ChatGPT

UGC BCP CRM – Target-State Architecture and Flow Specification

This document defines the **target-state** design of the UGC BCP CRM module. It prescribes a cleaned-up architecture that **eliminates orphaned states**, enforces role-based visibility invariants, strengthens automation (cadence) flows, and aligns with core business rules (single source of truth, no duplicate data). The specification is organized into sections covering pages/routes, data flow from lead intake to customer, folder structure, automation & cadences, query/view mappings, schema validation, and specific fixes. Together these form the single source of truth (SSOT) for the CRM system's desired end-state.

1. Pages and Routes Mapping

CRM Pages and Routes: The CRM module consists of several interrelated pages for different record types. Below is the list of all current pages (in the existing implementation) and the **required pages** in the target state, along with their route paths and purpose:

- **Lead Inbox –** Route: `/crm/lead-inbox` (Marketing Lead Queue). Displays incoming leads awaiting or undergoing triage by marketing. This page shows leads in **New** or **In Review** status (filter logic detailed in Section 5) [1]. It includes UI for triaging leads (e.g. marking as Qualified, Nurture, etc.) and a button to "Send to Sales Pool" for handover [2].
- **Sales Inbox –** Route: `/crm/sales-inbox` (Sales Lead Pool & Work Queue). Shows leads that have been handed over and are awaiting claim by sales, as well as urgent sales tasks. Specifically, it lists leads in the **Handover Pool** (leads marked handover_eligible=true with no owner) [3] and may also list any overdue sales activities (pulled from the Activities list; see Automation below). Sales users can click "Claim Lead" to take ownership [3]. Once a lead is claimed, it will no longer appear here.
- **My Leads (Assigned Leads) –** *New page to add.* Route: `/crm/my-leads`. This page will list leads that have been claimed by the logged-in salesperson but not yet converted to opportunities. In practice, once a lead is claimed it immediately spawns an opportunity (see Lead-to-Account Journey), so this page essentially shows the salesperson's active **lead-turned-opportunity records in early stages**. It is a stopgap listing to ensure no claimed lead becomes invisible between claim and working the opportunity. (If the system immediately treats claimed leads as pipeline opportunities, this page may overlap with the Pipeline page filtered by owner.)
- **Pipeline Board –** Route: `/crm/pipeline` (Sales Opportunity Pipeline). This is a kanban-style board or list of **Opportunities** in various stages [4]. It shows all active sales opportunities (e.g. stages from Prospecting through Negotiation, excluding closed deals). Sales users manage opportunity stages here, and can perform quick actions like "Quick Add Prospect" (create a new opportunity) and updating stages [4]. In target state, this page will also support filtering by owner or account, and a view for overdue opportunities (e.g. deals past their next action date).
- **Activities Planner –** Route: `/crm/activities` (Tasks and Activities). Shows all **Activities** (calls, emails, meetings, tasks) associated with leads, opportunities, or accounts, typically grouped by due date [5]. Users (marketing or sales) can see planned activities, mark them as done, or create new ones. Invariants: every active lead/opportunity must have an owner and a next action scheduled [6]

[7], so this page ensures tasks are visible and up to date. (A future enhancement could include a calendar view, but currently it's a list by date.)

