ChatGPT

Plumbing Business AI Voice Assistant – Project Plan

Project Overview and Objectives

The objective is to develop an **AI-powered voice assistant** for a plumbing business that streamlines customer scheduling and business operations. By leveraging voice recognition and natural language understanding, the assistant will handle appointment bookings, provide service information, and update records hands-free. This system aims to **reduce missed calls and streamline appointment booking** for the business [1], acting as a virtual receptionist that operates 24/7. All data and processing will be cloud-based (no local storage), ensuring reliability and accessibility from anywhere.

Key goals include:

- **Voice-Driven Scheduling:** Allow the business owner (and eventually customers) to schedule, modify, or inquire about appointments via natural spoken conversation with the AI.
- **Seamless Calendar Integration:** Automatically create and update events in Google Calendar to keep the schedule accurate [2].
- **Customer Data Management:** Maintain a database of customer contacts and service history, so repeat customers are recognized and their past service info is readily available.
- **Business Dashboard & Analytics:** Provide a secure web dashboard for the business to review AI interaction logs, see upcoming jobs, and gain insights (e.g. jobs per week, revenue, common service locations) from the data.
- **Automated Notifications:** Send out SMS alerts or reminders to both the business and customers for important events (new inquiries, appointment confirmations, etc.), improving communication and reducing no-shows [3].
- **Scalability and Reliability:** Deploy on Google Cloud Platform (GCP) with container orchestration (Kubernetes) to ensure the service is reliable. The system should scale out if multiple calls or chats occur simultaneously, avoiding any busy signals by spinning up new instances on demand [4].

Ultimately, this voice assistant will save time, improve customer response times, and keep the business organized through intelligent automation.

Features and Requirements

1. Natural Voice Interaction: The core feature is a **conversational voice interface**. The user (initially the business owner, possibly expanding to customers via phone or web) can talk to the assistant to perform tasks. The system will use speech-to-text (STT) to understand user queries and text-to-speech (TTS) to respond with a **natural, high-quality voice**. We plan to use an open-source STT model with proven real-time accuracy (e.g. Facebook's Wav2Vec2, which is well-suited for low-latency voice assistant use [5]) and an open-source TTS model for output (e.g. Resemble AI's *Chatterbox*, which provides human-like speech and

sub-200ms response latency [6]). This ensures the assistant's voice is clear and professional, suitable for a service business.

2. Appointment Scheduling via Google Calendar: The assistant will handle booking and managing appointments. It should collect appointment details through conversation – e.g. customer name, address, service needed, preferred date and time – and then check the plumber's Google Calendar for availability. Using the Google Calendar API, the system can determine free time slots and **create new events for booked appointments** [2]. If the requested slot is already taken, the assistant will **suggest alternative times** to the user [7], just as a human scheduler would. The integration with Google Calendar ensures the schedule is always up-to-date and accessible on all devices. (Initially, we'll integrate with Google Calendar as the primary calendar system.)

3. Customer Database & Auto-Population: All customer details and past appointments will be stored in a cloud database (e.g. Firestore or Cloud SQL). When scheduling, if the assistant recognizes an existing customer (by name or phone number), it will automatically pull up their info (contact details, past services) and populate the appointment with those details. If it's a new customer, the assistant will create a new record in the database for future use. This **speeds up the scheduling process** and builds a CRM-like history for the business. Over time, this database will enable the assistant to provide personalized service (for example, "I see we replaced your water heater last year…").

4. Business Dashboard (Web App): We will develop a **business-facing web application** (frontend built with Node.js/JavaScript frameworks) that serves as an AI Dashboard. This secure dashboard lets the plumbing business owner (and authorized staff) review and interact with the system. Key functions of the dashboard:
- **Conversation Logs:** View transcripts of interactions between the voice assistant and users (with time stamps and details). This is useful for monitoring what the AI is telling customers and for training/quality improvement.
- **Calendar View:** Visualize upcoming appointments, possibly by pulling data from Google Calendar and the customer database. The owner can see all scheduled jobs, and possibly manually adjust or add appointments if needed.
- **Data Analytics:** See statistics and insights derived from the stored data – e.g., number of new leads vs. jobs completed in a given period, most common service locations or types, revenue estimates, etc. The dashboard might show charts or tables (e.g., jobs per week, revenue by month, popular services) and allow filtering by date range. These insights help the business identify trends.
- **Administrative Controls:** Manage settings like business hours, services offered, team member schedules, etc., and configure the AI (for example, updating the script for how it greets customers or what options to offer). Some controls could include linking/unlinking the Google Calendar account, updating SMS notification preferences, etc.

The **dashboard will be a Progressive Web App (PWA)** so that it's installable on mobile devices and provides a native-app-like experience. The owner can receive push notifications (for new bookings or messages) and even use the device microphone to talk to the assistant through the dashboard. This means the business owner could, for instance, use their phone to ask the assistant about tomorrow's schedule or to book a new customer while on the go, and the assistant will respond via voice.

