ChatGPT

AI Telephony Service & CRM Audit Tasklist

Consistency

- [ ] **Enforce Uniform Code Formatting:** Run the project's formatters (`ruff` and `black`) on the entire codebase and fix any remaining inconsistencies. Ensure all Python code adheres to PEP 8 (this is largely maintained by CI checks) [1]. Automate this via pre-commit hooks so every commit uses the same style.
- [ ] **Standardize Naming Conventions:** Review class and file names for consistency. For example, unify how database models are named – currently `Business` (no suffix) vs. `CustomerDB`, `AppointmentDB` (with suffix) [2] [3]. Decide on one convention (either add "DB" to all ORM models or remove it from those classes) and apply it across the codebase.
- [ ] **Normalize Environment Variable Usage:** Eliminate duplicate env var names. The owner dashboard token is referred to by two names (`OWNER_DASHBOARD_TOKEN` and legacy `DASHBOARD_OWNER_TOKEN`) [4] [5]. Choose a single canonical name and update the code and documentation to use it exclusively (while possibly supporting the old name for backward compatibility if needed).
- [ ] **Ensure Consistent API Endpoint Patterns:** Audit all API routes for naming and versioning consistency. Most endpoints are versioned under `/v1` (e.g. `/v1/owner/*`, `/v1/crm/*`) [6], but some like Twilio webhooks and telephony use top-level paths (`/twilio/*`, `/telephony/*`) [7]. Consider prefixing those with `/v1` or clearly document why they are exceptions. Also, verify that multi-word endpoints use a consistent style (hyphen vs. underscore; e.g., `/propose-slots` uses a hyphen).
- [ ] **Review Project Structure for Clarity:** The overall layout is logical (Python backend under `backend/app`, static frontend in `dashboard/` and `widget/`, tests in `backend/tests/`). Ensure each directory's purpose is clearly documented. If the generic `app` module name could cause confusion, consider renaming it to something more descriptive (e.g., `telephony_service`) to avoid generic names, and update import references accordingly. Maintain consistency in how sub-packages are organized (routers, services, etc., already follow a clear pattern [8] [9]).

Quality Assurance (QA)

- [ ] **Integrate Linting & Formatting in Developer Workflow:** Confirm that `ruff` and `black` run on every PR via CI (this is already done in **backend-ci.yml** [1]). To catch issues earlier, set up a local pre-commit config to run these tools before pushes. Address any linter warnings or formatting issues so that `ruff` and `black --check` pass cleanly.
- [ ] **Add Static Type Checking:** Leverage the existing type hints in the code (the project uses Python 3.11+ with hints [10]) by introducing a tool like `mypy`. Include a mypy run in CI to enforce type correctness. This will catch subtle bugs and ensure the code's type annotations remain accurate.
- [ ] **Introduce Security Static Analysis:** Implement a security linter or scanner (for example, Bandit for Python) to detect common vulnerabilities (using hardcoded secrets, unsafe usages, etc.). This can be run in CI to complement the existing tests and linters, aligning with the project's emphasis on safety and static analysis [11].

- [ ] **Measure and Enforce Test Coverage:** Use `pytest --cov` to generate a test coverage report. Aim to include this in CI and fail the build if coverage falls below an agreed threshold (or at least report the percentage in CI output). This will highlight untested code paths. Optionally, publish a coverage badge or report for visibility.
- [ ] **Set Up Code Quality Monitoring:** Consider enabling GitHub's CodeQL analysis or a similar static code analysis workflow for the repository. This can automatically flag security issues and code smells. Although the RavDevOps standards emphasize manual code review, an automated code scan provides an extra layer of QA.
- [ ] **Enhance Developer Tooling:** Provide a `pre-commit` configuration to automate running of tests, linters, and formatters on commits. This ensures consistency and catches issues early. Document its usage in `CONTRIBUTING.md` (e.g., instruct contributors to install the hooks) so that local development mirrors CI expectations [12].
- [ ] **Dependency and Environment Audit:** Review `pyproject.toml` or equivalent for dependency versions. If not already pinned, pin critical library versions to prevent unexpected breaking changes. Additionally, verify that the provided `env.dev.inmemory` and `env.dev.db` profiles still work with the latest code and that all required env vars are documented. This ties into CI by making sure tests run under both profiles if possible (simulate both in-memory and DB-backed runs).