- **Accounts List (Customers) –** Route: `/crm/accounts` (Customer Accounts index). Lists all **Accounts** (organizations) in the CRM [8]. Each Account is the master record for a customer, and can have many contacts, opportunities, and activities. The list view will likely use an "enriched" view combining account info with status badges (e.g. tenure and activity status) [8]. A search bar filters by company name or domain. The target state fixes the **"Add Account"** button to open an Account creation form (or modal) instead of incorrectly linking to the pipeline [9]. In the UI and navigation, "Accounts" will be the primary term (the prior placeholder page `/crm/customers` will be removed or merged into Accounts to avoid confusion).
- **Account Details (360 View) –** Route: `/crm/accounts/[id]` (individual Account page). Shows a 360° view of a single account, including its profile info (address, industry, etc.), related contacts, open opportunities, recent activities, and other sub-sections (invoices, tickets, etc.) [10]. All key sub-records (contacts, deals, activities) tied to this account are visible here. The target state implements the **"Add Contact"** action on this page (bringing up a form to add a new Contact linked to the account) [10]. It also fixes the **"Add Opportunity"** action to properly create or navigate to a new opportunity for this account (rather than just linking generically to the pipeline) [10].
- **Prospecting Targets –** Route: `/crm/targets` (Prospecting workspace). Shows **Prospecting Targets**, which are pre-lead records (often bulk-imported lists of potential prospects) [11]. This page includes filters by target status (e.g. new_target, contacted, engaged, etc.) and search by company or contact. Sales users use this page to manage cold outreach without cluttering the main Leads/Opportunities until a target is qualified. The UI provides a **"Convert to Lead"** action for engaged/qualified targets, which will convert the target into a Lead/Opportunity (see Lead-to-Account Journey) [12]. The target state will fully implement the **Import** function here: an **"Import Targets"** button to upload CSV files and create new targets in bulk (backed by `/api/crm/targets` POST and an `import_batches` log) [13]. *Note:* The prior page labeled "Prospects" (`/crm/prospects`) was a redundant placeholder and will be removed or repurposed since the Pipeline and Targets pages cover its use-cases [14].
- **Import Management –** Route: `/crm/imports`. This page (currently a placeholder) will be implemented to show import histories and status. Users (with proper role) can initiate data imports (e.g. uploading a list of targets or leads) and see past import batches and any errors. This ties into the **import_batches** table and related logic. In target state, this page will include a file upload interface, preview of parsed data, and then utilize an API route to process and insert records (ensuring deduplication via `dedupe_key` and proper ownership assignment).
- **Other Modules –** (For completeness, the CRM sits alongside other top-level modules like Dashboard, KPI, Ticketing, DSO in the sidebar [15]. Each of those has their own pages, not detailed here.)

Route Structure and Nesting: In the current code, CRM pages are under an `app/(protected)/crm/` segment (to enforce authentication). The target state will **simplify the route structure** to match the blueprint's canonical paths [16] [17]. All CRM pages will reside under a single `app/crm/` folder (with appropriate layout to require login). This means moving files out of the `(protected)` wrapper to eliminate any duplicate route segments [17]. For example, `app/(protected)/crm/lead-inbox/page.tsx` becomes `app/crm/lead-inbox/page.tsx`, and so on for all pages. Each entity page will have a clear, singular route (no aliases or overlapping pages). The API endpoints follow a parallel structure under `app/api/crm/`. For instance: - `app/api/crm/leads` – listing and creating leads (GET, POST) [16]. - `app/api/crm/leads/[id]/triage` – triage status update (PATCH) for a specific lead. - `app/api/crm/leads/[id]/handover` – handover lead to sales (POST) [18]. - `app/api/crm/leads/[id]/claim` –

salesperson claims a lead (POST) [19]. - `app/api/crm/leads/[id]/convert` – convert lead to opportunity (POST) [20]. - `app/api/crm/contacts` – create new contact (POST). - `app/api/crm/accounts` – list/create accounts (GET, POST). - `app/api/crm/accounts/[id]` – fetch one account with related data (GET). - `app/api/crm/opportunities` – list/create opportunities (GET supports query params like `?view=pipeline` or `?quick_add=true`) [4]. - `app/api/crm/opportunities/[id]/stage` – change an opportunity's stage (PATCH) via RPC [21]. - `app/api/crm/activities` – list/create activities (GET, POST). - `app/api/crm/activities/[id]/complete` – mark activity done (POST) and trigger follow-up. - `app/api/crm/targets` – list/create targets (GET supports filtering by status, POST for new targets). - `app/api/crm/targets/[id]/convert` – convert a prospecting target to an opportunity (POST) [22] [23]. - etc.

All of these will be organized under a clear folder hierarchy (no mixing of unrelated modules). This structure ensures a one-to-one mapping between front-end pages and backend endpoints: e.g. the Lead Inbox page fetches data from `/api/crm/leads?view=inbox`, the Sales Inbox from `/api/crm/leads?view=handover_pool`, the Pipeline page from `/api/crm/opportunities?view=pipeline`, etc. There should be **no duplicate or dead-end routes** – every menu button and CTA corresponds to a real page or API handler (the audit found broken links like "Add Account" and placeholders like `/crm/customers` which we fix in this target state [9] [14]).

