ChatGPT

Comprehensive Task List for Code Quality, Features, QA, and GCP Deployment

Below is an exhaustive list of GitHub issues (with sub-tasks) required to improve the AI Telephony Service & CRM project. These tasks cover code improvements, feature completion, quality assurance, and efficient deployment to Google Cloud (using Docker and Kubernetes). Each issue is derived from the project's design documents and backlog, ensuring alignment with the RavDevOps coding standard and overall project goals.

Deployment & Infrastructure (Docker, Kubernetes, GCP)

- **Containerization & GCP Deployment:** Containerize the backend and deploy on Google Cloud for scalability [1].
- *Sub-task:* Create a production-ready **Dockerfile** for the backend service (ensuring deterministic builds and minimal image size) [2] [1].
- *Sub-task:* Write Kubernetes manifests (or a Helm chart) to run the container on **Google Kubernetes Engine (GKE)**. Include a horizontal pod autoscaler with a **60-second idle timeout** before scaling pods down, so instances can handle back-to-back requests without frequent cold starts (per the 60s reuse requirement).
- *Sub-task:* **Provision GCP infrastructure**: set up the project, enable required services (Cloud Run/Registry, GKE, IAM) [1], and create a Container Registry (or Artifact Registry) for images.
- *Sub-task:* Implement secure **secret management** for API keys and credentials (Twilio, OpenAI, Google Calendar OAuth, etc.), using GCP Secret Manager or Kubernetes secrets (no sensitive config in code or images).
- *Sub-task:* Ensure the **Firestore/Cloud SQL database** (whichever chosen for persistence) is provisioned and configured. This includes setting up VPC access or service accounts for the backend to connect to the database [3].

- *Sub-task:* Write a **Deployment Checklist** to verify all cloud resources (DNS, SSL, Twilio webhooks, OAuth redirect URIs) are configured before go-live [4].

- **CI/CD Pipeline for Deployment:** Automate building, testing, and deploying to GCP.

- *Sub-task:* Configure **Continuous Integration** to run tests and linting on each commit (if not already in place) [5]. On a successful merge to main, build the Docker image and push to the registry.
- *Sub-task:* Set up **Continuous Deployment** (or a manual deploy step) to GKE. This may involve Terraform or `kubectl` scripts to apply Kubernetes manifests, or using Cloud Build/Deploy pipelines.
- *Sub-task:* Implement health checks and monitoring in the deployment (readiness/liveness probes for Kubernetes, uptime checks via GCP Monitoring) to ensure smooth rollouts.

- *Sub-task:* Document the deployment process in `DEPLOYMENT_CHECKLIST.md` (covering build steps, environment variable configuration, and rollback procedures) [4].

- **Resource Optimization & Scaling:** Tune the service for cost-effective performance on GCP.

- *Sub-task:* Use Kubernetes autoscaling to **reuse warm instances**: allow each pod to serve multiple sequential calls within a 60-second window before scaling down. This prevents spawning a new container for each call, improving efficiency (the cluster autoscaler should wait at least 60s of no traffic before scaling to zero).
- *Sub-task:* Limit each voice session to a single container (if needed) but allow moderate concurrency per node if memory/CPU permit, to maximize resource usage. Determine appropriate **pod resource requests/limits** to handle STT/TTS processing without starvation.
- *Sub-task:* Enable **Cloud Logging and Monitoring** on GKE. Ship application logs to Google Cloud Logging and set up custom metrics (from the `/metrics` endpoint) in Cloud Monitoring [6] for insight into performance and scaling.
- *Sub-task:* Test deployment with low traffic, then perform a load test (see **Testing & QA** section) to verify that autoscaling behaves as expected (new pods come up under load, and scale down after 60s idle). Adjust node pool size or cooldown configuration based on results.

Code Quality & Maintainability

- **Enforce Coding Standards:** Align the codebase with the RavDevOps Engineering Code Culture & Safety Standard [2].
- *Sub-task:* Run automatic formatters and linters (e.g. **Black** for Python code and **ruff/Flake8** for linting) and fix any style issues [7]. All code should be "boringly" consistent and easy to read [8].
- *Sub-task:* Add a **pre-commit hook** or CI step to reject code that doesn't meet formatting or linting rules. The main branch must remain free of new warnings or lint errors [9].
- *Sub-task:* Ensure naming conventions, module organization, and function lengths adhere to guidelines (e.g., small functions, clear names, one concept per module). Refactor any overly complex or lengthy functions for clarity and safety.
- *Sub-task:* Increase inline documentation: add docstrings for all public functions/classes and ensure complex logic (like the conversation state machine) is explained in comments or markdown docs [8]. This aligns with the standard's emphasis on explicit design documentation for non-trivial features.

- *Sub-task:* Verify that builds are **deterministic** – for example, pin dependency versions in `pyproject.toml` or `requirements.txt` to avoid drifting, and ensure the Docker build produces the same artifact each time [2]. Any nondeterminism in tests or builds should be identified and removed.

- **Static Analysis & Type Safety:** Integrate deeper static analysis to catch bugs early.

- *Sub-task:* Introduce **type checking** (using MyPy or pyright) on the Python code. Add type annotations to functions and data structures in the codebase, especially for critical functions like telephony handlers and scheduling logic.
- *Sub-task:* Run a security linter (like **Bandit** for Python) to detect common security issues (e.g., use of `eval`, weak cryptography, etc.). Address any findings (though this project mainly uses high-level libraries, it's good to check).
- *Sub-task:* Implement **policy-as-code** if possible – e.g., use CI rules to enforce test coverage thresholds, lint rules, and even dependency auditing (checking for known vulnerabilities in

packages). This automates the RavDevOps policy that coding rules and safety checks are enforced as part of the pipeline [10].

- *Sub-task:* Perform a code **cleanup of TODOs and stubs**: search the code for any `TODO`, `FIXME`, or temporary hacks and turn them into tracked issues or resolve them. This ensures no half-implemented code goes unnoticed in production.

- **Documentation & Traceability:** Improve maintainability by updating documentation.

- *Sub-task:* Keep the **API docs** (`API_REFERENCE.md`) up to date with any changes as features are completed, so that the reference remains accurate [11].
- *Sub-task:* Update the **data model docs** (`DATA_MODEL.md`) if any schema changes occur (for example, if we add fields for audit logging or new entities) [12].
- *Sub-task:* For any major feature implementations, create or update design docs in the repo (following the practice of explicit design docs). For instance, if implementing billing or multi-user authentication, write a short design note in the wiki or an `.md` file describing the approach (this aids future developers in understanding the rationale [8]).
- *Sub-task:* Ensure the **Project Wiki** (domain knowledge and use-case descriptions) remains consistent with the implementation. If the call flows or behavior deviate from the original plan, update the wiki or backlog to reflect the actual system so that documentation and code are in sync [13].

