ChatGPT

AI Telephony Service & CRM Repository Review

Code Quality and Structure

The codebase demonstrates **high code quality** with a logical modular structure. The backend is implemented in Python (FastAPI) with clear separation of concerns: configuration, database models, routers (API endpoints), and service logic are all in distinct modules. Naming conventions follow Python standards (snake_case for variables, CamelCase for classes), and type hints are used. The repository maintains **consistent style and formatting** by using tools like *Black* and *Ruff* in CI [1]. This enforces PEP8 compliance and a uniform code style across the project. Functions and classes are generally small and focused, aiding readability. For example, the FastAPI router functions are kept thin – they primarily validate input and delegate to the service layer – which aligns with the project's contributing guidelines [2]. Critical operations are wrapped with error handling and logging; the code logs important events (e.g. conversation start, errors) and updates metrics accordingly [3], which improves maintainability and observability.

Overall, the code is easy to navigate and understand. There are docstrings or comments explaining non-trivial logic (e.g. the conversation flow state machine is commented stage by stage), and the project even provides a **Contributing Guide** that emphasizes clarity over cleverness and keeping modules cohesive [4] [2]. The presence of automated linters and formatters in the development workflow further ensures that new contributions remain clean and consistent.

Suggestions for Improvement:

- *Refactor complex modules gradually:* The `ConversationManager` (voice assistant logic) is quite lengthy (~750 lines) as it handles many dialog stages. While it's well-structured with if/elif stages, consider breaking it into helper functions or state-specific handler methods if it grows further. This would improve readability and make the dialog flow easier to modify or extend.
- *Increase inline documentation:* Key classes and functions could benefit from docstrings explaining their purpose. For example, adding docstrings to each router endpoint method (to supplement the API docs) and to complex service functions (like those in `conversation.py`) would aid future maintainers.
- *Continue enforcing coding standards:* The project already uses **Black**, **Ruff**, and **mypy** in CI. Ensure all modules are covered by type checking (currently CI runs mypy on select files [5]). Expanding mypy to the entire codebase once types are stable will catch more issues early.
- *Keep functions focused:* Thus far the code follows this well. As new features are added, continue the practice of small, single-purpose functions and methods. Avoid overly deep nesting or long functions by refactoring logic into smaller pieces when needed.
- *Leverage Python features:* Consider using dataclasses or Pydantic models for internal data structures where appropriate to simplify boilerplate. For instance, the `ConversationResult` and session state could potentially be dataclasses for clarity (though this is a minor style preference).

By addressing these points, the code will remain **clean, readable, and maintainable** as the project evolves.

Project Architecture and Scalability

The project's architecture is well-planned and oriented toward future scalability. It follows a **modular monolith** approach: one FastAPI application hosting multiple logical components (telephony voice assistant API, Twilio webhooks, CRM endpoints, etc.), plus a lightweight static frontend (dashboard and widget). This separation is evident in the code, where different routers handle different domains (voice, SMS, CRM, admin) [6]. The architecture is **layered**: API routers delegate to service modules, which in turn use repository/DB layers, keeping a clear separation of concerns. For example, the "voice assistant backend" encompasses REST APIs, a conversation manager, and integrations (STT/TTS, calendar, SMS) [7] [8], while the web dashboard is an independent static client interfacing via those APIs [9]. This design promotes scalability by allowing each piece to be modified or scaled somewhat independently.

Notably, the system is built with **multi-tenancy in mind**. The code checks for multiple business tenants on startup and can enforce tenant-specific API keys [10] [11]. Each request is resolved to a `business_id` via headers, and the `ensure_business_active` dependency blocks suspended tenants [12]. In single-tenant dev mode, it gracefully falls back to a default tenant. This flexibility means the architecture can support multiple client businesses down the line with minimal changes. The data model is also scoped by business (customer, appointments, etc. all carry a `business_id`) [13] [14], ensuring data isolation per tenant.

Scalability considerations are evident: the app can run with an in-memory store or a real database depending on environment profile [15]. By default, external integrations (calendar, SMS, voice) have *stub modes* which allow development without heavy dependencies, but in production those can be switched to real providers via environment variables [16]. This design allows easy substitution of components (e.g. swapping the STT/TTS engine or SMS provider) and supports scaling out functionality by configuring different backends.