Tabs and Subsections: Several pages contain internal tabs or subviews to separate data by status or owner:

- The **Lead Inbox** may include sub-tabs for "New" vs "In Review" leads (or simply a combined list with badges – in practice both statuses are shown together for marketing's queue [24]). In target state, if needed, this page can have filters or toggle to view New vs In Review separately, but by default it covers both as the *Marketing Inbox*.
- The **Sales Inbox** might show separate sections for "Unclaimed Leads" (handover pool) and "Overdue Activities" – currently the design shows both in one page for a salesperson's immediate attention [3]. We will maintain that, possibly with a tab or accordion for Leads vs Activities.
- The **Pipeline** page is inherently segmented by opportunity **Stage** columns (if a kanban board). Additional filter tabs could be added (e.g. "All Active", "My Deals", "Overdue"). In target state, we will implement an Owner filter and an Overdue toggle as described later.
- The **Activities** page could later have tabs for "My Activities" vs "All" (for managers), or "Planned" vs "Completed", etc. Initially it shows all planned activities (with ability to view completed via filters).
- The **Accounts** page might not need tabs, but could in future separate "Active Customers" vs "Prospects" via filters. For now, it's a single list with search.
- The **Targets** page should provide quick filters for each target status (counts of New, Contacted, Engaged, etc., likely as segment buttons) [11]. This allows focusing on e.g. "Engaged" targets ready to convert.

All pages and tabs adhere to role-based visibility: e.g. the Lead Inbox is only visible to Marketing roles (sales users won't even see the menu or be able to navigate there), while Sales Inbox and Pipeline are for Sales roles [25]. There will be no disabled menu items – if a user isn't allowed to access a page, it won't appear in their sidebar (enforced by role checks in the layout) [26].

2. Lead-to-Account Journey (State Transitions)

This section defines the end-to-end lifecycle of a record moving through the CRM funnel: **Target → Lead → Opportunity → Account**. It enumerates every valid state transition, the required fields and backend logic at each step, side effects (like creating cadence tasks or audit logs), and how visibility of the record changes with each transition. The goal is to ensure that at no point does a record fall into a "dead" state unseen by any user (addressing the current bug where qualified leads vanished) [27] .

2.1 State Machine for Lead Progression

Lead Status (Triage) State Machine: A lead record goes through a marketing qualification process before becoming a sales opportunity. The states and transitions are:

- **New** (initial state) – A new lead has just been captured (e.g. via form or manual entry) and awaits triage.
- **In Review** – The lead is being reviewed by marketing (validating contact info, assessing needs). Marketing can move a New lead to In Review as they begin qualification.
- **Qualified** – The lead is determined to be sales-ready (has valid contact, clear need, etc.). This is a terminal marketing state – next step is to hand over to Sales.
- **Nurture** – The lead is not ready now but could be in the future; marketing will keep it for long-term nurturing (follow-ups). It stays in marketing's Nurture list until it either re-engages or is disqualified.
- **Disqualified** – The lead is rejected (e.g. not a fit or invalid). No further action in pipeline; it's essentially closed for now (though potentially could be revisited or reported on).

For leads that go to Sales, we introduce two additional flags/states: - **Handed Over** – A flag `handover_eligible=true` indicates the lead has been handed to the Sales pool (awaiting claim). In the triage workflow this corresponds to putting a Qualified lead into the **Handover Pool**. - **Assigned to Sales** – Once a salesperson claims the lead, it has an owner (`sales_owner_user_id`) and is in an active working state for that rep (this roughly corresponds to the lead being "in Sales' hands" but not yet converted to an opportunity in the old model).

Finally, when Sales converts the lead: - **Converted** – The lead is converted into an opportunity (and linked to an Account). At this point the lead record is essentially closed/archived from the Leads list.