Documentation

- [ ] **Complete API Reference Documentation:** There are placeholders for `API_REFERENCE.md` and `DATA_MODEL.md` that are referenced in the docs but not fully realized [13] [14]. Create or finish these documents with a route-by-route summary of the API and an entity-relationship overview of the data model. This will ensure internal and external users have a single source of truth for API contracts.
- [ ] **Keep Docs in Sync with Code:** Establish a practice (already hinted in `DEV_WORKFLOW.md` [15]) to update documentation whenever behavior or endpoints change. For example, if a new route is added or an existing one is modified, update `API_REFERENCE.md` (for the API) and `CHANGELOG.md` accordingly. Perform an audit of current docs versus the code to identify any discrepancies (e.g., ensure all `/v1/owner/*` and `/v1/admin/*` endpoints in code appear in the reference docs).
- [ ] **Expand the README with Usage Examples:** The README provides a great overview and quick start [16], but it could benefit from concrete examples of API usage. Add example API calls (perhaps a few `curl` or HTTPie examples for common actions like creating a customer or scheduling an appointment) and their expected responses. This complements the interactive docs at `/docs` [17] and helps new users understand the system without running the UI.
- [ ] **Improve Inline Documentation:** Increase the number of docstrings and code comments for complex logic. Critical functions (e.g., emergency detection in `conversation.py`, Twilio request handling in `twilio_integration.py`) should have clear docstrings describing their behavior and edge cases. This will aid future contributors in understanding the code. For instance, the `_build_heuristic_qa_suggestions` logic in the CRM router [18] could be moved to a service function and documented thoroughly, rather than living in-line without explanation.
- [ ] **Audit and Update Operational Docs:** Ensure that operational documentation (`RUNBOOK.md`, `DEPLOYMENT_CHECKLIST.md`, `SECURITY.md`, etc.) reflects the current state of the system. If any configuration fields were added (e.g., new env vars) or if procedures changed, update these docs. The `RUNBOOK.md` should include how to interpret new metrics or logs introduced in recent

updates, and the security doc should reflect all authentication mechanisms (e.g., the need for `X-Owner-Token` on certain routes as enforced in code [19]).

- [ ] **Add Architectural Diagrams:** To complement the textual documentation (like the Architecture Overview in the README), include one or two simple diagrams. For example, a flow diagram of an inbound call (showing Twilio -> our API -> calendar, DB, etc.) or an architecture diagram of components. Visual aids can help newcomers grasp the system more quickly. These can be added to the README or a dedicated `docs/architecture.png` with references.
- [ ] **Review the Contribution Guidelines:** The `CONTRIBUTING.md` is thorough about engineering principles [20] [21] and workflow, but after implementing the above changes, make sure it instructs contributors on new practices (like running mypy or using pre-commit). Add any missing sections, such as coding style examples or a note on how to run the full test suite in both in-memory and DB modes. This keeps contributors aligned with project standards.