Core Voice Assistant & Telephony Features

- **Integrate Speech-to-Text (STT) and Text-to-Speech (TTS):** Finalize the voice interface so the assistant can hear and speak effectively [14].
- *Sub-task:* **Select and configure STT/TTS providers** – for example, use OpenAI Whisper for STT and a high-quality TTS voice (Google Cloud Text-to-Speech or another) [15]. Ensure the choices meet real-time performance needs (low latency for phone calls).
- *Sub-task:* Implement the **audio pipeline** from Twilio into the STT service. When Twilio sends voice audio (via webhook or TwiML `<Gather>`), forward it to the STT engine and obtain text transcripts in streaming or near-real-time fashion [14]. This may involve chunked audio streaming if supported, to keep latency low.
- *Sub-task:* Likewise, integrate TTS responses: after the dialogue manager decides a reply, use the TTS API to synthesize speech. Ensure the TTS responds quickly and the audio is delivered to Twilio via TwiML `<Say>` or `<Play>` (possibly using `<Say>` with an SSML if using Twilio's built-in voices vs. playing a pre-generated file) [16].
- *Sub-task:* **Test STT/TTS in realistic conditions:** use sample call recordings with background noise to evaluate accuracy. Tune the configuration (e.g., vocabulary hints for plumbing terms or adjust voice gender/tone for professionalism [17]). If latency is an issue, consider using faster models or partial transcriptions per utterance.

- *Sub-task:* Implement fallback behavior if STT fails (e.g., if audio is unintelligible or the STT service times out, the assistant should say a prompt like "I'm sorry, could you repeat that?" rather than just failing). Similarly, handle TTS failures by falling back to a simpler Twilio `<Say>` as a backup.

- **Dialog Flow & Intent Handling:** Refine the conversation state machine to cover all scenarios [18].

- *Sub-task:* Finish implementing **intent recognition** for all core flows: new appointment requests, rescheduling, cancellations, general questions, and emergency escalation [18]. The `ConversationManager` (`backend/app/services/conversation.py`) should handle each of these intents/stages reliably based on caller input.
- *Sub-task:* **Emergency detection rules:** finalize the list of keywords/phrases that signal an emergency (from the domain analysis) [19]. Implement this in the conversation flow such that if any are detected in the problem description, the flow immediately flags the call as an emergency and triggers the special handling (e.g., mark emergency, notify owner) [20]. Include unit tests for these keywords to avoid false negatives.
- *Sub-task:* Enhance the **rescheduling/cancellation flow**: if a caller wants to reschedule or cancel an appointment (detected via voice intent or SMS keyword), implement a branch in the dialog that confirms the details (which appointment, new date, etc.) and updates the calendar accordingly. Currently, SMS keywords "NO" or "CANCEL" trigger cancellation of the next appointment [21]; ensure a similar capability exists via voice (e.g., the caller can say "I need to cancel my appointment").
- *Sub-task:* Add an **intent for general inquiries**: for example, if the caller asks a question like business hours or pricing (something the AI can answer from a FAQ), either handle it via a small FAQ repository or gracefully defer ("I'll have the owner follow up with you on that"). This makes the assistant more robust to off-script questions.
- *Sub-task:* Review and polish the **conversation scripts**: Compare the implemented dialog prompts to the reference call scripts from Bristol Plumbing. Ensure the tone and wording match the desired professionalism and friendliness. For instance, verify the greeting includes the business name and that the wrap-up includes confirmation details and next steps [22] [23]. Adjust any prompts that are unclear or not aligned with the business's style.

- *Sub-task:* **Owner voice queries:** finalize the feature where the owner can call in and ask questions (e.g., "What's on my schedule tomorrow?") [24]. This likely requires detecting the owner's caller ID or an authentication PIN. Implement recognition of the owner and routing to an "owner query" dialog flow. Ensure the assistant can fetch the requested information (schedule, job details) and read it out via TTS [25]. Test this flow with the owner's phone number and verify the privacy (only the owner gets their info).

- **Twilio Call Handling & Webhooks:** Ensure the telephony integration is solid and can handle real call traffic.

- *Sub-task:* Verify the **Twilio Voice webhook** (`/v1/telephony/voice` or similar, as configured) returns correct TwiML to initiate our IVR flow [16]. It should use `<Gather input="speech">` (or DTMF as fallback) with our TTS prompt, then receive the speech events. Test that our webhook endpoints handle Twilio's expected request/response format (Twilio may send partial transcription events, etc., depending on settings).
- *Sub-task:* Implement call **time-out and error handling**: if a call is disconnected or the user is silent for too long, make sure to end the session gracefully. Twilio can time out a `<Gather>` after some seconds of silence — ensure our system recognizes that and perhaps retries once or ends the call politely ("Sorry, I have to hang up now. Please call back if you need anything else.").
- *Sub-task:* **Call bridging (if required):** For emergencies, consider using Twilio's ability to bridge the call to the owner. The design suggests notifying via SMS, but a possible enhancement is to say "I'm connecting you to the on-call technician now" and then <Dial> the owner's number. If we choose to

implement this, ensure Twilio <Dial> is used properly and the call hand-off is smooth (this could be an advanced feature beyond MVP, so mark it accordingly).

- *Sub-task:* Validate **DTMF fallback**: In case the STT fails or if a caller prefers using the keypad (e.g., press 1 for confirmation), our flow currently doesn't specify it, but ensure we have a plan if needed (perhaps for future accessibility, though not in current scope).
- *Sub-task:* Update Twilio configuration guides in documentation: ensure `Twilio/Voice` webhook URL and behavior are clearly described in README or RUNBOOK (so that the deployment person knows how to set Twilio phone number webhooks to point to our service) [16] .

Scheduling & Calendar Integration

- **Google Calendar Integration:** Complete and harden the integration with Google Calendar for scheduling appointments [26] .
- *Sub-task:* Finalize the **OAuth flow** for Google Calendar API. Currently, the design likely uses a service account or assumes a token; instead, implement the proper OAuth 2.0 for each tenant owner during onboarding (e.g., `/auth/google` routes) [27] . Store and refresh tokens securely (in the database or GCP Secret Manager) so that the backend can access the calendar on behalf of the user.
- *Sub-task:* Implement full **read/write operations**: checking availability, creating events, updating or deleting events on cancel. Ensure that when an appointment is scheduled via the AI, an event is created on the owner's Google Calendar with all relevant details (customer name, address, phone, service type, notes about the issue, emergency flag if any). Likewise, if the assistant cancels or reschedules, it should update the calendar event (or create a new one) accordingly [28] [29] .
- *Sub-task:* **Service duration & business hours:** incorporate the business's working hours and default service durations when finding time slots [30] [31] . For example, avoid proposing times outside of 9-5 (configurable), and if a job type like "water heater install" is usually 4 hours, the slot finding logic should block out that duration. This may involve storing a configuration for each service type's duration and the business's hours in the Business configuration entity [32] .
- *Sub-task:* Handle **calendar conflicts and edge cases**: if two callers attempt to book overlapping times, the system should detect that the slot was taken by the first and propose the next best slot to the second. Also handle Google Calendar failures (API not responding or returning an error) by having a fallback message ("I'm unable to schedule right now, but I will have the office follow up with you to confirm a time").

- *Sub-task:* Test the calendar integration thoroughly with a real Google Calendar in a test GCP project. Simulate multiple bookings, cancellations, and ensure events appear correctly. Also test that the event details are readable and make sense to the business owner (e.g., all info captured by the assistant is in the description or fields of the calendar event).

- **Appointment Management & Follow-ups:** Improve how appointments are tracked and updated.

- *Sub-task:* Implement **appointment rescheduling flows** in the backend. When a customer says they need to reschedule (or sends "RESCHEDULE" via SMS), mark the appointment as `PENDING_RESCHEDULE` in our system [33] . Create an endpoint or background task for the owner/admin to see these pending reschedules (the owner dashboard might already have a card for "Appointments marked for reschedule" – ensure it's populated) [34] . The assistant could also proactively offer the next available slot, but likely this is left for human follow-up (per design).