The deployment architecture targets cloud-native scalability. There is a Dockerfile for containerization and a Kubernetes manifest provided. The K8s config defines a deployment with multiple replicas and an autoscaler (HPA) that can scale pods based on CPU [17] [18]. Health and readiness probes are configured [19] to facilitate stable load balancing. These choices mean the service can be scaled horizontally to handle higher loads (with the caveat that the in-memory session store would need to be adjusted for truly stateless scaling).

Suggestions for Improvement:

- *Externalize session state for scaling:* Currently, call session data is kept in memory (as indicated by `SESSION_STORE_BACKEND=memory` default [20]). In a multi-replica scenario, this could lead to lost session context if a call's requests hit different pods. In the future, consider implementing a distributed session store (e.g. Redis) or use a database table for sessions when running with >1 replica. This will enable truly stateless scaling and resilience (so one instance can pick up a call session from another).
- *Database scalability:* For now, a single database (SQLite or a single Postgres instance) is used. As tenants or traffic grow, ensure the database can scale (migration to a managed SQL or sharding by tenant if needed). The data model is straightforward and should scale to moderate volume, but keep an eye on indexing (e.g., ensure indexes on foreign keys like `business_id` for large multi-tenant tables).

- *Microservice boundaries:* The monolithic design is manageable now, but monitor if any component becomes a bottleneck. For example, if speech transcription or calendar integration start consuming significant resources or latency, you might split them into a separate service or background worker. The current architecture already isolates these logically (via service classes), so extracting them later (e.g., a dedicated microservice for heavy TTS processing) is feasible if needed.
- *Caching and performance:* Consider introducing caching for repeated queries to external services. For instance, if certain calendar data or business config is frequently accessed, an in-memory cache (with TTL) could reduce external calls and speed up responses. The current design's use of fast async HTTP calls and efficient libraries is good; just be ready to add caching if latency becomes an issue.
- *Maintain configuration profiles:* The project uses environment profiles (in-memory vs db-backed, stub vs real). Continue to maintain these profiles and test both modes. This ensures that as the system scales or moves to production, configuration toggles (like turning off `CALENDAR_USE_STUB`) won't reveal hidden issues. A `staging` profile similar to production (with all real integrations on) would be useful for load testing the full stack before live deployment.

By anticipating these changes, the architecture will remain **robust and scalable**, capable of onboarding more tenants, handling higher call volumes, or evolving into a suite of focused services if required.

Service Design

The service design within the application is logically organized and exhibits **high cohesion**. Each functional area of the system is encapsulated in its own service module or class, with minimal unnecessary coupling between them. For example, SMS functionality is handled by an `SmsService` class that abstracts sending messages. It provides a stub implementation by default and only calls the Twilio API if the provider is configured as "twilio" [21] [22]. This design means the rest of the application (e.g. conversation flow) doesn't need to know anything about Twilio – it just calls `sms_service.send_sms(...)` and the service handles details internally. Such **abstraction layers** are present for other concerns as well: the speech-to-text and text-to-speech logic is likely abstracted behind a `speech_service` (with a stub vs. real provider), and calendar integration behind a `calendar_service`. This makes each service component independent and easier to test or replace.

The coupling that does exist is deliberate and managed. The **ConversationManager** (in `services/conversation.py`) orchestrates across multiple services – it invokes the speech service, uses NLU helpers, updates the CRM via repositories, etc. – but this is appropriate as it acts as the core "brain" coordinating those parts. Outside of this orchestrator, most services remain self-contained. The telephony integration is a good example of **low coupling**: the Twilio-specific webhook handler is kept separate in its own router (`twilio_integration.py`), which primarily adapts Twilio's HTTP POSTs into calls to the generic conversation/telephony services [23]. This means Twilio logic (like signature verification, TwiML formatting) doesn't pollute the core conversation code, and if another telephony provider were used, one could add a new adapter without changing the conversation service.

Each service is also designed with **single responsibility** in mind. The `SmsService` only handles messaging (including owner/customer notifications) and keeps track of messages sent for auditing [24] [25]. The calendar service (from glimpses in the design docs) encapsulates Google Calendar interactions (finding slots, creating events). This separation ensures that changes in one service (say, switching SMS provider or modifying calendar logic) have minimal impact on others.