5. SMS/Text Notifications: The system will integrate with an SMS service (such as Twilio) to send out text message alerts and confirmations. For example: when a **new estimate request** comes in from a customer

(via the website chat or phone call handled by the AI), the owner gets an instant SMS alert so they don't miss the lead. Similarly, if a **customer accepts a proposal/bid** and wants to schedule service, the system can send a text notification to the owner with the details. We can also have automated texts to customers: e.g., appointment reminders or confirmations ("Your plumbing service is scheduled for tomorrow at 10am, reply YES to confirm"). Twilio's programmable messaging API allows scheduling such notifications and even handling replies to confirm or reschedule appointments [3]. This feature will improve communication efficiency – customers get immediate responses and confirmations, and the business is promptly alerted to any new activity.

6. Website Chatbot Integration (Future): Although the initial frontend is business-facing, a future enhancement will embed the AI assistant on the business's website as a chat widget or voice bot for customers. This would allow website visitors to **chat with the AI assistant** to request an estimate or schedule an appointment. For instance, a customer could type or speak, "I have a leaking faucet, can I get someone to fix it on Friday?" and the assistant (using the same backend logic) would gather information and potentially schedule an appointment or forward the request. If the customer goes through with a booking or asks for a quote, the system would capture that in the database, create calendar events, and trigger the SMS alerts as described. This extension will **capture leads directly from the website** and free up the business owner from having to answer common questions. (This is noted for a later phase, but we keep it in mind during design to ensure the backend can handle multi-channel input.)

7. Cloud-Only Deployment and Scalability: The entire application will run on the cloud (GCP). We will containerize the backend (Python) and frontend (Node.js) and deploy them on **Google Kubernetes Engine (GKE)**. Using Kubernetes will allow us to scale the system as needed; for example, if multiple users are interacting with the assistant simultaneously, Kubernetes can **automatically spin up additional pods** to handle the load [4], ensuring no user gets a busy signal or slow response. All user data (customer info, logs, etc.) will be stored in cloud databases (no local files), which not only supports scalability but also makes it easier to backup and maintain data securely. We'll implement horizontal scaling for the voice assistant service and possibly utilize a load balancer to distribute incoming requests (especially important if voice interactions come via phone calls or web sockets from the chat). GCP's cloud storage and possible use of a CDN will ensure the service is reliable and globally accessible with minimal latency.

Additionally, deploying on GCP means we can take advantage of their security and monitoring tools. We will secure all communications (HTTPS for the dashboard, secure tokens for API access, etc.) and ensure sensitive data (like customer contact info or recordings) are properly protected. Cloud deployment also simplifies integrating with Google services (like Calendar and possibly Dialogflow if needed) since everything is within the Google ecosystem.

System Architecture

Architecture Overview: The system is composed of a **Python backend** (the "brain" of the voice assistant) and a **Node.js frontend** (the business dashboard web app). These communicate over secure APIs. Below is a breakdown of components:

- **Voice Assistant Backend (Python):** This will be built using Python (for example, leveraging frameworks like FastAPI or Flask for web endpoints, and possibly an AI framework for dialogue management). It includes:

- *Speech Processing:* Utilizes an STT module to convert incoming audio to text, and a TTS module to synthesize responses. For STT, we consider open-source models like **OpenAI Whisper** or **Wav2Vec2**, aiming for realtime transcription accuracy under noisy conditions [8]. For TTS, we will integrate a high-quality model such as **Resemble's Chatterbox** or **Coqui TTS**, which are open source and capable of near human-like speech output [6]. The combination allows the assistant to *hear* and *speak*.
- *Natural Language Understanding & Dialogue:* This layer interprets the transcribed text and determines the appropriate response or action. We may utilize a conversational AI library or framework (for example, **Rasa Open Source** for intent recognition and dialogue management, or a lightweight LLM-based approach for flexibility). The assistant needs to handle intents like scheduling an appointment, checking the calendar, updating a booking, answering basic FAQs (e.g., "What services do you offer?"), and small talk. The dialogue manager will manage context – e.g., if the user says "Schedule Mrs. Smith next Tuesday at 3pm for a pipe repair," the assistant will confirm the details, check the calendar, and so on. If something's missing (like the date wasn't fully specified), it will ask clarifying questions.
- *Calendar Integration:* A sub-module will interface with the **Google Calendar API**. It will use the business's Google account (authenticated via OAuth) to **check availability and create events**. For example, when an appointment is being scheduled, the backend will form a query to Calendar: "Is there any event between 3:00–3:30pm on Tue, Nov 10?" [9]. If free, it books the slot by creating a new event with the customer's name and service details [2]. If busy, it can query for the next available slot or a list of open slots that day to suggest alternatives [7]. This tight coupling with Google Calendar means the assistant always works with real-time schedule data, and any manual changes on the calendar (by the owner) will be seen by the assistant as well.
- *Database & Business Logic:* The backend will have a database connector (to a cloud database). This handles reading/writing **customer info, appointment records, and logs**. When a scheduling request comes in, it checks the database for the customer's record (matching by phone or name); if found, it retrieves it so the assistant doesn't ask for info we already have (address, etc.), making the interaction smoother. It also logs each interaction (transcripts, timestamps, outcome of conversation). Business logic rules (like business hours, types of services and default durations, etc.) are enforced here as well – e.g., if a user tries to book outside of working hours or if double-booking occurs, the assistant will handle that scenario gracefully (perhaps by apologizing and offering a different time).