- *Sub-task:* Ensure **appointment cancellation** is fully handled. If a cancellation is requested (via voice or SMS "NO"), the system should mark the appointment as canceled in our DB and attempt to remove or mark as canceled on Google Calendar [21]. Verify that SMS "NO" triggers this and that voice cancellation (if implemented) does the same. The owner should be notified of cancellations (perhaps via an SMS alert or at least visible in the dashboard).
- *Sub-task:* Maintain a **single source of truth** for appointments. Our database should store each appointment and its status (confirmed, pending reschedule, canceled, completed, etc.), syncing changes to Google Calendar but not relying solely on Google Calendar for state. Implement any missing fields in the Appointment model for status or cancellation reason if needed.
- *Sub-task:* Implement an **appointment reminder system** (could be considered part of Notifications). Possibly have a daily cron job or scheduled function to send reminders (SMS or email) to customers X hours before an appointment. This ties into calendar integration by pulling upcoming appointments. If this is in scope, create endpoints or background tasks to handle sending and allow customers to confirm/cancel via reply (which is covered in Notifications section).

CRM & Data Management

- **Repeat Customer Recognition:** Implement logic to identify returning callers and pre-fill their info [35].
- *Sub-task:* On an incoming call, check the caller's phone number against the **Customer** database. If a match is found, fetch the customer's name and any saved address. The conversation flow should then skip asking for name/address, instead confirming we have their details ("Hi [Name], welcome back! I have your address as [XYZ], is that still correct?") [36] [37].
- *Sub-task:* If the caller mentions they are a returning customer ("You fixed my water heater last year"), ensure the system can handle it even if phone number didn't auto-match (maybe they called from a different number). Provide a way for the operator (owner) later to merge or link such records. This might be a stretch goal – at minimum log that info for manual reconciliation.
- *Sub-task:* After booking a repeat customer's new job, **link the appointment** to the existing customer record and record any references to previous jobs. E.g., if they mention a prior job, capture that context in the conversation notes ("customer referenced previous tankless heater install in 2022"). This provides continuity in the CRM.

- *Sub-task:* Test with sample data: create a customer in the DB, simulate a call from their number, and verify the system recognizes them and that the flow follows the returning-customer path as designed [38].

- **Conversation Logging & Summaries:** Store detailed records of conversations for QA and history [39].

- *Sub-task:* Decide on a **storage format** for call transcripts. Options: store the full text transcript of the call, or a sequence of messages (customer said X, assistant replied Y). Implement this by capturing the STT results and the prompts at each dialog turn. Save these in a **Conversation** record in the database, linked to the customer and appointment (if one was scheduled) [40].
- *Sub-task:* If storing full transcripts is too verbose or raises privacy concerns, consider storing a **summary** of the conversation. For example, after a call completes, generate a short summary (possibly template-based or using an LLM in the future) such as "Caller John Doe reported a leaking faucet. Appointment scheduled for 2025-12-10 at 3pm." Store that summary along with key

metadata (was it an emergency, did they confirm via SMS, etc.). This summary can be shown in the dashboard's conversation log for quick review [39].

- *Sub-task:* Implement **privacy controls** on conversation logs: if sensitive info is captured (like addresses, phone numbers), ensure it's stored securely. Follow any data retention policy – e.g., maybe delete or anonymize transcripts after N months unless needed for legal reasons (consult `PRIVACY_POLICY.md`). This might involve a scheduled job to purge old conversation records or an admin tool to do so.
- *Sub-task:* Enable **search or filtering** of conversation history in the owner dashboard. For example, allow the owner to search conversations by customer name or keyword to quickly find past calls. This could be a stretch goal; at minimum ensure conversation records are accessible via the API so the dashboard can list recent conversations for QA/training purposes [41].
- *Sub-task:* If call recording is enabled via Twilio (not explicitly mentioned, but if desired), make sure recordings are handled properly (stored in a bucket, or links saved) and that usage complies with privacy guidelines (e.g., obtain consent or at least mention in privacy policy). This is optional, as transcripts might suffice.

- **Persistent Data Store & Model Validation:** Finalize the database integration and ensure data integrity.

- *Sub-task:* Move beyond the dev SQLite/in-memory store to a cloud database. Likely use **Firestore** (NoSQL) or **Cloud SQL** (PostgreSQL) as planned [3]. Implement the repository layer or SQLAlchemy models such that switching the env to production DB is seamless. This may include writing migration scripts or ORMs for a SQL database, or testing Firestore queries for performance.
- *Sub-task:* Implement **input validation and sanitization** on all data going into the DB. For example, ensure phone numbers are stored in a consistent format, addresses are not overly long or malicious (if coming from user input, though mostly voice), and any text fields are checked for strange content (to avoid issues like injection or broken records). Use pydantic models or FastAPI schema validation for request bodies where possible.
- *Sub-task:* Add **database indexes/queries** for common access patterns [3]. E.g., ensure quick lookup by phone number (for repeat customer), by appointment date (for daily schedules), and by business/tenant. If using Firestore, this means defining composite indexes if needed (like on business_id + phone). If using SQL, ensure proper indices on foreign keys and date fields.
- *Sub-task:* Test **multi-tenant data isolation** at the data layer: all queries should be scoped by `business_id` where applicable [42]. Write tests to verify that one tenant's API key cannot access another tenant's data (attempt cross-IDs in requests and expect denial or empty results). This is crucial for SaaS correctness.
- *Sub-task:* Implement any **missing model fields or relationships**: The conceptual data model includes Business, Customer, Appointment, Conversation, User, Configuration [43]. Verify that all these entities are represented in code and that relationships (like linking Conversation to Appointment or Customer) are implemented. For example, if we have a ConversationMessage or similar, ensure it's captured; if not, adjust the design to fit transcripts in Conversation. Also, ensure the **Configuration** (business hours, emergency keywords, service types) is persisted and used in logic.

- **Integration Stubs (QuickBooks, etc.):** Address or clarify the stubbed integration points in the system.

- *Sub-task:* **QuickBooks Integration:** The code has stub routes for QuickBooks OAuth and sync [27] [44]. Decide whether to implement a basic integration (e.g., connect to QuickBooks sandbox and pull a list of customers or invoices). At minimum, ensure the OAuth handshake completes and perhaps store a token, even if we do not deeply integrate data. If full integration is out of scope, clearly document that it's a future feature and make sure the stubs fail gracefully (e.g., return "not implemented" message rather than just being placeholders).
- *Sub-task:* **Email Integration:** (Not heavily mentioned in docs, but possibly needed for sending confirmation emails or for login). If any email service integration is planned (like sending a summary to the owner's email or customer's email), implement that using a service like SendGrid or a simple SMTP, following the privacy guidelines. If not planned, ensure any reference to email (like collecting customer email) is either used for something (like sending a calendar invite maybe) or clearly optional.
- *Sub-task:* **Other provider stubs:** The onboarding mentions LinkedIn, Gmail/Workspace integrations [45]. These are likely beyond MVP. Create issues to track these as future enhancements (so they are not forgotten), but mark them as low priority. Ensure that any currently exposed endpoints for them are either disabled or return a friendly message so as not to confuse users/testers.