Suggestions for Improvement:

- *Define clear interfaces for providers:* Right now, services use internal `if` logic to choose stub vs real implementations (e.g., SMS provider check inside `send_sms`). As the codebase grows, consider formalizing this via strategy patterns or interface classes. For instance, an abstract `SpeechProvider` class with `transcribe()` and `synthesize()` methods, implemented by a `OpenAISpeechProvider` and a `StubSpeechProvider`. The `speech_service` could then hold an instance of the chosen provider. This isn't strictly necessary now, but would make it easier to add new providers (say, a different TTS engine) without cluttering the service code with conditionals.
- *Reduce shared state via dependency injection:* Many services are imported as singletons (e.g. `sms_service = SmsService()` is a module-level instance [26]). This is convenient, but for testability and decoupling, it can help to inject these into components that use them. For example, the ConversationManager could accept a `sms_service` and `calendar_service` in its constructor. This would allow swapping out implementations in different contexts (like using a fake SMS service in tests explicitly, though the design already defaults to stubs which mitigates this). It's a minor point, as the current global singletons are working due to the small scope.
- *Increase cohesion within services:* Most services are already cohesive. Just ensure future enhancements don't overload a single service with too many responsibilities. For instance, if the conversation flow logic grows to handle significantly different scenarios (like separate flows for owner vs customer calls), you might split those into separate managers or state machines. Keep each service focused on a specific sub-domain (telephony, messaging, scheduling, etc.).
- *Document service boundaries:* It might help new developers if the README or architecture docs included a brief section on "Service Components", explaining which module handles what (e.g. "**Speech Service** – handles speech-to-text and text-to-speech (see `services/stt_tts.py`), **SMS Service** – handles outgoing texts (`services/sms.py`), **Conversation Manager** – core dialogue logic, etc.). While the code is self-explanatory to those who read it, a high-level summary in docs could speed up understanding of the moving parts.
- *Monitor coupling via metrics:* The current design tracks metrics per service (SMS sent, errors, voice sessions, etc.). Continue to use these to identify if any service becomes a choke point. For example, if metrics show a high number of errors or slow responses in one area, it might indicate that service's design needs to be revisited or scaled out. Thus far, the design's decoupling has been effective in isolating issues (e.g., a Twilio failure won't crash the app – the SMS service simply falls back to stub and logs it).

In summary, the **service design is solid**, with well-defined boundaries and minimal tangling between components. The suggestions above aim to preserve that clarity as the codebase grows and ensure each part of the system remains modular and replaceable.

CI/CD and DevOps

The repository exhibits strong DevOps practices and a good CI/CD foundation. A GitHub Actions workflow is in place for Continuous Integration, which triggers on pushes/PRs for the backend code [27]. The **CI pipeline** installs the backend in a fresh environment and runs a series of checks: linting with Ruff, formatting check with Black, static security analysis with Bandit, type checking with mypy, and then the test suite with coverage enforcement [28] [29]. This comprehensive automation ensures that code quality remains high and that security issues or type errors are caught early. Notably, the CI uses a test coverage

threshold (currently 62% required) [30] – this is a good start to prevent test regression, and can be raised over time to encourage even more testing.

Security and code quality are further reinforced by a CodeQL analysis workflow [31], which scans the code for vulnerabilities on a scheduled basis and on new commits. The presence of a `codeql.yml` and the Bandit step means the project is proactively checking for common security pitfalls (like SQL injection, use of secrets, etc.) as part of CI, aligning with the team's emphasis on safety.

On the **CD (Continuous Deployment)** side, the groundwork is laid but appears to require some manual steps (as is common in early-stage projects). The repository includes a `backend/Dockerfile` and documentation for deploying to GKE (Google Kubernetes Engine). The *Deployment Guide* outlines building the Docker image and mentions a GitHub Actions release workflow that can build/push images when a tag is created [32] [33]. There is also a Kubernetes manifest (`k8s/backend.yaml`) that defines the deployment, service, ConfigMap, and Secret for the application [34] [35]. Impressively, the manifest even includes an autoscaler (HPA) for the backend, indicating forethought in handling variable load [18]. Environment-specific configuration is managed via ConfigMap/Secret, which is a robust practice for separating config from code.

The DevOps setup demonstrates that the team values **automation and reliability**: they have pre-commit hooks configured for local checks [36], a detailed deployment checklist, runbooks for operations, and even load testing scripts [37] to verify performance. These are signs of a mature engineering culture around deployment and operations.

