UI concerns.

### Code Modularity and Flow Management

To maintain clarity and facilitate scalable development, the human request handling is architected as a modular component with well-defined responsibilities:

* **Request Manager Module:** Handles creation, updating, and querying of help request entries in the database.
* **Notification Module:** Abstracts console and webhook-based supervisor alerts, enabling easy extension to real notification channels.
* **AI Escalation Controller:** Coordinates AI messaging to callers, invokes the request manager for logging, and triggers notifications.

Each module exposes clear interfaces and performs graceful error handling. For example, if database write fails, the system logs errors and informs the AI to apologize for the inconvenience without crashing the call session.

This modular design supports straightforward unit testing, future integration of authentication for supervisors, and potential migration to more sophisticated persistence layers or notification services.

By combining a well-defined request lifecycle, elegant data structures, simulated notification mechanisms, and modular code design, the system ensures efficient, transparent escalations that bridge AI limitations with human expertise in a seamless workflow.

## Supervisor Response Handling and UI

The supervisor interface is a key component in the human-in-the-loop AI supervisor system, enabling efficient management of help requests escalated by the AI receptionist. Built as a minimalistic web-based UI, it prioritizes clarity, responsiveness, and smooth workflow integration over cosmetic polish, thereby aligning with project goals of modularity and simplicity.

### User Interface Components Overview

The supervisor UI is composed of three main sections to guide user interactions:

1. **Request List Panel**  
   This panel displays all currently *Pending* help requests in a concise, scrollable list. Each entry shows summarized information including:
   * Request ID
   * Customer ID (or anonymized caller label)
   * Timestamp of request creation
   * Preview snippet of the unanswered question

* The list supports sorting by recency and filtering by status (Pending, Resolved, Unresolved) to give supervisors control over workload management.

1. **Request Detail View**  
   Selecting a request from the list loads its comprehensive details on the right-hand pane, including:
   * Full question text
   * Customer context if available (e.g., previous calls or notes)
   * Current status with timestamps
   * Supervisor response input area (multiline text box for writing an answer or comments)
   * Submit buttons to mark the request as *Resolved* or *Unresolved*

* The UI enforces validation to ensure a response must be supplied before a resolution status can be set.

1. **History and Archive Section**  
   Below or accessible via tabs, this view provides supervisors with the ability to browse previously handled requests. Requests are listed with their resolution status and the supervisor’s responses, supporting accountability and retrospective analysis.

### Supervisor Response Workflow and Data Linkage

When a supervisor submits their resolution through the UI, the following sequence occurs:

* The response, accompanied by the updated status (Resolved or Unresolved), is sent to the backend API which updates the corresponding help request record in the database.
* The system links the supervisor’s user ID to the request entry (supervisorId field) and records the response text and update timestamp.
* Immediate state synchronization updates the UI components to reflect the new status and move the request out of the Pending queue, enhancing UX transparency.

This tight coupling between supervisor actions and request records guarantees data integrity across the system, preventing race conditions or orphaned requests.

### AI Follow-up Triggering and Simulation

Crucially, upon supervisor resolution submission, the system invokes the AI follow-up module to “close the loop” with the original caller. The AI generates a polite, clear update message based on the supervisor’s response. For example:

“Hello, thank you for your patience. Regarding your question about [topic], here is the information: [supervisor’s answer]. If you need further assistance, please let us know.”

Since the project scope excludes real voice calls, this follow-up is simulated by either:

* Logging the AI follow-up text to the console with request and customer identifiers, or
* Sending a webhook POST payload to a specified endpoint representing the call/text delivery service, containing the response details linked to the original customer.

This simulation ensures the supervisor’s contributions visibly impact the customer’s experience without complex telephony integration.

### Emphasis on Simplicity and Clean State Management

The UI’s design intentionally avoids feature bloat, focusing on essentials required for effective supervisor intervention:

* Clear visibility into pending workload
* Straightforward response entry and request status updates
* Easy access to historical cases for context and reference

Additionally, the system architecture ensures that every state transition—such as moving from Pending to Resolved—is cleanly persisted and triggers the appropriate downstream effects, like AI knowledge base updates. This enables continuous learning by the AI receptionist from supervisor feedback, fulfilling the key requirement of the human-in-the-loop paradigm.