Web Dashboard & User Experience

- **Owner Dashboard Completion:** Finish all features on the business owner's web dashboard (the static `dashboard/index.html`) [46].
- *Sub-task:* Implement the **Schedule view** and **Job list** in the dashboard with live data. The cards for "Tomorrow's Schedule" and "Today's Jobs" should call the corresponding API endpoints (`/v1/owner/schedule/tomorrow`, `/v1/owner/summary/today`, etc.) and display the results [34]. Verify that these endpoints are working and adjust the backend if needed (for example, ensure they filter by the correct tenant and date, and format the response with all needed info).
- *Sub-task:* Build out the **Conversation log viewer** for QA/training. This likely involves calling `/v1/crm/conversations` to get recent conversation records [47]. On the UI, display a list of conversations with maybe a snippet or summary, and allow the owner to click to see full details (transcript or summary, plus whether it was an emergency or resulted in an appointment). If transcripts are long, consider a scrollable area or modal dialog for details.
- *Sub-task:* Implement the **Analytics cards**: e.g., jobs per week, common services, maybe revenue estimates. The backend might not yet have dedicated endpoints for some analytics (except those mentioned like `/v1/owner/service-mix?days=N` which gives service type mix [48]). Use what's available: service mix for common services, Twilio metrics for call/SMS counts, etc. If needed, create new endpoints for any missing metric (like revenue or conversion rate) or adjust the UI to what data we have.
- *Sub-task:* **Business configuration UI:** If not already present, add a section or modal for the owner to view/edit business settings (hours of operation, list of offered services and their default durations, emergency keyword list, notification preferences). While these could also be edited in a config file, a simple UI helps. Back the changes with new API routes (e.g., `PATCH /v1/owner/business` to update hours or services). If implementing, ensure proper validation (no crazy hours, etc.) and security (only the owner can do this).

- *Sub-task:* Review **PWA functionality** of the dashboard. The dashboard is a simple static app; ensure it has a manifest and can be added to home screen if intended. It might not need offline capability like the chat, but ensure it at least loads fast and works on mobile browsers (in case the owner opens it on phone). Test on a mobile device for responsiveness of the layout.

- **Admin Dashboard & Multi-Tenant Management:** Complete the admin interface for platform-wide oversight [49].

- *Sub-task:* Ensure the **tenant list and usage metrics** display correctly. The admin dashboard's "Tenants" card should call `/v1/admin/businesses` to list all businesses [50], and possibly `/v1/admin/businesses/usage` for per-tenant stats [50]. Verify that these endpoints aggregate data properly (appointments count, SMS volume, etc.) and fix any issues in their implementation.
- *Sub-task:* Implement **tenant status controls**: the admin should be able to suspend or activate a tenant. Wire up the UI to the `PATCH /v1/admin/businesses/{id}` endpoint which updates a business's status or notification settings [42] [51]. Test that when a business is suspended, their calls/SMS are indeed blocked by the backend (the backend likely checks `status` on each request).
- *Sub-task:* Add actions for **API key rotation** and **Widget token rotation** on the admin UI (buttons that call the POST endpoints to rotate keys) [52]. After rotation, ensure the new keys are displayed to the admin so they can communicate them securely to the tenant.
- *Sub-task:* Display **Twilio/Webhook health**: use the data from `/v1/admin/twilio/health` [50]. This should show if each tenant's Twilio number is configured correctly, and counts of recent voice/SMS requests and errors per tenant. Provide a clear indicator for any tenant with high error rates or missing webhook calls (to prompt support action).

- *Sub-task:* Test the admin dashboard with multiple businesses: create a couple of dummy tenant entries and simulate some traffic for each (appointments, etc.), then ensure the admin dashboard cleanly separates these and sums up global metrics. This will validate multi-tenant support end-to-end from an admin perspective.

- **Web Chat Widget Enhancement:** Finalize the customer-facing web chat channel (embedded widget) [53].

- *Sub-task:* Verify that the **web widget** (`widget/chat.html` and `embed.js`) works for text conversations. It should use the `X-Widget-Token` and communicate with the backend's chat endpoints. Test a flow: open the widget, start a conversation, and see that it creates a conversation tied to a business and returns answers (the backend has `/v1/widget/start` and `/v1/widget/{conversation}/message` presumably). Ensure the conversation also shows up in the CRM conversation log so that owners can see chats as well as calls [54].
- *Sub-task:* Ensure **offline support** via the service worker is working. The service worker `chat/sw.js` is designed to queue messages when offline [55] [56]. Test this by going offline and sending a message, then coming online to see it flush. Fix any bugs in queueing (e.g., the UI should indicate the message is queued).
- *Sub-task:* Add any needed **UI polish** to the widget: for example, ensure it can be easily embedded on a website (the `embed.js` should allow insertion via a script tag). Possibly provide a minimal customization like allowing the business name or logo in the chat header. Ensure the widget does not expose the `api_key` – it should use only the widget token for security [57].

- *Sub-task:* Plan for **voice support in web widget** (future consideration). The project mentioned future channels like a voice website widget [53]. While not for MVP, design an issue for adding microphone access in the widget so a user could have a voice chat through the browser. This would require WebRTC or audio streaming to the backend STT, which is complex, but noting it as a future feature is important for completeness.

- **User Experience Review:** Conduct an end-to-end UX review for both customers (callers/chatters) and the business owner.

- *Sub-task:* For callers: make test calls covering various scenarios (new customer standard booking, emergency call, repeat customer, after-hours call if applicable) and note any confusing prompts or long pauses. Improve prompt wording or logic as needed (e.g., if the assistant takes too long between STT and TTS, consider adding a filler like "One moment while I check that for you…").
- *Sub-task:* For the owner: use the dashboard and ensure it provides the information an owner needs without confusion. Are the analytics meaningful? Are the conversation logs easy to navigate? Gather any feedback (if available from a pilot user) and create tasks to address UX pain points (like adding a loading spinner, or a clarification in the UI).
- *Sub-task:* Mobile experience: test the owner dashboard and chat widget on mobile devices. The owner dashboard might need some responsive tweaks; ensure the tables or cards stack nicely on a narrow screen. The chat widget should be mobile-friendly (since it's a PWA too). Create follow-up tasks for any mobile UX issues found (like text overlapping, buttons too small, etc.).

Notifications & Messaging

- **Owner SMS Alerts:** Ensure the business owner is immediately notified of important events via SMS [58] .
- *Sub-task:* On **new lead** (new appointment booked by the AI), send an SMS to the owner's phone number with a brief summary (e.g., "New plumbing job booked for [Customer Name] on [Date] at [Time]. Check your dashboard for details.") [59] . This feature should be configurable per tenant (some may not want every booking via SMS). Use the Twilio API (or another SMS service) to send these; ensure the owner's number is stored in the Business config.
- *Sub-task:* On **emergency detected**, send an immediate SMS alert (and/or phone call if desired) to the owner stating "Emergency call from [Name]: [Issue]. Appointment set for ASAP." [20] [60] . This is critical so the owner can react quickly. Test that the emergency flag triggers this alert every time. Possibly include a high-priority marker (if using Twilio, maybe use a different messaging service or simply clear wording).
- *Sub-task:* For **missed/failed calls** (if the AI could not schedule or something went wrong), alert the owner so they can follow up manually. For example, if STT failed and the call ended without booking, send an SMS: "Missed lead: call from [number] at [time] needs follow-up (AI couldn't process).". This ensures no call truly slips through unnoticed [61] .

- *Sub-task:* Implement configuration for **alert preferences** – maybe the owner can choose which events trigger SMS (perhaps in the business config UI). Ensure that turning off an alert actually stops the messages. Default should be important alerts on.

- **Customer SMS Confirmations & Reminders:** Build out the SMS interaction with customers for appointment confirmations [62] .