- *Notification Service:* The Python backend will also trigger notifications via external APIs. For SMS, we'll integrate **Twilio's REST API** to send text messages. For example, after successfully booking an appointment, the backend can automatically send a confirmation SMS to the customer with the appointment details and a link to reschedule if needed (this could be a future enhancement). Or send an SMS to the business owner for certain events (new lead alerts, etc.). Twilio supports two-way messaging, so if a customer replies "Yes" to confirm or "No" to cancel, we can catch that via webhook and update the appointment accordingly [10] [11]. (Initially, we might implement one-way alerts and add two-way interactivity later on.)

- **Frontend Dashboard (Node.js + Web Framework):** The front end will likely be built with a Node.js framework (for example, an Express.js backend serving a React or Vue single-page app). Its responsibilities:

- *User Authentication:* Ensure only authorized business users can access the dashboard (simple login, or Google auth if we tie it to their Google account).

- *UI Components:* Provide pages or views for conversation logs, calendar/appointments, analytics, and settings. We will use modern responsive design so it works well on desktop and mobile. The calendar view might use an existing calendar UI component to display the schedule. Analytics might use a chart library for visualizations.
- *Voice Interface:* For the owner to converse with the AI via the dashboard, we can utilize the browser's microphone. The app could either use the Web Speech API for local speech recognition or, more consistently, capture audio and stream it to the Python backend for transcription (ensuring the same STT model and understanding is used across voice channels). We may implement a **push-to-talk button** on the dashboard: the owner presses it, speaks a command (like "Show me tomorrow's appointments" or "Schedule a new job for John on Monday at 2pm"), the audio is sent to the backend which returns the AI's answer (which could be text plus an audio URL). The dashboard then plays the audio response and/or displays the textual response. This essentially creates a two-way voice chat between the owner and the AI assistant through the web app. Making the app a Progressive Web App (PWA) allows it to send push notifications (for alerts) and be used offline to some extent (e.g., access cached data or records if no internet, though voice queries would need connectivity to the backend).
- *Data Fetching:* The frontend will call backend APIs to get data: e.g., fetch the list of customers, fetch stats for analytics, load log transcripts, etc. We will design RESTful endpoints (or GraphQL) for these. Some data like appointment list could also come directly from Google Calendar's API (through the backend or possibly via a direct embed of Google Calendar if convenient, but direct embed likely not flexible for our needs).

- *Logging and Error Display:* If the AI encounters errors (like couldn't reach Google Calendar), those should be communicated to the user via the UI. The dashboard might have a "Notifications" section for system alerts.

- **Cloud Services & Infrastructure:** On GCP, we will set up a **Kubernetes cluster** to host both the backend and frontend (likely as separate deployments/services). A Load Balancer will route external requests: e.g., web app traffic goes to the Node frontend, voice API or webhook traffic goes to the Python service. We will also use GCP's Cloud Storage or a database service for storing data. A likely choice is **Firebase/Firestore** for its ease of integration with web and ability to store JSON-like docs (for customers, logs, etc.), or **Cloud SQL (PostgreSQL)** if a relational approach is preferred (e.g., for structured queries and easier relational mapping of customers to appointments). Firestore would allow real-time updates (could even sync to frontend), but Cloud SQL might be more familiar for complex querying. We will evaluate based on the complexity of queries needed for analytics.
  For sending SMS and possibly handling voice calls in the future, we use external APIs (Twilio). For voice calls (future scope), Twilio Voice or a SIP integration could direct phone calls to our AI system, which then treats audio similarly to the web voice input. This would effectively let the AI answer phone calls from customers, talk to them, and schedule appointments – a powerful extension once the core system is stable.

- **Security:** We will implement secure authentication and data protection measures. The Google Calendar integration requires OAuth tokens which we will store securely (possibly encrypted in the database). API communications will use HTTPS. The dashboard will have user auth (at least a password login, or using Google account login). Role-based access could be added if the business grows (e.g., owners vs technicians with limited views). All PII (customer phone, addresses) will be stored securely and we'll ensure compliance with relevant data protection practices.

In summary, the architecture is a **modular voice assistant service** with a web interface, connected through cloud services. It takes advantage of cloud scalability and integrates with third-party APIs (Google Calendar, Twilio) to provide a comprehensive solution (from voice interaction to scheduling to notifications).

Implementation Plan and Milestones

We will implement the project in stages, delivering a Minimum Viable Product (MVP) first, then adding enhancements. Below is a high-level plan:

- **Phase 1: MVP – Core Voice Scheduling System**
  Timeline: (~8-10 weeks)
  Features: Focus on getting the end-to-end voice scheduling working for the business owner through the dashboard. This includes setting up the core backend services, connecting to Google Calendar, and a basic dashboard.
  Tasks: In this phase, we will set up the project infrastructure (repos, CI/CD, cloud environment), implement speech recognition and synthesis in the backend, implement the conversation flow for booking an appointment (happy path scenarios), integrate Google Calendar (read/write events), and develop a simple web UI that allows the owner to initiate a voice conversation or view the day's appointments. The customer database can be basic at this stage (even a simple in-memory or small cloud DB just for storing contacts and preventing duplicates). SMS notifications might be deferred unless easy to plug in. The goal is that by end of Phase 1, the owner can press a button in the web app, speak "Schedule John Doe for a pipe repair on Friday at 2pm," and the assistant will understand it, create the event on Google Calendar, and confirm back (voice or text). We will also ensure that if the assistant is unsure of details, it asks for clarification (e.g., "What is the address for John Doe?" if not on file). Basic error handling (like if Google Calendar API fails or internet outage) will be included. At the end of this phase, we'll have a **working MVP** that showcases the core functionality.