Below is the lead triage state diagram summarizing these transitions (Marketing states on left, Sales states on right):

[28] [29]

The logic rules for each transition are as follows:

- **New → In Review:** A marketing user begins triaging the lead. (No special fields required beyond what a New lead has; this is just a status update.)
- **In Review → Qualified:** Marketing decides the lead meets the criteria to hand over to sales. At this point, **required fields** must be present: e.g. valid contact info (phone or email), some initial service need or volume/timeline info should be recorded [30] . The backend will enforce that certain minimum data exists before allowing a lead to be marked Qualified (if not, an error prompts user to

Nurture instead) [31] . When a lead's triage_status is set to Qualified, **the system immediately triggers a handover**.

- **Qualified → Handover Pool (Sales Pool):** This is an automatic backend transition in target state. As soon as a lead is marked Qualified, the server will call the **rpc_lead_handover_to_sales_pool** function (or equivalent) to create a record in the `lead_handover_pool` table and set `handover_eligible=true` [29] [32] . This moves the lead into the Sales Inbox without requiring the user to click a separate "Send to Sales" button (which was a source of error before). In other words, **Qualification implies Handover** (removing the chance for a qualified lead to languish unseen) [27] [33] . The handover RPC will log who handed it over, any notes, and optionally a priority flag [34] . The lead's triage_status may be updated to a special value like "Handed Over" for clarity [35] , though the presence in the pool is the primary indicator.
- **Handover Pool → Assigned to Sales:** A salesperson claims the lead from the pool. This triggers **rpc_sales_claim_lead**, which does several things atomically: it locks the pool record (so two people can't claim at once), sets `claimed_by` and `claimed_at` on that pool entry, assigns the lead's `sales_owner_user_id` to the claiming user, and – critically – **creates an Opportunity and Account if not already existing** [36] [37] . At claim time, the system finds or creates the Account (matching on company name or domain) [38] , then creates a new Opportunity linked to that account (populating fields from the lead like service, route, etc.) [37] [39] . The lead's status fields are updated (e.g. legacy `status` might be set to "Contacted" to indicate sales has engaged) [40] , and an initial **Activity** is auto-generated (e.g. "First contact: [Lead Company]" call scheduled for today) [41] [42] . After claiming, the lead now has an owner and an associated opportunity ID – it will appear in that salesperson's pipeline. The lead is effectively out of the marketing list and into sales' world.
- **Assigned (Working) → Converted:** When the sales process advances to a certain point, the lead is formally converted to an opportunity in the system of record. In our target design, this "conversion" is mostly taken care of at claim time (since we already created the opportunity on claim). However, we might define **"Converted"** as the moment when the lead/opportunity results in a sales outcome. One interpretation: when a lead is **successfully closed-won**, it's no longer a lead at all. In practice, by the time an opportunity is closed, the lead record isn't in use. So, converted is more of a final marker that the lead fulfilled its purpose. The implementation we have uses the **/api/crm/leads/[id]/convert** endpoint to handle conversion in cases where a lead wasn't claimed via pool (for example, a lead directly assigned to a salesperson could be converted). This conversion route ensures idempotently that an Account and Opportunity exist (creating them if needed), seeds a standard cadence of follow-up activities on the new opportunity, and updates the lead's status to "Closed Won" with an opportunity link [43] [44] . It logs the conversion in audit tables [45] . After this, the lead will **no longer appear in any lead view** (triage_status isn't used beyond handover; and status=converted/Closed Won disqualifies it from lead queries) [46] . The Opportunity now carries the torch in the sales pipeline.
- **In Review → Nurture:** If a lead lacks the criteria for sales handover (e.g. missing key info or not ready to buy), marketing can set triage_status to **Nurture**. Required fields: none in particular (maybe a note), but by doing so the lead is removed from the active inbox and placed into a Nurture list that marketing will periodically revisit. An audit log should capture this status change (with maybe a reason or next outreach date). Nurture leads will not appear in Sales views at all, only in a marketing "Nurture" page.
- **In Review → Disqualified:** Marketing can disqualify a lead. Required: a **disqualified_reason** should be provided (the system will enforce that a reason/comment is given before finalizing) [47] . The lead's triage_status becomes Disqualified and it moves to the closed leads list. Like Nurture, Disqualified leads are visible only in a special marketing view for reference, not to sales.