- *Sub-task:* After booking an appointment, send a **confirmation SMS** to the customer (if they haven't opted out). Include the appointment time, perhaps a summary of the service, and a line like "Reply YES to confirm, NO to cancel, RESCHEDULE to change the time." [21] . Ensure this SMS is only sent if we have the customer's mobile number and they haven't opted out of texts [60] .
- *Sub-task:* Implement **SMS reply handling**: the Twilio SMS webhook (`/twilio/sms`) should already route incoming texts to our system [63] . Add logic to parse replies for confirmation keywords:
    - If the customer replies "YES" or "Y" [64] , mark their appointment as confirmed in the DB (and possibly send a thank-you response).
    - If they reply "NO" or "CANCEL", mark the appointment canceled (and notify owner, remove from calendar) [65] .
    - If they reply "RESCHEDULE", mark it as pending reschedule (and perhaps the system can reply with "We will contact you to reschedule ASAP" to acknowledge).
    - For anything else, or if they have multiple appointments, consider adding a clarification response ("Which appointment are you referring to?") – though as MVP we assume one upcoming appointment at a time per customer.
- *Sub-task:* Set up **reminder messages** ahead of appointments (e.g., 1 day before). These can be implemented via a scheduled function or simply triggered when the appointment is made (schedule a delay). The reminder should include a brief "Reminder: you have an appointment with [Business] tomorrow at [Time]. Reply CANCEL to cancel or RESCHEDULE to change." Only send if the appointment is still confirmed and not too far out.
- *Sub-task:* Honor **opt-out keywords** for customer messages [60] . The system should detect if a customer ever replies with STOP/UNSUBSCRIBE or similar, and then flag that number as opted out. Implement this flag in the Customer model (if not already) and check it before sending any future SMS to that number [60] . Also handle "START" or "UNSTOP" to resume messages if they opt back in. Write tests for these to ensure compliance.

- *Sub-task:* Implement basic **SMS conversation logging**: when customers text things other than our keywords (e.g., they ask a question via SMS), decide how to handle it. Possibly treat it as part of a conversation and have the assistant respond via SMS (if within capability), or send a polite default reply ("Thank you, we'll get back to you soon"). Log these SMS exchanges in the conversation history as well, so the owner can see if a customer tried to communicate via text outside the automation.

- **Retention and Follow-up Campaigns:** Utilize SMS for retention (as hinted by a retention endpoint).

- *Sub-task:* Implement the `/v1/retention/send-retention` endpoint to send a retention campaign message [66] . For example, identify customers who haven't had jobs in, say, 6 months and send a courtesy follow-up ("It's been a while since we heard from you – if you have any plumbing needs or want a checkup, let us know!"). Make this configurable (which customers to target, content of message) and **throttle** such campaigns to avoid spamming.
- *Sub-task:* Ensure these retention messages also respect opt-out flags and include proper opt-out instructions (to be legally compliant). Since this is marketing-ish, including "Reply STOP to unsubscribe" is important.

- *Sub-task:* Provide a simple way for the owner to initiate a retention send (perhaps via the admin dashboard or a script). Since this might be a low priority feature (depending on project scope), at least outline it in docs or backlog for future implementation if not doing it now.

- **Twilio Configuration & Testing:** Solidify the Twilio integration beyond just code.

- *Sub-task:* Double-check the Twilio **phone number setup**: voice webhook pointing to `/twilio/voice`, messaging webhook to `/twilio/sms` [16], with appropriate parameters (`business_id` query param if using one number for multiple tenants). Document this in the deployment or runbook so whoever sets up Twilio knows how to configure it [16].
- *Sub-task:* Test Twilio error scenarios: e.g., if our webhook is down, Twilio will retry – ensure our endpoints handle duplicate webhook calls **idempotently** (for example, Twilio might resend a voice start request, so our session creation should recognize if a session with that CallSid exists already and not create a duplicate). Build idempotency keys or checks where appropriate.
- *Sub-task:* Monitor Twilio usage: use Twilio's console or API to confirm we aren't hitting any rate limits or account limits during testing. The admin dashboard Twilio health metrics [50] should reflect the number of calls and any errors – use this to catch issues early (like misconfigured TwiML or message delivery failures).
- *Sub-task:* If the project will scale, consider implementing **multiple Twilio numbers** (one per tenant or region) and handling them. For now, if using one number with a business_id parameter, test that scenario. If wanting to support multiple numbers, add a mapping from Twilio phone -> business in the DB and adjust the webhook to figure out business_id from either the query or the from number. This can be a future enhancement, but note it if multi-tenant scaling is planned.

Multi-Tenancy & SaaS Readiness

- **Tenant/User Account Model:** Complete the implementation of multi-tenant support by introducing user accounts and roles [27].
- *Sub-task:* Implement the **user registration and login** endpoints (`/v1/auth/register`, `/v1/auth/me`, etc.) that are currently stubbed [27]. This includes creating a **User** model (if not already) with fields like email, password hash (if using local auth), and linking users to one or more Business (tenant) with a role (Owner, Staff, Admin). Even if the pilot uses a single tenant, having the structure in place will support expansion.
- *Sub-task:* Implement **authentication** for the owner/admin dashboards. For instance, the owner dashboard currently uses `X-API-Key` and `X-Owner-Token` which might be provided after login or from a config [67]. We should integrate a proper auth flow: owners log in (using email/password or Google OAuth if desired), get a token, and the dashboard uses that. This could be a JWT or a session token. If out of scope for MVP, ensure at least a basic API key mechanism is securely in place (perhaps one per user rather than one per business, to allow revocation per user).
- *Sub-task:* **Active business selection:** If a user can belong to multiple businesses (e.g., a virtual assistant managing two companies or an admin overseeing all), implement the `/v1/auth/active-business` so the user can switch context [27]. This might involve storing a current business_id in their session or token. Ensure that all subsequent API calls respect that active business for data isolation.

- *Sub-task:* Test the system with at least two tenant businesses in parallel. Sign up two dummy businesses (via the self-signup flow or manual DB insert), create some data for each (appointments, calls, etc.), and verify that using one business's API key or user login never exposes data from the other business. This includes checking that websockets or background tasks, if any, don't mix up tenants. The metrics endpoint should have separate entries for each [68]. Any cross-tenant leakage would be a serious bug to fix.

- **Self-Service Signup & Onboarding:** Finalize the public signup process for new businesses to onboard themselves [69].

- *Sub-task:* Complete the **public signup API** (`/v1/public/signup`) and the static `dashboard/signup.html` flow [69]. This involves capturing business details (company name, owner contact, maybe payment info if required for subscription) and creating a new Business record and an Owner user. Ensure proper email verification or admin approval steps if needed (to avoid spam signups, perhaps). At minimum, make the process create an inactive tenant that can be activated by an admin or by email confirmation.
- *Sub-task:* Implement the **onboarding steps** in `dashboard/onboarding.html` [69]. This likely walks the new user through connecting their Google Calendar, Twilio, and any other integrations (OpenAI for voice AI if needed). For each integration:
    - Google Calendar: direct the user to an OAuth consent screen, get the token (this ties into the earlier Calendar integration tasks).
    - Twilio: since Twilio can't easily be OAuth'ed, we might ask the user for their Twilio credentials or a number to use – this is tricky. Alternatively, the platform could provide a number for them if we manage Twilio centrally. Define how a new tenant gets a phone number: possibly manual for now (admin assigns one). Document this step (maybe in onboarding, just instruct "We will contact you to set up a phone number" if not automated).
    - Other services: QuickBooks (likely optional – provide the OAuth link stub and mark as coming soon, or if implemented, allow them to connect and import sample data).
    - OpenAI API: if the AI assistant uses an external model (like GPT-4 for some answers), let them plug in an API key. If using our own, skip.
    - Each step should update the Business record (store tokens, IDs, etc.). Provide feedback on the onboarding page (e.g., "Google Calendar connected" after success). Use the stubbed endpoints under `/auth/{provider}/*` as needed, turning them into actual integration flows.
- *Sub-task:* **Onboarding error handling:** ensure that if any step fails (e.g., invalid credentials or user skips a step), the system handles it gracefully. They should be able to retry integration steps. Also, if not all integrations are completed, the system should still be functional (maybe with limited capabilities). For example, if they didn't provide a Twilio number, we might not handle calls for them yet – warn them of that limitation.