Suggestions for Improvement:

- *Raise the test coverage bar:* With a current minimum of 62% test coverage [30], there is room to aim higher. As the project stabilizes, consider increasing this threshold (to 70%+, and eventually ~80-90%). This will encourage contributors to write tests for new code. Also, expand coverage to any untested areas; for example, ensure error paths and edge cases in each service have corresponding tests.
- *Expand mypy type checking:* The CI runs mypy on a few core files but not the entire codebase [5]. Gradually extending static type checks to all modules (once any blocking type issues are resolved) can catch bugs before they hit runtime. This can be done by removing the file filters in the mypy step and addressing any new type errors that appear.
- *Automate release deployments:* The documentation references a GitHub Actions workflow for releases (building Docker images on git tags), but it wasn't directly seen in the repository search. If not already implemented, set up that **CI/CD pipeline** to automatically build and push Docker images when a new version is tagged. This could be extended to deploy to a staging or production environment (perhaps using GitHub Environments or a tool like ArgoCD if using Kubernetes). Even if production deployment isn't fully automated, having an automated image build saves time and reduces human error.
- *Infrastructure as Code:* The single `backend.yaml` manifest is great for a starting point. As the infrastructure grows, ensure it remains in version control (which it is) and perhaps break it into components (deployments, services, config) if it becomes large. Also, validate the K8s config as part of CI (there are tools that can lint Kubernetes YAML). This can catch config issues early.
- *Monitoring & Logging:* DevOps doesn't end at deployment. While outside the repository's scope, ensure that the deployed service has monitoring. The app exposes a `/metrics` endpoint for Prometheus [38] – setting up Prometheus/Grafana dashboards and alerts based on these metrics

(e.g., high error rate, increasing voice_session_errors [39], etc.) will be crucial in production. Similarly, ensure logs are aggregated (the structured logs with key events are already in place).

- *Secrets management:* The project correctly uses environment variables and K8s Secrets for sensitive data (API keys, tokens) [40]. Double-down on this by integrating a secrets manager or vault if complexity grows. Also, periodically rotate keys (the admin API provides an endpoint to rotate the widget token, which is a good practice [41] [42]).
- *CI performance:* As tests and linters grow, CI might slow down. Keep an eye on CI run time and consider strategies like caching dependencies, or splitting jobs (e.g., run linting in parallel with tests). The current setup is fine, but optimization can keep the feedback loop fast for developers.

By continuing to invest in these CI/CD improvements, the project will maintain a **fast and reliable delivery pipeline**, enabling the team to ship updates confidently and frequently.

Documentation and Onboarding

Documentation is a major strong point of this repository. The amount and quality of written guides, reference materials, and design documents is exceptional. The README provides a clear overview of the project's purpose, features, and quick start instructions for development [43] [44]. It also points to an API reference and various policy documents, signaling to developers what resources are available [45] [46]. Beyond the README, the repository includes a comprehensive set of Markdown documents covering architecture (e.g. `OUTLINE.md`), a detailed API reference for each endpoint [47], data model explanations [48], security and privacy policies, an engineering whitepaper, runbooks for operations (`RUNBOOK.md` and `PILOT_RUNBOOK.md`), a deployment checklist, and more. This level of documentation is rarely seen in early-stage projects and greatly facilitates onboarding.

New developers are given explicit guidance on how to get started. The **Developer Workflow** doc outlines step-by-step how to set up the environment, run the app with different profiles, seed data, run tests, and even perform load testing [49] [50]. The Contributing guide and Engineering guide emphasize reading the design PDFs and aligned markdown docs before making changes, ensuring that contributors understand the context and standards [51] [52]. There's even traceability mentioned: many of the docs are distilled from higher-level design PDFs (e.g., business analysis, project plan) [53], which grounds the documentation in real requirements and use cases. This helps maintain a clear vision and rationale behind the code.

For onboarding, the presence of example `.env` files (like `env.dev.inmemory` and `env.dev.db`) and instructions to use them is very helpful. The quickstart shows how to launch the backend and open the dashboards, including what headers/tokens to use [54] [55]. This means a developer or tester can follow the steps and have a working system with minimal friction. Additionally, the system provides demo data seeding via the `/v1/admin/demo-tenants` endpoint [56], which is documented in the workflow. This is fantastic for trying out the application without having to manually create data.