- **Assigned → Disqualified (Sales Closure):** A salesperson who has a lead (e.g. they claimed it, but then found it to be a dead end before converting) can mark it disqualified as well. In the current model, once a lead is claimed, the expectation is they convert it (since an opportunity exists). If they decide not to pursue, effectively that opportunity would be marked **Closed Lost**, and we can mirror that by also marking the lead triage_status as Disqualified (to keep data consistent). The target system will treat a Sales-Disqualified lead similar to a Closed Lost opportunity (no further action). This transition will be captured in both the opportunity's stage (Closed Lost) and possibly an audit entry on the lead.

Visibility Implications: At each state, which team can see the record? We enforce a **visibility invariant**: *each lead appears in exactly one place at a given time, and every state has a corresponding view so that no lead is "lost"*. Specifically [24] [33] :

- New/In Review leads – visible in Marketing's Lead Inbox only [48] .
- Qualified leads (not yet handed over) – in target state, this situation is brief or nonexistent, because qualification triggers handover. If there were a slight delay (say, if auto-handover fails), a "Qualified Leads" view for marketing could catch them – but ideally **auto-handover makes this immediate** [49] .
- Leads in Handover Pool – visible in Sales Inbox (all sales can see unclaimed leads) [50] .
- Leads Assigned to Sales (claimed) – visible to that sales owner in either a "My Leads" list or implicitly via the Pipeline (since an opp is created). In our design, we introduce a **My Leads page** so the salesperson can view all their claimed leads that are in early-stage (alternatively, they can see these as opportunities in Prospecting stage filtered to themselves) [50] . No one else sees these in a list except perhaps sales managers (via RLS rules).
- Nurture leads – visible in a **Nurture Leads** page for Marketing (sales won't see them) [51] .
- Disqualified leads – visible in a **Disqualified Leads** archive list for reference (Marketing and possibly Admin roles can see; sales do not need to) [51] .
- Converted leads – not visible as leads at all (they will show up as opportunities on the Pipeline Board for sales) [52] . Once converted, a lead is considered closed from the marketing perspective and will not clutter lead lists.

This mapping guarantees that at any given time, a lead record is accounted for in some view. For example, the audit identified a bug where leads triaged as Qualified were **neither in marketing's list (because they weren't New/InReview) nor in sales' list (because handover hadn't occurred)** – effectively invisible [27] . In our target state, that cannot happen: the moment a lead becomes Qualified, it immediately appears in Sales' queue (Sales Inbox) via the automatic handover, **ensuring no "vanishing leads"** [53] .

2.2 Opportunity and Account Lifecycle

Once a lead is converted or claimed by sales, it spawns an **Opportunity** (sales deal record) and is linked to an **Account** (customer record). The opportunity then goes through its own stages in the sales pipeline, and if won, contributes to the account's status as a customer. Key transitions and rules in this part of the journey:

- **Lead → Opportunity creation:** As described, this happens either when a lead is claimed (via RPC) or through the explicit convert endpoint. In both cases, the system will **find or create an Account** matching the lead's company (to avoid duplicates) [54] [55] . If an account is created, the lead's company info (name, contact) is used. The lead is then attached to that account via the opportunity.

The **Opportunity** is created with an initial Stage (by default "Prospecting") [56] or another appropriate first stage, and is owned by the salesperson who claimed/converted the lead [57]. A link `source_lead_id` on the opportunity is set for traceability [58]. The lead record gets `opportunity_id` set to maintain the connection [59]. At this point, the **Account** exists in the accounts table (with fields like company_name, etc.) and can be seen on the Accounts list, and the **Opportunity** exists for the Pipeline board.