- *Sub-task:* After onboarding, make sure the new tenant's data is initialized properly: e.g., generate an API key and widget token for them (and display it), create default business hours or service configurations, and perhaps a welcome conversation or test entry so they can try things out. Essentially, verify the tenant is ready to go.

- **Billing & Subscription Management:** Although low priority in the backlog [70], outline tasks for handling SaaS billing if needed.

- *Sub-task:* Design a simple **subscription model** (if this will be offered beyond the pilot customer). For instance, have a "Free trial" vs "Paid plan". Implement endpoints like `/v1/billing/plans` to list plans and `/v1/billing/create-checkout-session` (likely for Stripe Checkout as hinted by naming) to initiate payment [71]. Even if not fully implemented now, structure the code to accommodate checking a tenant's plan and enforcing any limits (like number of calls per month).

- *Sub-task:* Integrate with a payment processor (Stripe) for recurring subscriptions. Use webhooks (`/v1/billing/webhook`) to listen for payment events and update tenant status (active if paid, suspended if payment failed) [71]. If this is too much for now, leave stubs but document the manual step (e.g., manually mark business as active in DB).
- *Sub-task:* Add **usage tracking for quotas**: for example, if the plan has a limit (say 100 calls/month), ensure we count calls (we have metrics counting requests per business [68]). Build a mechanism to enforce the limit (warn or suspend service when exceeded). This can be a future feature, but planning it now helps.
- *Sub-task:* Expose basic **billing info** on the dashboard: maybe an Admin-only view of each tenant's plan and renewal date, or an Owner view for their own subscription status. This is optional, but if billing is live, it's necessary.

- *Sub-task:* Clearly mark all billing features as **beta/future** if not done, to set expectations. We don't want to promise functionality that doesn't exist yet, so update any documentation (Terms of Service, etc.) accordingly if needed.

- **Scalability Considerations for Multi-Tenant:** Ensure the platform can scale to multiple businesses.

- *Sub-task:* Confirm that the **metrics and monitoring** are segmented by tenant and also viewable in aggregate. For example, the global metrics `/metrics` has per-business breakdowns [68] – test that adding more tenants doesn't degrade performance (the metric structure might grow).
- *Sub-task:* Evaluate whether any **per-tenant resource limits** are needed. For instance, if one tenant suddenly has a spike in calls, do we have isolation (so they don't starve others)? In Kubernetes, this could be tackled by scaling out pods (if heavy usage by one tenant, it simply uses more pods). If any resource is shared (like a single global ASR service instance), ensure it can handle concurrent requests or plan to scale it.
- *Sub-task:* Plan for **data backup and migration** for each tenant's data. If using Firestore/CloudSQL, set up backups. If a tenant leaves and wants their data, have a procedure (even if manual) to export their customers/appointments (perhaps via the existing CSV export endpoints [72]).
- *Sub-task:* Internationalization & Localization (future): If the platform will onboard businesses in different locales, consider language support for TTS/STT and units. Not needed for MVP, but note if any architectural decision now could complicate that later (for instance, hard-coding English prompts – maybe keep prompts in a config for easy swapping if needed).

Security, Privacy & Compliance

- **Security Hardening:** Implement strong security measures as outlined by the engineering standard [73].
- *Sub-task:* Enforce **TLS everywhere**: ensure that the deployed service is only accessible via HTTPS. In GCP, use HTTPS load balancers or Cloud Run's HTTPS endpoints. Also ensure internal service communication (if any) is secure (though likely all in one service).
- *Sub-task:* Apply **principle of least privilege**: limit the permissions of the GCP service account running the backend (only allow it access to necessary services like Firestore, not everything). Similarly, if the app uses third-party APIs (Google Calendar, Twilio), restrict those credentials to limited scope (Calendar API tokens only grant calendar access, Twilio key only for certain actions, etc.) [73].

- *Sub-task:* Implement robust **authentication & authorization** in the API. Any endpoints that require admin or owner privileges should check the provided token/API key and also verify the role (e.g., an `X-Admin-API-Key` should be required for admin endpoints and must match the configured admin key) [74]. Ensure no sensitive endpoint is left unprotected (double-check that all `/v1/admin/*` routes indeed check for the admin key, etc.).
- *Sub-task:* **Audit logging:** For sensitive actions like scheduling an appointment, canceling, or modifying business settings, create an audit log entry [73]. This could simply be a log line with level INFO or a dedicated database table. Include who (which user or system) did what and when. This is invaluable for troubleshooting and security reviews.

- *Sub-task:* Protect against **common web vulnerabilities**: run tests or scans for SQL injection (if using SQL, ensure queries are parameterized or use ORM), XSS (the dashboards are static and consume data – make sure any data rendered in HTML is sanitized, especially conversation transcripts or customer inputs). Also consider adding security headers for the dashboard (content security policy, etc., since it’s static files). For the API, ensure proper CORS settings so only allowed origins (the dashboards domain) can call it in browsers.

- **Privacy Compliance:** Ensure the system respects user data privacy and applicable laws.

- *Sub-task:* Clearly post the **Privacy Policy and Terms of Service** in the application or website (perhaps link from the dashboard or signup page) [75]. They are provided in the repo; make sure they cover how call recordings/transcripts are used and retained. If needed, update these documents as features evolve (e.g., if we start recording calls, mention that explicitly).
- *Sub-task:* Implement **data retention and deletion policies**: for instance, if the privacy policy says call transcripts are kept for 1 year, enforce that by purging or archiving data older than that. Possibly provide an endpoint for a tenant to delete a conversation or a customer if requested.
- *Sub-task:* **Consent for call recording or AI handling:** Depending on jurisdiction, if calls are recorded or even transcribed, the caller might need to be informed. The greeting might need to include a phrase like “This call may be recorded or monitored.” If required, ensure the voice script includes this. Also, if using an AI voice, some places require notifying the user they are speaking to an AI. Consider adding a brief hint in the greeting (“You’re speaking with an automated assistant.”) if legally needed or for transparency.
- *Sub-task:* **Opt-out compliance:** We've handled SMS opt-out in features; ensure that if a customer opts out, we not only stop messaging them, but also that we don't inadvertently include their data in any future retention campaigns, etc. Basically, respect all “do not contact” flags thoroughly.
- *Sub-task:* Safeguard **personal data**: make sure that sensitive personal info (customer contacts, conversation content) is not exposed to unauthorized parties. For example, the admin should not be able to see customer details of a tenant unless needed (maybe admin can, since they manage system, but that’s a business decision – at least it should not be displayed wantonly). Also ensure any logs or monitoring tools do not accidentally log PII (e.g., don’t log full conversation text at ERROR level in a shared log). Use pseudonymization in logs if needed (like log customer ID instead of name).