Overall, the supervisor UI and response handling deliver a streamlined and reliable interaction point between human expertise and AI automation, bridging gaps in coverage while maintaining clear audit trails and reinforcing data consistency.

## Knowledge Base Updates and Maintenance

A critical aspect of the human-in-the-loop AI supervisor system is the **knowledge base**, which enables the AI receptionist to learn from supervisor interventions and incrementally improve its autonomous responses. This section outlines the design and operational strategy for implementing a persistently stored, dynamically updatable knowledge base.

### Knowledge Base Data Model and Storage

The knowledge base is structured as a collection of *Learned Answers* keyed by normalized question intents or canonical query forms. Each knowledge entry contains:

| Field | Type | Description |
| --- | --- | --- |
| entryId | string (PK) | Unique identifier for each knowledge entry |
| questionKey | string | Normalized or hashed representation of question intent |
| answerText | string | The supervisor-provided answer text |
| createdAt | timestamp | Timestamp when the entry was created |
| updatedAt | timestamp | Timestamp when the entry was last updated |
| sourceReqId | string | Reference to the help request that originated the answer |

For storage, a lightweight NoSQL database (such as DynamoDB or Firebase Firestore) is recommended, matching the scalability and low latency requirements of the help request system. This allows rapid lookups using the questionKey to retrieve answers during AI call handling.

### Update and Augmentation Logic

When a supervisor resolves a help request by providing an answer, the system extracts the normalized question context and performs an **upsert** operation on the knowledge base:

* If an entry for the questionKey does not exist, a new record is created.
* If an entry already exists, the supervisor’s current answer may either overwrite the existing answer or be appended as a separate version depending on update policy. Overwriting is simpler but could risk losing past context; versioning or answer history can be implemented for robustness at the cost of complexity.

The update process includes mandatory data validation checks (e.g., minimum answer length, prohibited content filtering) to ensure quality and consistency of knowledge base entries.

### AI Agent Integration with Knowledge Base

During subsequent simulated calls, before defaulting to preset FAQ responses or escalation, the AI agent queries the knowledge base using the normalized form of the incoming question. If a matching questionKey is found, the corresponding answerText is retrieved and returned to the caller autonomously.

This direct integration closes the learning loop: human supervisor knowledge is codified into the knowledge base, enabling the AI to handle similar queries confidently without repeated escalation.

### Knowledge Base UI: "Learned Answers" View

To maintain transparency and facilitate oversight, a dedicated **Learned Answers** section is incorporated into the supervisor UI. This simple view lists entries from the knowledge base with columns such as:

* Question snippet or normalized key
* Supervisor-provided answer preview
* Date of last update
* Linked help request ID(s) for provenance

Supervisors can search, filter, and optionally flag or edit entries to correct or refine knowledge. This promotes ongoing quality assurance and governance of the AI’s learned knowledge.

### Consistency, Reliability, and Error Handling

Maintaining consistency is vital, particularly to avoid stale or conflicting answers. The system applies optimistic concurrency control (e.g., updating only if timestamps match) or locking mechanisms to prevent race conditions on simultaneous knowledge base edits.

Error handling ensures database write failures or invalid updates log clear diagnostic messages and alert the system without interrupting ongoing call sessions. In such cases, the system may fallback gracefully by escalating the query or reverting to default answers to maintain caller experience.

By systematically capturing, validating, and persisting supervisor responses as reusable knowledge base entries, this architecture enables the AI receptionist to progressively reduce escalations, improve accuracy, and deliver reliable, evolving customer interactions.

## System Design Considerations and Scalability

Designing the human-in-the-loop AI supervisor system requires careful consideration of several architectural and operational aspects to ensure robustness, maintainability, and scalability. This section explores critical design decisions, including request lifecycle management, knowledge base updates, timeout handling, modularization, and trade-offs between rapid development and thorough engineering.