- **Phase 2: Enhanced Functionality & Dashboard Expansion**
  Timeline: (~6-8 weeks)
  Features: Now we add the next layer of features: robust customer database integration, SMS notifications, and a more full-fledged dashboard with analytics. Also improve the AI's conversation handling with more use cases.
  Tasks: Implement storing every appointment and customer detail in the cloud database persistently and building the auto-populate capability for returning customers. Introduce Twilio SMS integration: e.g., send an SMS to the business when the AI schedules an appointment or when a new website chat request comes in; send confirmation texts to customers with a thank-you and appointment details (optionally, allow them to confirm or reschedule via reply – if we do this, handle reply webhooks). Expand the dashboard: add pages for viewing the customer list and their service history, an analytics page with charts (like number of jobs this month vs last, revenue estimates, common services, etc.), and a settings page for configuration (business hours, notification preferences, etc.). We will also refine the voice assistant's dialogue – handle edge cases like overlapping appointments (suggest next available slot automatically), cancellations ("Cancel Mr. Smith's appointment on Monday"), and possibly answering frequently asked questions (we can hard-code a small Q&A like business hours, service area, basic pricing info, so the assistant can answer questions like "Do you charge for estimates?"). During this phase, we'll incorporate feedback from the MVP testing to improve the speech recognition accuracy (perhaps by choosing a different model or tweaking audio

processing) and voice response quality (e.g., adjusting speaking rate, adding conversational fillers for naturalness). We should also implement more robust error logging and monitoring in this phase.

- **Phase 3: Additional Channels and Scalability**
  Timeline: (~Beyond 8 weeks, ongoing iterations)
  Features: This is the forward-looking phase where we introduce the AI to customer-facing channels and ensure the system can scale to production levels.
  *Tasks: Integrate the voice assistant into a website chat widget or chatbot interface. This could involve developing a small web widget that uses text (and possibly speech via the browser) to communicate with the same backend AI. We'll need to implement user-facing conversation flows (which might be slightly different in tone from the owner-facing ones). Another task is to enable phone call integration – using a service like Twilio Voice to forward incoming calls to the AI. This requires the backend to handle streaming audio in real-time and possibly tweaking the dialogue to be suitable for a phone call context. On the infrastructure side, set up Kubernetes Horizontal Pod Autoscaling*** so that if usage increases, new instances of the backend are automatically launched [4]. We will conduct load testing to verify the system can handle concurrent conversations (for example, two customers chatting at once plus the owner using the system). Also, focus on optimization: ensure the STT/TTS models are running efficiently (possibly use GPU acceleration on GCP for faster processing of audio). We might containerize the models separately or use a service if latency becomes an issue. In this phase, we also aim to refine the analytics with more data and possibly introduce predictive insights (for example, the assistant could eventually analyze common issues by season, etc., if we have enough data). Security hardening (penetration testing, ensuring compliance with privacy laws if applicable) will also be part of this phase.

Throughout all phases, we will maintain thorough documentation and testing. Unit tests for the backend logic (like scheduling logic, database ops) and integration tests (end-to-end simulation of a conversation) will be developed. We'll also have user testing sessions with the business owner to collect feedback on the assistant's effectiveness and the dashboard's usability.

Regular milestones/demos are planned at the end of each phase to show progress. By the completion of Phase 2, the system should be fully usable internally by the business. Phase 3 ensures it is robust, scalable, and ready for customer-facing deployment. Post-development, we will continue to monitor system performance and gather user/customer feedback for continuous improvement.

Risks and Mitigations

- **Speech Recognition Accuracy:** If the chosen STT model struggles with certain accents or noisy backgrounds (e.g., the plumber might talk to the assistant from a job site with noise), it could lead to misunderstandings. **Mitigation:** Choose a model known for robustness in real-world audio (Whisper has high accuracy on diverse audio [8]) and possibly allow the system to confirm critical details ("Did you say 3 PM?") if confidence is low. We can also incorporate a feedback loop where the owner corrects the assistant if it gets something wrong, helping improve the model or at least logging common misrecognitions to address.

- **Integration Reliabilities:** The system depends on external APIs (Google Calendar, Twilio). Outages or API changes could impact functionality. **Mitigation:** Implement graceful error handling – e.g., if Calendar API is down, the assistant might say "Sorry, I'm having trouble accessing the calendar. I will

save this request and confirm the schedule shortly." Also, keep these integrations updated and use official SDKs which handle token refresh, etc.

- **Security and Privacy:** Handling customer contact info and possibly recording conversations means we must protect data. **Mitigation:** Use encryption for data at-rest and in-transit. Follow best practices for storing API keys (in secure GCP Secret Manager, not in code). Provide an easy way to export or delete customer data if needed (compliance). Limit access to the dashboard with proper auth. Regularly update dependencies to patch vulnerabilities.