- *Sub-task:* Plan for **incident response** in case of a data breach. The RUNBOOK should have a section for security incidents. If not, create an issue to write down steps: how to identify affected data, how to communicate to users, etc., in line with best practices and possibly legal requirements.

- **Compliance and Regulatory:** Address any industry-specific or regional regulations.

- *Sub-task:* If servicing customers in certain regions, consider **GDPR** (right to access, right to delete personal data). We should be prepared to export all data for a customer or delete it on request. Implement tools or at least document how to do this manually (e.g., a script to gather all conversations and appointments for a given customer phone or name).
- *Sub-task:* If voice calls are recorded or even just transcribed, comply with **telecom regulations**. For example, in the US, automated calls (robocalls) have regulations – but here the customer is calling us, so it's interactive, which is usually fine. Just ensure we don't do outbound robocalls without opt-in. Also, for SMS, ensure compliance with 10DLC (application-to-person messaging rules) if sending volume, although for a small business pilot it might not be an issue.
- *Sub-task:* **Accessibility**: Though not exactly security, ensure the system is usable by people with disabilities. E.g., the web dashboard should be screen-reader friendly (basic HTML likely is), and voice interactions should account for those who might have difficulty (for instance, maybe provide a way to speak to a human, though single-person business might not have one immediately). Not a strict requirement, but a consideration to note.
- *Sub-task:* Periodically conduct a **security audit** or **penetration test**. This could be an external task (engaging a security firm) – for now, create an issue as a reminder for after MVP launch. Any findings from such tests would generate further sub-issues to fix, but having it in the plan shows completeness.

Testing & Quality Assurance

- **Unit Testing Coverage:** Develop thorough unit tests for all critical components [76] [77].
- *Sub-task:* Write tests for the **conversation flow** logic (e.g., functions that decide the next prompt based on state). Simulate various paths: standard booking, emergency detected, user asks to reschedule mid-call, etc., and assert that the state transitions and outputs are as expected. Use dummy STT input strings to drive the state machine through each stage.
- *Sub-task:* Test the **emergency keyword detection** function explicitly. Feed it sentences with emergency phrases (“my basement is flooding”) and without, and ensure it correctly flags emergencies [20]. Also test edge cases (e.g., “I have no water pressure” vs “I have no time today” to avoid false positives on the word “no”).
- *Sub-task:* Test **scheduling logic** in isolation. If there’s a function for finding an available slot given a desired date range, feed it some sample busy schedules and verify it returns the correct next slot. If using Google Calendar API in tests is hard, abstract the calendar as an interface and stub it with sample data for testing.
- *Sub-task:* Test the **CRM repository** functions. For example, ensure that creating a customer and then querying by phone returns the customer (for repeat customer logic), or that appointments can be retrieved by date or status correctly. If using an in-memory or SQLite for tests, verify that the SQL logic or Firestore queries behave as intended.
- *Sub-task:* Test **SMS parsing logic** for confirmations. Simulate incoming SMS webhook payloads for “YES”, “NO”, “RESCHEDULE” messages and assert that the appropriate changes happen (appointment confirmed/canceled flags, etc.). These can be unit tests if the SMS handling is in a function that can be called with text input.

- *Sub-task:* Aim for high coverage on “safety-critical” paths (emergency handling, scheduling, Twilio webhooks) [78]. The standard calls for strong tests especially around those areas, so ensure near 100% coverage there, if possible.

- **Integration & End-to-End Testing:** Validate how components work together in a staging-like environment [76].

- *Sub-task:* Set up an **integration test environment** using the in-memory or SQLite mode with stubbed providers (as per `env.dev.inmemory` or `env.dev.twilio` profiles) [79] [80]. Write integration tests that spin up the FastAPI app (perhaps with `TestClient`) and simulate a full call: hitting the telephony webhook with a starting call event, then sending a few speech inputs (as if from Twilio), and verify an appointment is created in the DB and the final TwiML ends the call. This will test the API routing, state management, and DB together.
- *Sub-task:* Test the **Twilio SMS -> appointment flow** in integration: e.g., create an appointment, then simulate an incoming "YES" SMS via the `/twilio/sms` endpoint, and check that the appointment status flips to confirmed. Similarly test "CANCEL" SMS leading to appointment removal and a notification to owner (perhaps by checking a log or a flag since sending actual SMS in test isn't feasible).
- *Sub-task:* If possible, test a basic **Google Calendar integration** on a staging Google Calendar. This might be more of a manual or semi-automated test due to OAuth. But we could use a service account or a test token to call our calendar functions in an integration test to ensure events are created and deleted as expected.
- *Sub-task:* Perform an **end-to-end manual test** with the real Twilio sandbox and a Google Calendar: call the Twilio number, talk to the AI, go through a booking, and verify on the actual Google Calendar the event appears. Then reply to the confirmation SMS and see the system's behavior. This manual QA step is important to catch any issues that automated tests might miss (audio quality, real-world latency, etc.). Document the results and log any issues encountered to fix them.

- *Sub-task:* Test failure modes in integration: e.g., have the calendar API call fail (maybe by providing an invalid token) and ensure the system handles it (returns an apologetic message rather than crashing). Or have the DB fail (simulate by pointing to a wrong DB connection mid-call) to see if the system fails gracefully. While these aren't normal, knowing the behavior helps ensure resilience.

- **Load and Performance Testing:** Validate the system under load and optimize as needed [81] [82].

- *Sub-task:* Use the provided `backend/load_test_voice.py` tool to simulate concurrent call sessions [83]. For example, run 50 sessions with 10 concurrency [84], and measure the latency (p50, p95, p99) for the end-to-end voice session handling [85]. Do this in a controlled environment (perhaps on a staging deployment or locally) and record the performance.
- *Sub-task:* Analyze load test results: if p95 latency is above acceptable thresholds (for voice, ideally responses should be under a couple seconds max to avoid caller frustration), identify bottlenecks. It could be STT processing time or our own logic. If STT is too slow, consider solutions like streaming partial responses or switching to a faster model. If our code is slow (e.g., doing blocking I/O), consider async improvements. Optimize database calls if needed (e.g., caching business config in memory instead of fetching each time).
- *Sub-task:* Test **scalability**: ramp up the number of concurrent sessions until failure to see where the breaking point is (could be CPU bound by STT, or I/O bound by DB or external API limits). Use this to inform configuration (like how many replicas we might need per X calls). Ensure the autoscaling (in Kubernetes) responds by adding pods under high load and that no single pod gets overwhelmed if we intend one call per pod scenario.

- *Sub-task:* Perform a **longer run soak test**: simulate moderate traffic (like 5 concurrent calls continuously for an hour) to see if memory usage stays stable (no memory leaks) and if the system remains responsive. Monitor for any errors or crashes.
- *Sub-task:* Verify that the system meets the defined **SLOs** for performance and error rates. If the design target is, say, < 3s response time for each turn at p95 and 99% uptime [86], check if we are close. If not, plan mitigations (e.g., more powerful instances, further code optimizations, etc.). Document these findings and adjustments.

- **Quality Assurance Processes:** Establish ongoing QA practices in line with the safety standard.