### Lifecycle Management and Request-Supervisor Linkage

A central design choice is the formal modeling of the help request lifecycle, which ensures clear state transitions and accountability. Each help request progresses through *Pending*, *Resolved*, or *Unresolved* states, with timestamped records and references to the supervising agent responsible for resolution. This linkage not only maintains auditability but also facilitates accurate reporting and retrospective analysis.

To support this, the system implements strong consistency guarantees within the database layer, ensuring that updates to request status and linking fields (supervisorId, resolution text) occur atomically. The backend API validates these updates to prevent invalid transitions, such as marking a request resolved without a supervisory response.

### Graceful Handling of Supervisor Response Timeouts

Another essential design aspect is the handling of supervisor response timeouts to maintain operational flow and prevent indefinite pending states. If a request remains unanswered beyond a configurable timeout period (e.g., 24 hours), the system automatically marks it as *Unresolved* and triggers an AI-generated follow-up notifying the customer of the delay.

This timeout logic is realized via scheduled background tasks or event-driven serverless functions that periodically scan for stale pending requests. By automating this process, the system reduces manual oversight requirements and enhances customer experience by providing timely updates—even when supervisors are unavailable.

### Modular Architecture for Scalability

Scalability considerations have strongly influenced the system’s modular architecture, which separates concerns into discrete components:

| Module | Responsibility |
| --- | --- |
| **AI Agent Module** | Manages call simulation, AI prompt handling, and query classification |
| **Help Request Manager** | CRUD operations on help requests, status transitions, timeout enforcement |
| **Notification Module** | Abstracts supervisor alert mechanisms (console, webhook, future push notifications) |
| **Knowledge Base Manager** | Stores and updates learned answers indexed by normalized question keys |
| **Supervisor UI Backend** | Handles supervisor actions, validates input, synchronizes request states |

This clear modularization enables independent scaling of busy subsystems. For instance, as requests increase from a handful per day to thousands, database performance can be optimized separately, notification throughput expanded, and AI agent concurrency increased without cross-module interference.

### Maintainability, Code Quality, and Error Resilience

To ensure the system runs reliably without constant human oversight, maintainability practices are integrated from the start. These include:

* **Strict typing and interface definitions** across modules to minimize integration bugs.
* **Comprehensive error handling and recovery**, such as retry mechanisms on database writes and fallback responses when AI or supervisor components fail.
* **Extensive logging and monitoring hooks** for operational visibility.
* **Unit and integration tests** for critical flows in escalation, resolution, and knowledge base updates.

Additionally, code is structured logically to facilitate future enhancements like multi-supervisor support or enriched notification channels (e.g., SMS, email).

### Trade-offs Between Speed and Thoroughness

Given the one-week delivery timeline, the project balances development speed with architectural soundness. For example, using a NoSQL database with generous consistency models accelerates prototype delivery but may sacrifice some strict transactional guarantees. Similarly, simulated notifications (console/webhook) enable rapid iteration without complex telephony integration.

These trade-offs are consciously documented and designed modularly to allow incremental hardening post-delivery. Such flexibility ensures rapid demonstration of core concepts while preparing the codebase for future production-readiness and scaling.

By anchoring the design around robust lifecycle management, automated timeout handling, modular scalability, and maintainable code practices, the system is well-positioned to evolve gracefully from a small proof-of-concept to a reliable, high-capacity AI receptionist augmentation platform.

## Conclusion and Next Steps

This project successfully implemented a human-in-the-loop AI supervisor system that escalates unresolved queries, manages help request lifecycles, and incorporates supervisor feedback into a dynamic knowledge base. The design emphasizes clean, modular code and thoughtful error handling, enabling clear separation of concerns and maintainability. The system handles ambiguity gracefully by combining AI prompt engineering with structured escalation workflows. Future enhancements could include Phase 2 live call transfer functionality and richer notification channels. Ultimately, this scalable architecture sets a solid foundation for deeper integration into Frontdesk’s AI ecosystem to continuously improve receptionist accuracy and customer satisfaction.