Testing

- [ ] **Increase Test Coverage for All Features:** Identify any significant functionality not covered by tests and add tests for it. One notable gap is the **retention** feature – there's a router for retention endpoints but no corresponding `test_retention.py` (the tests directory lists many other areas [22], but not retention). Create tests for retention flows (e.g., sending retention messages, pruning old data) to prevent regressions.
- [ ] **Test Multi-Tenant Scenarios:** Add tests specifically for tenant isolation and authentication. This includes cases like: requests with no API key (should be rejected when `REQUIRE_BUSINESS_API_KEY` is true) [23], requests with a suspended tenant (should receive 403 Forbidden) [24], and proper functioning of `X-Widget-Token` and `X-Admin-API-Key` auth. These tests ensure that the dependency functions `ensure_business_active` and auth requirements in `deps.py` are doing their job under various conditions.
- [ ] **Mock External Integrations in Tests:** Wherever the backend calls external services (Twilio, Google Calendar, OpenAI for STT/TTS), introduce mocks or stubs in tests to simulate success and failure scenarios. For example, simulate a Calendar API failure during appointment booking and assert that the system handles it gracefully (perhaps by returning an error message without crashing). The project already uses stub modes for dev [25]; leverage those stubs or monkeypatch network calls in tests to cover error handling paths (like Twilio signature verification failures, STT timeouts, etc.).
- [ ] **Distinguish Unit vs Integration Tests:** As the test suite grows, consider organizing tests with markers or folders (e.g., `tests/unit` vs `tests/integration`). Unit tests (logic in isolation, using fakes for DB or external calls) should cover most modules, and a few integration tests can cover the whole app (spinning up FastAPI test client with a test DB). This separation can help run fast unit tests on each commit, while integration tests (like those in `test_telephony_endpoints.py` or `test_twilio_integration.py`) can run in CI or on demand.
- [ ] **Utilize Coverage Reports to Find Dead Code:** Use the coverage data to find code blocks not executed by any test. Some defensive branches are marked with `# pragma: no cover` (for example, error handlers in audit logging [26]); evaluate if those could actually be triggered in tests by simulating errors (if so, write tests and remove the pragma). Remove truly dead code or ensure it's covered by tests if it represents important failure-handling logic.
- [ ] **Improve Test Reliability:** Review test cases for any nondeterministic behavior. For example, if any test depends on time (scheduling, TTL expiry) or random IDs, ensure they use fixed inputs or dependency injection for consistency. Given the asynchronous nature of some parts (FastAPI

endpoints), ensure tests await responses properly and use timeouts where appropriate. This will prevent flaky tests in CI.

- [ ] **Add Load/Stress Test Scenarios (Optional):** While unit/integration tests ensure correctness, consider using the provided `load_test_voice.py` script in a controlled CI or staging environment to do performance regression testing. Although not part of the regular CI, having a documented process to run load tests and check metrics (as described in `ENGINEERING.md` and `RUNBOOK.md`) is useful. Ensure any issues found under stress (e.g., memory leaks or slow queries) get corresponding tests or monitoring in place.

Service Provided

- [ ] **Clarify Service Layer Separation:** Audit each API router to ensure it only handles request/response and delegates core logic to service or repository layers. Some routers still contain significant logic (e.g., the CRM router computes QA suggestions via `_normalize_outcome_label` and `_build_heuristic_qa_suggestions` in-line [18]). Refactor these into the `services/` module with clear interfaces. This will improve maintainability and allow independent unit testing of business logic outside the request context.
- [ ] **Enhance Observability with Logging:** Expand structured logging to cover key events and decisions. The app currently logs configuration at startup and warns on insecure configs [27], and it records each request outcome via audit events [28]. Build on this by adding info or warning logs for notable domain events: e.g., log when an emergency appointment is flagged (include appointment ID and customer), when a third-party API call fails (include error details and affected tenant), and when automatic actions occur (like sending a reminder or a retention SMS). Use `logger.exception` for unexpected errors to capture stack traces (this is done in a few places like audit resolution [29]; extend it to other external integration points). These logs will be invaluable in debugging production issues.
- [ ] **Extend Metrics Coverage:** The existing metrics track high-level counts (requests, errors, SMS sent, Twilio calls, etc.) [30] [31]. Consider adding more granular metrics: for instance, track the number of appointments scheduled per tenant, or the latency of external API calls (maybe using custom events for Google Calendar calls). If certain operations are critical (like voice session handling), histogram metrics for their duration could be useful. Exposing these via `/metrics` (perhaps in Prometheus format as already done [32]) and documenting them will allow ops to set up dashboards/alerts (e.g., alert if Twilio error count spikes for a tenant).
- [ ] **Audit Configuration Management:** Verify that all configuration is handled through `AppSettings` and environment variables (no hard-coded values). The current design loads everything from env with sensible defaults [33] [34], which is good. Going forward, if new config options are added, update `config.py` accordingly and list them in docs. Also, enforce that secrets (API keys, tokens) are only loaded from env or a secret manager – the contribution guide already warns not to commit secrets [35]. Periodically run through `DEPLOYMENT_CHECKLIST.md` to ensure nothing is missing (for example, if a new env var `XYZ` is added for a feature, that checklist should mention it for production deployments).
- [ ] **Graceful Failure and Retry Strategies:** Test and harden the service's behavior under failure conditions. For example, if the database is down or slow, do the APIs time out gracefully? If an external API (Twilio or Google) fails or returns an error, is that caught and returned as a 5xx/4xx response without crashing the server? Implement retries or fallbacks where appropriate (e.g., if a Calendar booking fails, perhaps retry once or mark the appointment as pending for manual intervention, with a log). Ensure such failure paths are logged (with context) and, if recurring, are