- **User Adoption:** The business owner and customers might need time to trust and effectively use the AI. If the AI makes errors in scheduling or misunderstandings, it could frustrate users. **Mitigation:** Start with the owner-facing use (so the owner gets comfortable with it), and gradually introduce it to customers. Provide training to the owner on how to speak queries for best results. We can also start the assistant with a somewhat limited scope (just scheduling and FAQs) and not critical tasks until it's proven. Logging and reviewing interactions will help identify and fix any problematic responses early. Also, allow easy fallback to human: e.g., if the AI isn't confident, it can flag the conversation for the owner to follow up.

By carefully managing these risks, we aim to deliver a reliable and helpful assistant that enhances the plumbing business operations without causing disruptions.

Conclusion

This project will result in a powerful voice assistant tailored for the plumbing business, automating many of the routine scheduling and customer interaction tasks. By utilizing open-source AI models and cloud services, we keep costs manageable while retaining control over data and customization. The **OUTLINE.md** above has detailed the project plan, and the **BACKLOG.md** below will break down the specific tasks and user stories required to implement this plan. With a clear roadmap and the flexibility to scale on GCP, the voice assistant is poised to improve efficiency, responsiveness, and data-driven decision-making for the business. We will now translate the plan into a backlog of actionable items.

Backlog – Voice Assistant Project Tasks and User Stories

(This backlog lists the tasks and GitHub-style issues needed to implement the voice assistant project. It is organized by functional areas and priority. Each item includes a short description and references to relevant details from the plan or external sources.)

Setup & Infrastructure

- **DevOps: Repository and Environment Setup** – *Initialize project structure.* Create two code repositories (or a monorepo) for **backend (Python)** and **frontend (Node.js)**. Set up version control and CI/CD pipelines for automated testing and deployment. Define the development environment

(linting, formatting, basic project README). *(Outcome: Developers can build and run backend and frontend locally, with CI ensuring quality checks.)*

- **DevOps: Containerization** – Write Dockerfiles for the Python backend and Node frontend. Ensure containers can run the app, including necessary dependencies (AI models, etc.). Test building and running containers locally. This is preparation for cloud deployment on Kubernetes.

- **Cloud: Google Cloud Project & Kubernetes Cluster** – Provision a GCP project for the app. Set up a Kubernetes cluster (GKE). Configure necessary services (e.g., enable Google Calendar API, Secret Manager for storing API keys/secrets, etc.). Deploy a simple "Hello World" of both services to verify the pipeline: e.g., the frontend serves a static page, backend has a simple health-check API. *(Outcome: Cloud infrastructure is ready for app deployment, and we have verified connectivity and permissions.)*

- **Infrastructure: Database Setup** – Choose and set up the cloud database. Likely options: **Firestore** (NoSQL) or **Cloud SQL** (Postgres). For now, set up the instance or database and create initial schemas/collections for Customers, Appointments, and Logs. *(Outcome: The app can read/write to the database. Example: a test to create a dummy customer entry and retrieve it.)*

- **Security: Authentication & OAuth Config** – Set up OAuth 2.0 credentials for Google Calendar access (create a client ID/secret in Google Cloud Console). Also, implement a basic auth for the dashboard (could be a simple username/password or using Google Sign-In). Store credentials securely (e.g., use environment variables or GCP Secret Manager for sensitive keys). Ensure the OAuth consent screen and scopes for Calendar API are configured. *(Outcome: The app can obtain authorization to manage the Google Calendar on behalf of the user* [9] *.)*

Voice Assistant Backend (Python)

- **Backend: Speech-to-Text (STT) Module** – *User Story:* "As the voice assistant, I need to convert the user's spoken words into text so that I can understand commands." **Implementation:** Integrate an open-source STT engine. Evaluate **OpenAI Whisper** vs **Wav2Vec2** for real-time use. Whisper offers strong accuracy on diverse audio [8], while Wav2Vec2 (with streaming) offers low-latency performance [5]. We may start with Wav2Vec2 for responsiveness. Install the model and set up a function to input an audio stream or file and return transcribed text. Ensure it can handle at least English speech with good accuracy. Test with a few sample audio clips (simulate different voices/noises) to fine-tune parameters. *(Outcome: Given an audio input (from file or mic), the system returns a text transcription.)*

- **Backend: Text-to-Speech (TTS) Module** – *User Story:* "As the voice assistant, I want to respond in natural-sounding speech so that users have a comfortable, human-like conversation experience." **Implementation:** Integrate an open-source TTS model. We plan to use **Chatterbox (Resemble AI)** or a similar model known for high quality output [6]. Set up the model to generate audio from text responses. Ensure the voice is clear and professional (we might choose a neutral North American English female or male voice unless we can customize). The model should generate audio quickly (target < 1 second per sentence for real-time feel). Develop a function that takes a text string and returns/streams an audio waveform (e.g., as MP3 or WAV). Test by feeding sample sentences and listening to the output. *(Outcome: The assistant can speak any given response text with a natural voice.)*

- **Backend: Conversation Logic & NLP** – *User Story:* "As a user, I can speak to the assistant and it will understand my intent (e.g., scheduling a new appointment) and respond appropriately or perform an action." **Implementation:** Develop the core **dialogue manager**. This could use a rule-based approach initially: define intents like "schedule_appointment", "cancel_appointment", "check_schedule", "greet", etc., and use keyword matching or a small ML model to classify user utterances. Alternatively, integrate **Rasa** to define intents/entities (customer name, date/time, service type) and stories for dialogue flow. At first, implement the **appointment booking flow**:

  - Trigger: user says something that indicates they want to schedule (e.g., "I need to book an appointment" or "Can you schedule John Doe for a pipe repair?").
  - The assistant will then ensure it has all info: name, service needed, address, preferred date/time. Use slot-filling strategy – ask for missing pieces ("What is the customer's name?", "What service is this for?", "When would they like the appointment?" etc.).
  - Once all details are gathered, the assistant confirms the details back to the user for verification.
  - It then invokes the calendar integration to check availability on that date/time. If available, proceed to book (and skip to event creation task); if not, go into a sub-flow of suggesting an alternate time ("That time is unavailable. Would 4 PM work instead?") [7].
  - After booking, the assistant should provide a confirmation ("Got it, I've booked the appointment for John Doe on Friday at 2 PM [2].").

Implement this flow with error handling (if user says something out of context, have fallback responses like "I'm sorry, I didn't catch that. Could you repeat or rephrase?"). For now, focus on this primary use case; additional intents (cancellation, checking today's schedule, etc.) can be added after the basic booking works. *(Outcome: The backend can carry a multi-turn conversation to schedule an appointment, maintaining context and storing the necessary info.)*

- **Backend: Google Calendar Integration** – *User Story:* "As the assistant, I need to check and update the Google Calendar to avoid double-booking and to record new appointments." **Implementation:** Use Google Calendar API with the OAuth token of the user. Implement functions for: checking free/busy for a given time range, creating an event, updating or deleting an event (for reschedule/cancel). In the booking flow, after gathering details, call the **Calendar API's events.list** (with timeMin/timeMax as the requested slot) to see if something exists [9]. If free, call **events.insert** to create the event with details (title like "Plumbing Service – [Customer Name]", date/time, duration, description with address and service type, and perhaps a default notification) [2]. If busy, perhaps find the next available slot on that day or the same time next day, etc., to suggest. For now, a simple strategy: increment in 30-minute steps until a free slot is found on the requested date. Also, implement retrieval of upcoming appointments for the dashboard (e.g., list events for the next day or week). Test the integration with a test Google Calendar (initially in a dev environment). *(Outcome: The assistant can programmatically read and write calendar events, enabling automated scheduling.)*

- **Backend: Data Persistence (Customers & Appointments)** – Now that we have a DB set up, implement logic to **create or update customer records** during conversation. For example, when scheduling an appointment, after confirming details, upsert the customer info (name, phone, address, any notes) into the Customers collection/table if not already present. Also store an entry for the Appointment in our DB with relevant details (even though Google Calendar has it, storing internally allows richer analytics and cross-link with customer records). Include a field in Appointment linking to the Google Calendar event ID for reference. Also implement a lookup at conversation start: if user initiates scheduling and provides a name or phone, we check DB for

existing customer to personalize the dialogue ("Sure, I found Jane Doe in our system. Last service was a water heater repair on Oct 5. What would Jane like this time?"). This task also involves designing the schema for these records. *(Outcome: Customer and appointment data are stored in our system's database, enabling auto-fill of known info and historical record-keeping.)*

- **Backend: SMS Notification Triggers** – *User Story:* "As the business owner, I want to get a text message when important events happen (new appointment booked, etc.), so I'm immediately aware even if I'm not looking at the dashboard." **Implementation:** Integrate **Twilio API** using their Python SDK. Set up configurations for the Twilio account (account SID, auth token, from phone number) as secrets. Identify points in the flow to trigger SMS: one likely point is after a new appointment is successfully booked by the AI – send an SMS to the owner: "New appointment booked for [Customer Name] on [Date] at [Time] for [Service]." Additionally, when a web chat request or estimate request is captured (in Phase 3, perhaps), send an alert. Implement a utility function `send_sms(to_number, message)` and call it with appropriate content. Also consider customer notifications: e.g., after booking, send the customer a confirmation text if their number was provided. (This might require asking "Can I send a confirmation text to the customer?" during the conversation, or assume yes if number available.) Use Twilio's API to send these. We can later expand to handle incoming replies (Twilio can hit a webhook on our backend if someone replies "Yes" or "No"). For now, implement outbound texts. Test by sending to a sandbox or test number. *(Outcome: The system sends SMS alerts for new bookings or key events, improving communication [3].)*

- **Backend: Logging & Error Handling** – Implement comprehensive logging of events and errors. Every conversation turn should be logged to the database (or at least to a log file) with timestamp, user utterance, assistant response, and any action taken (event created, etc.). This will feed the dashboard's log view. Also log errors/exceptions (e.g., unable to reach Calendar API, or STT transcription failed) for debugging. Possibly use a structured logging library. Ensure that sensitive info (like OAuth tokens or Twilio creds) are not logged. This task is ongoing, but initial setup is needed early. *(Outcome: A log of interactions and errors is maintained, which will be displayed in the admin dashboard for transparency.)*

Frontend Dashboard (Node.js)