- **Opportunity Stages:** Opportunities progress through defined stages (Prospecting, Discovery, Quote Sent, Negotiation, etc. – the exact names can be configured via the `opportunity_stage` enum) [60]. The app will allow stage changes via drag-and-drop or using a Stage change modal (which calls `/api/crm/opportunities/{id}/stage`). The backend enforces some rules on stage transitions: for example, it may require that a Quote record exists before moving to "Quote Sent", or require a Lost Reason when moving to "Closed Lost" [21] [61]. These rules are implemented in the RPC `rpc_opportunity_change_stage` [21] [62]. Each stage change is recorded in an `opportunity_stage_history` table for auditing stages.
- **Opportunity Closure (Won/Lost):** When an opportunity reaches **Closed Won**, that means the deal is successful – typically at this point the first invoice or transaction will be created in the system (outside CRM scope but integrated). A Closed Won opportunity signals that the Account is now a **Customer**. In contrast, a **Closed Lost** means the opportunity ended without success; the account might remain a prospect for future, or if no other open deals, could be considered inactive.
- **Account Status Updates:** The system will not rely on manual toggling of account status. Instead, it uses **computed statuses** for Accounts to reflect their lifecycle [63] [64]. Two dimensions are tracked:
- *Tenure Status* – e.g. "Prospect" (no closed deals yet), "New Customer" (recent first deal), "Active Customer" (ongoing), or "Winback Target" (inactive, being re-engaged) [65]. For example, when an opportunity is marked Closed Won and it's the first deal for that account, that account's tenure becomes "New Customer" (starting a 90-day window) [66].
- *Activity Status* – e.g. "Active", "Passive", "Inactive", based on last transaction date [67]. These indicate how recently the account shipped or was billed (Active might mean activity in last 30 days, Passive 31-89 days, Inactive >=90 days, etc.) [67]. The combination might be shown on the Accounts page as a badge (e.g. *"New Customer · Active"* or *"Prospect · No Transaction"*) [68]. The database can compute these via views (e.g. `v_accounts_enriched`) using invoice ledger data. **No manual editing** of these statuses – they are derived from actual data (enforcing SSOT principle).
- **Winback and Reactivation:** If an account goes inactive (no transactions for >=90 days), the system flags it for potential reactivation. Sales can then enroll that account in a **Winback Cadence** (an automated sequence of touchpoints to revive the customer) [69]. If the account does resume shipping, its status updates to "New Winback Customer" etc. This aspect ties into **cadence enrollments** (see Section 4), and ensures that even after a long gap, accounts can re-enter a sales workflow systematically.

Atomic Logic & Data Integrity: All the above transitions are implemented to be **atomic and idempotent** on the backend. For example, the claim lead RPC encapsulates lead update + account create + opportunity create + activity create in one transaction, so we don't end up with partially moved leads [37] [40]. These RPCs use an idempotency key to avoid double-processing if the request is retried [70] [71]. This means if a salesperson accidentally clicks "Claim" twice, it will still only create one opportunity and return the existing IDs on subsequent calls, rather than duplicating records [72]. Similarly, converting a target or changing an opp stage are single-call operations, not multiple client calls. This design ensures consistency in the Lead→Opportunity→Account linkage, aligning with the business rule *"transitions multi-entity always 1 atomic action (UI should not do 3–5 requests that could fail halfway)"* [73].

Side Effects & Audit Logs: Key transitions produce side effects: - When a lead is handed over, we log an entry (who handed over, when, notes) in `lead_handover_pool` and possibly an audit trail. In target state, we'll also create a system Activity (e.g. an Activity "Lead handed over to Sales" could be logged for accountability) [74] . - When a salesperson claims a lead, the auto-created "First contact" activity is a side effect, as are the creation of account/opp. We insert records in `audit_logs` or a custom `crm_audit_log` for these events (Lead Claimed, Opportunity Created, etc.) [75] [76] . This provides a historical trail for compliance and debugging, per SSOT/audit requirements. - When a lead is converted (either via claim or via convert API), we seed a series of follow-up tasks (the sales cadence) – see Section 4 on Cadence. Also, the conversion is logged with before/after data on the lead (status changed to Closed Won, opportunity_id set) [77] . - Disqualifications should log reasons in the lead record and possibly an audit event ("Lead disqualified by X on date Y with reason Z"). - Opportunity stage changes are logged in `opportunity_stage_history` (already implemented with triggers or RPC) [78] [79] . - If an opportunity is Closed Won, we might log an "Opportunity won" event and potentially trigger downstream actions (e.g. notify finance or trigger KPI calculations). If Closed Lost, ensure the lost_reason is saved for reporting (the RPC enforces it) [80] .