- *Sub-task:* Maintain a **test case checklist** or BDD specs for key user stories (as described in the WIKI use cases [22] [87]). For example, have a checklist for "Inbound Call - Emergency" scenario: verify that outcome matches expectations (emergency flagged, SMS sent, earliest slot offered, etc.). This can guide manual testing before each release.
- *Sub-task:* Do a **UX/call flow audit** with non-developers if possible (have someone else interact with the system thinking it's real, and gather feedback on where it might be confusing or fail). Use this feedback to create issues for improving prompts or handling certain natural language variations.
- *Sub-task:* Set up a **staging environment** that mirrors production (with perhaps a separate Twilio number and test Google Calendar) where new versions can be deployed and tested by the team (and maybe by the client) before production. This environment can be used to run the integration tests and manual tests as final verification.
- *Sub-task:* Implement a process for **regression testing**: whenever a new feature is added or bug fixed, run through the core call flows and critical scenarios to ensure nothing else broke. Automate what we can, and keep a short manual regression list for things not easily automated (like listening to the voice prompts to ensure quality).
- *Sub-task:* Continuously track **error logs and user reports** once the system is live. If an incident occurs (e.g., the AI mis-scheduled or missed an emergency), perform a *blameless postmortem* [88] per the standard: find root cause, add tests or safeguards to prevent it next time, and update documentation. Treat these follow-ups as high-priority tasks to maintain the system's reliability.

Observability & Monitoring

- **Monitoring Dashboards:** Implement robust monitoring and alerting for the system's health [86].
- *Sub-task:* Finalize the **/metrics endpoint** which already exposes global and per-tenant metrics [68]. Ensure it includes all relevant metrics: total requests, errors, active sessions, average latency perhaps, and per-business breakdowns for key stats (so we can see if one tenant is having issues). If needed, add custom metrics collection in code for things like STT response time or Twilio callbacks success.
- *Sub-task:* Set up **dashboards** in a monitoring tool (could be Grafana if using Prometheus, or directly in GCP Cloud Monitoring since GKE can export metrics). Include graphs for request rate, error rate, voice session duration distribution (to watch P95/P99 latency), CPU/memory of pods, etc. Also track external calls like Twilio API errors or Google API quota usage.
- *Sub-task:* Define **Service Level Objectives (SLOs)** such as uptime (e.g., 99.9% monthly), error rate (<1% of calls result in error), and latency (e.g., <2s for STT response 95% of the time) based on the business requirements [86]. Configure **alerts** that trigger if these SLOs are threatened – for example,

if error rate spikes above a threshold in a 5-minute window, or if no voice sessions succeeded in the last X minutes (which could indicate a system outage or Twilio issue).

- *Sub-task:* Implement **uptime monitoring**: use an external service or Cloud Monitoring to regularly hit a health endpoint (or just the `/docs` or root URL) to check if the service is up. This will catch issues like the service being down or unreachable from the internet.

- *Sub-task:* Monitor **third-party integrations** as well. For instance, if Google Calendar API starts failing (perhaps due to expired token or API changes), have an alert or at least a log that's noticeable. Similarly, if Twilio is sending errors (the admin Twilio health metric can reveal if Twilio calls are failing frequently for any tenant [50]), surface that so we know to investigate (could be misconfigurations or Twilio outages).

- **Logging and Tracing:** Improve logs for better debugging and possibly add tracing.

- *Sub-task:* Ensure that all major events are logged with context. Use structured logging (key-value pairs) if possible so that logs can be filtered by session, business_id, etc. For example, log the start of a call with the call SID and business_id, log each stage transition in the conversation (with stage name and perhaps snippet of user input), and log the outcome (scheduled/canceled/emergency). These logs will be invaluable for debugging issues reported by users.
- *Sub-task:* Redact or omit sensitive info in logs. Don't log full customer personal info or transcripts at INFO level. Perhaps log that an appointment was created for customer ID X rather than name/address, to be privacy-conscious, unless troubleshooting needs the full data (those could be DEBUG level if needed).
- *Sub-task:* If feasible, integrate a **tracing system** like OpenTelemetry. This could allow insight into performance bottlenecks (e.g., a trace for a voice session might show spans for STT, for DB write, for TTS, etc.). Even if not implementing fully, design an issue for adding tracing instrumentation in the future, especially if the system grows more complex or if debugging distributed issues (like between our service and external APIs) becomes important.

- *Sub-task:* Log **metrics** to logs as well if needed – sometimes having key metrics printed can help (though we have /metrics). For example, log at the end of each call: "Call [id] completed – duration X sec, [n] turns, booked: true/false, emergency: true/false." This summary log line can be used to quickly scan how calls are generally going, without digging into metrics UI.

- **Operational Runbooks & On-call:** Utilize runbooks to handle incidents and maintenance [89].

- *Sub-task:* Review and update the **RUNBOOK.md** and **PILOT_RUNBOOK.md** with any new procedures. For instance, if we've added steps for "What to do if Google Calendar API quota exceeded" or "Twilio number misconfigured – how to check", ensure those are captured in the runbook [81]. Include troubleshooting tips for common alerts (e.g., if error rate alert fires, check X, Y, Z).
- *Sub-task:* Define an **on-call rotation or responsibility** even if it's just one person (the developer) for now. The engineering standard emphasizes blameless postmortems [88]; prepare a template for incident reports in the repo so that if something goes wrong, it can be documented and learned from.
- *Sub-task:* Schedule **regular drills** or at least reviews: e.g., test the backup restore process, or do a mock incident walkthrough from the runbook to ensure it's accurate. This might be overkill at this stage, but noting it as a practice to adopt is good (perhaps mark as future to-do if team grows).

- *Sub-task:* Monitor **costs and resource usage** as part of operations. Given this is on GCP with potentially heavy AI workloads, keep an eye on Cloud costs (STT/TTS API usage, Twilio bills, etc.). Set up budgets/alerts on GCP and Twilio so that any runaway usage is caught early. This is more of a governance task but important for a sustainable service.

Each of these issues and sub-tasks aligns with the project's design documents and the RavDevOps engineering principles. By tackling this comprehensive list, the team will improve code quality [2], complete all planned features, ensure rigorous testing [76], and deploy a reliable AI telephony service that is efficient, scalable, and ready for production on GCP [1]. This roadmap takes the project from its current prototype stage to a fully polished product, covering every feature, service component, test, and consideration for success.

Sources:

- Project Backlog and Design Documents – for feature requirements and engineering standards [90] [77]
- RavDevOps Engineering Code & Safety Standard – guiding coding practices, testing, and operations [91] [88]
- AI Telephony Service README and Wiki – for architecture, use cases, and integration points [60] [36]

---

[1] [2] [3] [5] [14] [15] [17] [18] [19] [26] [31] [35] [39] [46] [53] [54] [58] [59] [61] [62] [70] [73] [77] [78] [82] [86] [89] [90]
BACKLOG.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/BACKLOG.md

[4] [6] [16] [21] [27] [32] [33] [34] [41] [42] [44] [45] [47] [48] [49] [50] [51] [52] [57] [60] [63] [64] [65] [66] [67] [68] [69] [71] [72] [74]
README.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/README.md

[7] [13] [75] [81] RELEASES.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/RELEASES.md

[8] [9] [20] [22] [23] [24] [25] [28] [29] [30] [36] [37] [38] [40] [43] [76] [87] [88] [91] WIKI.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/WIKI.md

[10] ff6d65b6-f5af-4d09-92cf-21dad48dd201.pdf
file://file_000000001ef471f58c45d9696bc86c9b

[11] [12] [79] [80] [83] [84] [85] TOOLS.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/TOOLS.md

[55] [56] sw.js
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/a236ed1bc710b188e0a8518f2fbc2d499e99e245/chat/sw.js