- **Frontend: Dashboard Skeleton** – Set up a Node.js application (could use Next.js for server-side rendering or a simple Express with static frontend). Initialize a frontend framework (React recommended for dynamic UI, or use a lightweight alternative). Create the basic navigation/layout for the dashboard – e.g., a sidebar or header with links to "Calendar", "Customers", "Analytics", "Logs", "Settings". Implement dummy pages for each to confirm routing works. Ensure the app is responsive (use a CSS framework or custom styling). Protect the routes with a login (for now, even a hardcoded password in config, or integrate with Google OAuth if planning that). *(Outcome: A running web application that the owner can log into and navigate, albeit with mostly placeholder content initially.)*

- **Frontend: Calendar View Integration** – On the dashboard, create a **Calendar page** that shows upcoming appointments. For MVP, this could be a simple list or table of events for the current day/week, pulled from either the Google Calendar (via backend API) or from our Appointments DB. Implement an API endpoint on backend like `/appointments?from=DATE&to=DATE` that returns JSON of appointments, and have the frontend fetch and display that. If time permits, use a calendar component (there are React calendar libraries) to render a day/week view and populate events. This

view allows the owner to see what's scheduled and maybe click on an event to see details or delete/reschedule (the reschedule feature can be a future addition where the owner could instruct the AI to change something). *(Outcome: The business owner can see all scheduled jobs in a given timeframe on the dashboard, fulfilling the need for quick schedule overview.)*

- **Frontend: Voice Interaction UI** – Implement the interface for the owner to converse with the assistant via the dashboard. This likely includes a **microphone button** to start/stop recording. When pressed, capture audio (using HTML MediaRecorder or Web Audio API) and send it to the backend (possibly chunked via WebSocket for continuous streaming, or as a whole blob via an HTTP request if simpler to start). While processing, show a "listening..." indicator. Once the backend responds, play the audio reply for the owner and also display the transcript of the assistant's answer as text (so the owner has a visual confirmation of what was said). Essentially, create a chat-like interface on the dashboard where each exchange (owner query and AI answer) is logged. This will also be useful for the owner to refer back or correct the AI if needed. Start with a basic implementation: press-to-talk that handles one turn at a time (full duplex streaming can be an enhancement later). *(Outcome: The owner can press a button, speak a request, and hear the assistant's spoken response through the dashboard, enabling hands-free operation.)*

- **Frontend: Logs Page** – Create a page that displays the **conversation logs** stored by the backend. This might be a simple table or list: each entry shows timestamp, participant (User or AI), and message content. Possibly group by session or conversation. Provide filters like date or search by keyword (to find when a particular customer called, for example). This helps the owner review interactions. For MVP, even just dumping the last N interactions is fine. Ensure the frontend calls an API like `/logs` to retrieve this data. *(Outcome: Owner can review what the AI has communicated with users, adding transparency and trust in the system.)*

- **Frontend: Analytics Page** – Implement a basic Analytics dashboard. Define a few key metrics to display: e.g., *Total appointments this month*, *New customers this month*, *Most requested service*, *Revenue (if we track price)*, etc. Use the data from the database to compute these. Possibly the backend can provide an endpoint `/analytics` that returns precomputed stats (or compute on the fly). Also include any interesting insights like number of appointments by zip code or charts of appointments over time. This page can start simple (a few stats) and be expanded. Use a chart library for visual appeal (if time allows, a bar chart of weekly jobs done, etc.). The analytics fulfill the requirement of providing **insights sorted by date and other factors**. *(Outcome: The business can see at-a-glance how the business is doing via the data collected, helping identify trends such as busy periods or popular services.)*

- **Frontend: Settings Page** – Provide a UI for configuration. This includes connecting the Google Calendar (maybe showing which calendar/account is linked, with option to re-auth), Twilio settings (phone numbers for alerts, toggling which events trigger SMS), business info (hours of operation, list of services offered and their typical durations for scheduling templates), and user management if needed (maybe just one user for now). The settings page will read/write to backend endpoints (like GET/POST to a config). For example, store business hours to prevent scheduling outside those times. This also might include toggles for the AI behavior (like "allow double-booking: yes/no" or "auto-send confirmation to customer: yes/no"). *(Outcome: The owner has control over key parameters of the system without modifying code.)*

- **Frontend: PWA Setup** – Make the web app a Progressive Web App. Add a manifest.json with app name, icons, and specify it's installable. Implement a simple service worker for caching static assets and enabling offline page (maybe an offline fallback page). Test that on Android or Chrome the "Add to Home Screen" appears and the app can launch full-screen. This increases convenience for the owner to use it on mobile like a native app. *(Outcome: The dashboard can be installed on mobile and has basic offline support, providing a better mobile experience.)*

- **Frontend: Testing & UX Polishing** – Conduct usability testing of the dashboard. Ensure that the voice interaction is intuitive (maybe add guidelines like "Press and hold spacebar to talk" as a shortcut, etc.), the information is presented clearly (use readable tables, etc.), and the design looks professional (company branding or at least a clean look). Fix any bugs such as UI not updating after an appointment is added, or time zone issues in calendar display. Also test on multiple devices/browsers for compatibility. *(Outcome: A smooth, user-friendly interface for the business with minimal bugs.)*