Suggestions for Improvement:

- *Maintain documentation rigor:* With so many documents, one challenge is keeping them in sync with the code. Continue to update the docs as features evolve (the team already notes this in the Dev Workflow checklist for updating docs on behavior changes [57]). A good practice is to treat docs like code: require updates to relevant docs in each PR that changes functionality.

- *Add architecture diagrams:* The documentation is text-heavy. Introducing a few diagrams could greatly benefit comprehension. For example, a high-level architecture diagram showing how a call flows from Twilio -> backend -> services -> calendar, or how the components (voice API, CRM API, dashboards, DB) interact. Even simple flowcharts or sequence diagrams (perhaps derived from the call flow in the WIKI [58] [59]) would complement the detailed textual descriptions and help visual learners.
- *Consolidate or cross-reference docs:* Ensure that if information overlaps (say, the README and the OUTLINE both describe architecture), they reference each other or are clearly distinct to avoid divergence. It might help to have a single source of truth for certain details (for instance, API_REFERENCE.md could be generated from the FastAPI docs or openAPI schema to avoid manual errors – tooling can assist here).
- *Include example scenarios:* Consider adding a short "walkthrough" in the README or wiki: e.g., an example of an inbound call and how the system responds (with snippets from logs or API calls). Some of this is in the WIKI use cases. Highlighting one end-to-end scenario in the README might give new stakeholders a quick insight into system behavior.
- *User/Operator documentation:* In addition to developer docs, think about if any end-user documentation is needed (perhaps not, since this is a backend system and the "users" are internal or the business owner via the dashboard). The runbooks serve operators well. Just ensure the *RUNBOOK.md* (for on-call engineers) is kept current as deployment or operational procedures change.
- *Leverage the wiki or GitHub Pages:* The project has a WIKI.md inside the repo. Another approach could be to use GitHub's wiki or GitHub Pages for more user-friendly browsing of documentation. This is optional, but sometimes having the docs in a published site or a structured wiki can ease navigation. The markdown files are already well-structured, so this would be mostly about presentation.

In summary, the project's documentation is **excellent and thorough**. The above suggestions are mostly about keeping the momentum: as the system grows, continue the practice of well-documented designs and processes, and consider adding visuals to augment the text. This will ensure that new team members, as well as external stakeholders, can quickly get up to speed on the system.

Testing

The testing strategy for this project is robust, covering both unit-level logic and high-level integration tests. The repository includes a `backend/tests/` directory with tests for various functionality (e.g., conversation flow, Twilio integration, CRM/owner routes, admin endpoints). The tests make good use of FastAPI's `TestClient` to simulate HTTP requests to the API, allowing end-to-end verification of routes in memory [60]. For example, there are tests that create demo tenant data via the admin API and then retrieve usage stats, asserting the correctness of the results [56] [61]. This ensures that the system's REST endpoints behave as expected in a realistic scenario.

Particularly impressive is the coverage of the **conversation logic**. Tests in `test_conversation.py` instantiate the `ConversationManager` and simulate a full call flow, step by step [62]. They check that after greeting, the state moves to ASK_NAME, then after providing a name, it asks for address, and so on through scheduling. Edge cases like emergency detection are also tested – e.g., input containing "flooding" triggers the emergency flow and the state reflects `is_emergency=True` [63]. There are tests for returning customers (ensuring the assistant greets them differently and reuses stored address) [64] [65]. This level of

detail in tests shows a strong **test-driven approach** to the conversational aspect, which is the core complexity of the system.

The project also tests things like the Twilio webhook behavior (likely by simulating form posts and checking the TwiML response or side effects) and SMS opt-out flows, as hinted by the presence of `test_twilio_integration.py` and others. The admin and CRM routes are tested for proper authorization (skip conditions in tests indicate they require the database to be enabled, which is handled via the `SQLALCHEMY_AVAILABLE` flag in tests [66]). The test suite is integrated into CI, and coverage is measured, which is great for preventing untested code from slipping in.

In addition, the developers have provided a **load testing tool** (`load_test_voice.py`) to simulate multiple concurrent voice sessions [37]. While this might not be part of automated CI, it's extremely useful for manual performance testing and capacity planning. It shows foresight in validating how the system behaves under stress (20 sessions with concurrency, etc.).