surfaced in metrics (e.g., a metric for "calendar API failures"). Document these behaviors in the RUNBOOK so operators know what to expect.
- [ ] **Validate Security and Access Controls:** Double-check that all service endpoints enforce the intended security measures. According to the design, owner and CRM routes should require `X-Owner-Token` in production [19], and admin routes require `X-Admin-API-Key` [36] – confirm these are indeed in place and there are no routes accidentally unprotected. Consider adding automated tests for this (as noted above) or even runtime checks in non-production modes that log a warning if sensitive routes are hit without auth (to catch misconfiguration).
- [ ] **Plan for Scalability (State Management):** The current implementation keeps some state in memory (session and Twilio conversation maps) which is fine for development and single-instance deployment [37]. For better fault tolerance and scaling, implement the roadmap item to abstract the session store and use a distributed back-end (like Redis) when running in multi-instance mode [38]. This task involves creating a `SessionStore` interface, an in-memory implementation (already implicit in `services/sessions.py`), and a Redis or database-backed implementation. It ensures that adding more server instances or restarting the server won't lose track of ongoing voice calls or SMS threads. Document how to enable the distributed store in the deployment docs once implemented.
- [ ] **Improve Deployment Observability:** In addition to code changes, verify that the service exposes what ops teams need for monitoring. The health endpoint `/healthz` is present [39] – consider adding a more extensive readiness check (e.g., verify DB connectivity or external API availability and expose that status). Ensure logs are in a format suitable for aggregation (structured JSON if possible via `configure_logging`). Given that the logging uses `extra` fields for structure [40], confirm that the logging config outputs those fields (for example, in JSON or key=value format). Update `DEPLOYMENT_CHECKLIST.md` to include log and metric collection setup (e.g., “make sure to scrape the `/metrics` endpoint, and forward logs to monitoring with severity parsed”).

Each task above is designed to improve the consistency, quality, and reliability of the AI Telephony Service & CRM. By checking off these items, the team will align the codebase with best practices and the documented engineering standards, ensuring the service is maintainable, well-tested, and production-ready [20] [41].

---

1 backend-ci.yml
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/.github/workflows/backend-ci.yml

2 3 db_models.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/db_models.py

4 33 34 config.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/config.py

5 23 24 deps.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/deps.py

6 7 27 30 31 32 39 40 main.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/main.py

8 9 22 SOURCES.txt
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/ai_telephony_backend.egg-info/SOURCES.txt

10 12 20 21 35 CONTRIBUTING.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/CONTRIBUTING.md

11 OUTLINE.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/OUTLINE.md

13 14 CHANGELOG.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/CHANGELOG.md

15 DEV_WORKFLOW.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/DEV_WORKFLOW.md

16 17 README.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/README.md

18 19 crm.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/routers/crm.py

25 TOOLS.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/TOOLS.md

26 28 29 audit.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/services/audit.py

36 business_admin.py
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/backend/app/routers/business_admin.py

37 38 41 ENGINEERING.md
https://github.com/raven-dev-ops/ai_telephony_service_crm/blob/03ce6597e67bc2ba0b8e724f22797d44a33a4e7a/ENGINEERING.md