Additional Features & Future Tasks

- **Feature: Website Chatbot for Customers** – (Future) Develop a **customer-facing chat interface** that can be embedded on the company's website. This could be a small web widget (HTML/JS) that connects to our backend via WebSocket or HTTP. It will allow text chat initially: customers ask questions or request appointments, and the assistant responds (text or maybe voice). Use the same backend logic but possibly a different dialogue flow (more explanatory style for customers). Include a human fallback option (e.g., "Would you like us to call you?"). This task involves frontend work to create the widget and backend work to handle multi-user sessions (each chat is a separate session context). Security: ensure one customer can't see another's data – use session IDs. *(Outcome: Website visitors can interact with the AI assistant in real-time, generating leads and bookings that go directly into our system.)*

- **Feature: Phone Call Voice Assistant** – (Future) Integrate Twilio Voice or similar to handle phone calls. Provision a phone number that, when called, forwards the call's audio to our voice assistant. The assistant then converses as if it's a live agent. Twilio can send audio via webhook (either real-time streaming or in chunks). Implementing this requires our STT to handle streaming audio input continuously and respond in a timely manner (possibly using Twilio's <Pause> and <Say> in TwiML for responses, or Twilio's streaming API). Essentially, this would let customers call a number and schedule via the same AI. We'd reuse much of the logic, but need to handle telephony specifics (like greeting the caller, detecting hangups). This is a complex but high-value feature once the core system is stable. *(Outcome: The AI can serve as a virtual receptionist on calls, booking appointments without human intervention.)*

- **Optimization: STT/TTS Performance** – (Future) As usage grows, we may need to optimize the speech models. For STT, possibly fine-tune the model on domain-specific vocabulary (plumbing terms, local place names) to improve accuracy. Consider using a smaller model or a streaming endpoint for faster results if needed. For TTS, ensure voice is pleasant – maybe even create a custom voice that matches the business brand (could train a voice model on the owner's voice or a professional voice actor, if open source tools allow). Also monitor CPU/memory usage of these models in production; consider using GPU instances or model pruning/quantization for efficiency.

(Outcome: The assistant's voice interaction becomes faster and more accurate over time, providing a better user experience.)

- **Analytics: Advanced Insights** – (Future) Expand the analytics to be more insightful. For example, integrate with Google Maps API to plot service locations on a map (see where most jobs are coming from), or analyze the text of customer inquiries to identify common problems (could help the business target certain services). If call recordings or transcripts are stored, use NLP to gauge customer sentiment or satisfaction. These advanced analytics can be added as value-add features. *(Outcome: The business gains deeper understanding of operations, such as peak demand times, common service requests, repeat customer rate, etc., driving data-informed decisions.)*

- **Multi-Language Support** – (Future) If the business serves a multilingual community, consider enabling Spanish or other language interactions. This would involve using multilingual STT/TTS models (many open source models support multiple languages [12] [13]). The conversation logic might need translation or separate intent models. This can broaden the assistant's usability. *(Outcome: Non-English speaking customers could interact with the assistant, expanding accessibility.)*

- **Testing: End-to-End and User Acceptance** – *Ongoing:* Continuously write tests for new features. Before go-live, do an end-to-end test simulating an entire booking via voice, and test fail cases (calendar conflict, etc.). Also gather the business owner's feedback – does the AI handle the conversations as expected? Tweak responses to be more friendly or professional as needed (e.g., ensure it says "Thank you for choosing our services" at the end of a call). This task will be repeated as new features roll out.

Each of these backlog items can be tracked as GitHub issues or tasks, and as we implement them, we will update their status. The above list covers all major features requested (voice interface, scheduling, database, PWA dashboard, SMS alerts, cloud deployment) and future enhancements. By tackling these iteratively, we will build a robust voice assistant platform for the plumbing business.

References (from research):

- Elizabeth C., *"Ultimate Guide - The Best Open Source AI Models for Voice Assistants in 2025"* – on modern open-source TTS capabilities [6].
- AssemblyAI Blog, *"Top 8 open source STT options for voice applications in 2025"* – on choosing STT for real-time voice assistants [5].
- Sobemekun O. M., *"How I Built an AI Voice Assistant That Books Dental Appointments Automatically"* – describing a similar appointment-booking flow with Google Calendar [9] [2] and the benefit of reducing missed calls [1].
- Twilio Use-Case, *"Appointment reminder texts and emails"* – guidance on sending appointment notifications and confirmations via SMS [3].
- Kubernetes Documentation – on horizontal pod autoscaling to handle increased load without downtime [4].

---

[1] [2] [7] [9] How I Built an AI Voice Assistant That Books Dental Appointments Automatically (No Code) | by Sobemekun Oluwadamilare Matthew | Medium
https://medium.com/@darey/how-i-built-an-ai-voice-assistant-that-books-dental-appointments-automatically-no-code-06bb33aeca80

[3] [10] [11] Appointment reminder texts and emails | Twilio
https://www.twilio.com/en-us/use-cases/appointment-reminders

[4] Autoscaling in Kubernetes
https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/

[5] [8] Top 8 open source STT options for voice applications in 2025
https://www.assemblyai.com/blog/top-open-source-stt-options-for-voice-applications

[6] Best Open Source Text-to-Speech Models in 2025: A Developer and Enterprise Guide
https://www.resemble.ai/best-open-source-text-to-speech-models/

[12] [13] Ultimate Guide - The Best Open Source AI Models for Voice Assistants in 2025
https://www.siliconflow.com/articles/en/best-open-source-AI-models-for-voice-assistants