In summary, the **Lead-to-Account journey** ensures a seamless flow: every inbound lead is triaged (or stored for nurture) without loss [27] , every qualified lead reaches sales, every claimed lead becomes a deal with next steps scheduled, and successful deals turn into customer accounts with status tracked automatically. The process is fully **RBAC-compliant** – marketing can only move leads up to Qualified, sales can claim and work leads, neither can see the other's records until handover or conversion occurs (implemented via row-level security, see Section 6) [25] . The journey design reflects the business truth that *an Account is the single source of customer data*, a Lead is just an intake record that should not live forever on its own [81] , and an Opportunity is always tied to an Account (no free-floating opp without account) [81] . These guardrails prevent duplicate or orphan data across the CRM.

3. Folder Structure and Separation of Concerns

The project's folder structure for CRM will be refactored to enforce a clear separation of concerns between front-end pages, backend route logic, and database layer, adhering to the Next.js 16 App Router conventions and the UGC blueprint. All relevant CRM files reside under two primary directories: - **Front-End Pages:** `app/(protected)/crm/` (current state) → moving to `app/crm/` (target state). - **API Route Handlers:** `app/api/crm/`.

Current Structure Review: Under `app/(protected)/crm/` the following page components are present (as noted in the audit): - `lead-inbox/page.tsx` – Lead Inbox page (marketing leads list) [1] . - `sales-inbox/page.tsx` – Sales Inbox page (handover pool and tasks) [3] . - `pipeline/page.tsx` – Pipeline board page (opportunities kanban) [4] . - `activities/page.tsx` – Activities list page [5] . - `accounts/page.tsx` – Accounts list page [8] . - `accounts/[id]/page.tsx` – Account detail page [10] . - `targets/page.tsx` – Prospecting targets page [11] . - `imports/page.tsx` – Imports page (placeholder) [82] . - `customers/page.tsx` – (Placeholder, duplicative of Accounts) [14] . - `prospects/page.tsx` – (Placeholder, duplicative of Pipeline) [14] . - `page.tsx` (under `/crm/`) – Redirects `/crm` to `/crm/pipeline` in current code [83] .

Under `app/api/crm/`, we have route files implementing the backend logic: - `leads/route.ts` – likely handles GET (list leads with filtering by view) and POST (create lead). - `leads/[id]/triage/route.ts` –

handles triage status updates (PATCH). - `leads/[id]/handover/route.ts` – calls RPC to hand over lead to sales [84]. - `leads/[id]/claim/route.ts` – calls RPC for sales to claim a lead [85]. - `leads/[id]/convert/route.ts` – calls logic to convert a lead to opp/account [20]. - `opportunities/route.ts` – might handle listing opportunities (with views like pipeline) and creating new opp (especially quick add). - `opportunities/[id]/stage/route.ts` – calls RPC to update opportunity stage (PATCH) [21]. - `activities/route.ts` – list or create activities. - `activities/[id]/complete/route.ts` – marks an activity done and creates next (calls RPC). - `accounts/route.ts` – list/create accounts. - `accounts/[id]/route.ts` – fetch account details (possibly with related data via a join or multiple queries). - `contacts/route.ts` – create a contact. - `targets/route.ts` – list/create targets. - `targets/[id]/convert/route.ts` – convert target to lead/opportunity (calls RPC) [86]. - etc.

Restructuring and Missing Pieces: We will reorganize these to match the desired repo layout. According to the Blueprint, the repository should have:

```
app/
 ├─ crm/
 │   ├─ lead-inbox/
 │   ├─ sales-inbox/
 │   ├─ my-leads/        (new)
 │   ├─ nurture-leads/   (new, or combined with disqualified)
 │   ├─ pipeline/
 │   ├─ activities/
 │   ├─ accounts/           (list page)
 │   │   └─ [id]/
 │   ├─ targets/
 │   ├─ imports/
 │   └─ layout.tsx       (CRM-specific layout if needed)
 └─ api/
     ├─ crm/
         ├─ leads/
         │   ├─ route.ts
         │   ├─ [id]/
         │   │   ├─ triage/route.ts
         │   │   ├─ handover/route.ts
         │   │   ├─ claim/route.ts
         │   │   └─ convert/route.ts
         ├─ opportunities/
         │   ├─ route.ts
         │   └─ [id]/stage/route.ts
         ├─ activities/
         │   ├─ route.ts
         │   └─ [id]/complete/route.ts
         ├─ accounts/
         │   ├─ route.ts
         │   └─ [id]/route.ts
```

```
        ├─ contacts/route.ts
        ├─ targets/
        │   ├─ route.ts
        │   └─ [id]/convert/route.ts
        └─ imports/route.ts   (if import logic is exposed via API)
```