Suggestions for Improvement:

- *Increase breadth of tests:* While core functionality is well-tested, ensure that every API endpoint has at least one test. For instance, if there are less critical endpoints (maybe some GET info endpoints or the widget-related routes), add simple tests for them. This helps catch any contract changes or integration issues (e.g., if a refactor accidentally breaks an output format).
- *Test failure modes:* Augment tests to cover failure scenarios and security aspects. For example, test that protected routes indeed return 401/403 when called without proper tokens/keys (and 200 when called with them). There is likely some of this already (the dependency `ensure_business_active` and `require_owner_dashboard_auth` could be tested by hitting an owner route without the token to see it rejects). Similarly, tests for Twilio signature verification (if enabled) could simulate a bad signature to ensure the app rejects it. These kinds of negative tests solidify the security posture.
- *Mock external services in tests:* Currently, external calls (Twilio API, Google API) are stubbed out by design, which is excellent – it means tests don't rely on external connectivity. If any call isn't stubbed, consider using responses or httpx mocking to avoid real HTTP calls in tests. The code's default to stub providers likely covers this, so just maintain that strategy.
- *Performance tests in CI:* Consider running a basic performance test as part of CI or a nightly job. For example, using the provided load test script with a small number of sessions (to not overload CI) just to catch any drastic performance regressions. This might be overkill, but it could be useful if performance is critical.
- *Continuous fuzz or property testing:* For the conversation logic, one could employ property-based testing (with a tool like Hypothesis) to generate random sequences of inputs and ensure no crashes or illogical states occur. Given the deterministic state machine design, this is optional, but could be interesting for uncovering edge cases (like unusual input strings). Even without this, the current targeted tests are very effective.
- *Increase coverage metrics:* Tying back to CI, as mentioned earlier, raising the coverage requirement will encourage filling any test gaps. Before raising it, identify what's missing – e.g., if coverage is lower, is it due to lines in `main.py` or `logging_config.py` not being hit, or some branches in conversation logic not exercised? Add tests accordingly, then bump the threshold.
- *End-to-end testing in staging:* Outside of unit tests, when deploying to a staging environment, perform some end-to-end smoke tests (perhaps via Postman or integration tests) to ensure the system works with real external services (Twilio test credentials, Google Calendar API with a test

calendar, etc.). This isn't part of the repo per se, but it's a testing practice that ensures what passes in unit tests also works in the deployed context with all integrations live.

The testing practices in place already give confidence in the system's correctness. By extending them as suggested, the project can achieve even more **reliability and safety**, catching bugs early and ensuring that new changes don't introduce regressions. The combination of a thorough automated test suite and available load testing tools positions the team well to maintain quality as the project grows.

---

[1] [5] [27] [28] [29] [30] backend-ci.yml
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/.github/workflows/backend-ci.yml

[2] [4] [36] [51] [52] [53] CONTRIBUTING.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/CONTRIBUTING.md

[3] voice.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/routers/voice.py

[6] [10] [39] main.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/main.py

[7] [9] [43] [44] [45] [46] [54] README.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/README.md

[8] PHASE1_VOICE_ASSISTANT_DESIGN.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/PHASE1_VOICE_ASSISTANT_DESIGN.md

[11] [12] deps.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/deps.py

[13] [14] [48] DATA_MODEL.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/DATA_MODEL.md

[15] [16] env.dev.db
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/env.dev.db

[17] [18] [19] [20] [34] [35] [40] backend.yaml
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/k8s/backend.yaml

[21] [22] [24] [25] [26] sms.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/services/sms.py

[23] telephony.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/backend/app/routers/telephony.py

31 codeql.yml
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/.github/workflows/codeql.yml

32 33 DEPLOYMENT_GCP_GKE.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/DEPLOYMENT_GCP_GKE.md

37 49 50 55 57 DEV_WORKFLOW.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/DEV_WORKFLOW.md

38 47 API_REFERENCE.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/API_REFERENCE.md

41 42 56 60 61 66 test_business_admin.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/tests/test_business_admin.py

58 59 WIKI.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/WIKI.md

62 63 64 65 test_conversation.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/e72995ff0a098fceddcf3d2a216721581984a929/backend/tests/test_conversation.py