This layout ensures **separation of concerns**: - The **UI pages** (under `app/crm`) handle presentation and minimal client-side interactivity. They do not contain business logic or direct database calls. They fetch data by calling the appropriate API routes (or using Supabase client for reads when safe). - The **API route handlers** (under `app/api/crm`) act as the BFF (Backend-For-Frontend) layer. They contain the server-side logic: calling Supabase (via server client) to run queries or RPCs, enforcing any input validation or permission checks needed beyond RLS, and formatting responses. They use only service-role on the server side (never exposing it to the client). This prevents any direct writes from the client – all mutations go through these handlers. - The **Database** (Supabase Postgres with Row Level Security) enforces the core rules (ownership, role permissions, constraints). The API layer should not duplicate logic that the DB already guarantees (to avoid divergence), but it can perform additional checks that are user-experience-focused (e.g. returning a friendly message if required fields missing, rather than a low-level constraint error).

Violations addressed: In the current implementation, a few separation-of-concern issues were noted: - Some role-based access checks were done in both the UI and the DB, but not consistently. For example, the UI only enabled triage for Marketing roles, but the RLS policy on leads allowed any lead creator to update their lead [87] . In target state, we will **align the API and RLS**: if only Marketing should triage, then either RLS or the RPC will enforce that (and the UI will simply reflect it). We will remove redundant or contradictory checks. For instance, we might tighten the leads UPDATE policy so sales users cannot change triage_status, and ensure the triage API endpoint also restricts by role to be safe [88] . - The **service role key usage**: The client-side should never directly use the service role. In the current code, if any Supabase calls on the client were using elevated rights, that's a serious concern. The target state mandates all such usage be removed – the service key will only be used in Next.js server functions (Route Handlers) and never sent to the browser [89] . All client data retrieval should happen with the anon role + RLS, or via server which uses service key internally. - **Missing files & handlers:** The audit found placeholders where functionality was incomplete – e.g. no handler when clicking "Add Contact" (button did nothing) [10] , no page to create a new Account (the "Add Account" link was miswired) [9] , and imports page empty. In target state, we treat these as gaps to fill, not optional enhancements. Concretely: - Implement a form (modal or dedicated page) for **creating a new Account**. This could be a modal on `/crm/accounts` that calls POST `/api/crm/accounts` with the required fields (company_name, PIC info, etc.) [90] . After creation, it should redirect to the Account 360 page of the new account or show a success message. - Implement the **Add Contact** form on the Account 360 page. It will collect first name, etc., and call POST `/api/crm/contacts` to insert the contact (with the current account_id). On success, the UI will refresh the contacts list on that page to include the new contact [91] . - Implement the **Imports page** to handle uploading CSVs and initiating import jobs. Likely, this involves an upload component that sends the file to an API route (or directly to storage, then triggers a server process). The `import_batches` table and a function to process rows (perhaps calling an RPC for each row to create targets or leads) will be utilized. A simple approach: the Imports page calls a new `/api/crm/imports` route (to be created) which reads the CSV, creates an entry in import_batches, and enqueues row inserts (this could be synchronous if small or via a background worker if large). The page will then show the status (processing/completed) and any errors. - Remove or repurpose any duplicate pages: the `/crm/customers` page (which was just static text) will be removed, and navigation links that pointed there (if

any) will point to `/crm/accounts` instead [14]. The `/crm/prospects` page (also static text about pipeline rules) will be removed to avoid confusion with "Prospecting Targets" and "Prospects (Opportunities)". We will ensure the sidebar or breadcrumb doesn't list these nonexistent pages. - Provide a proper **Lead creation** mechanism for internal users. Currently, leads are mostly created via external forms or the seed data. The audit noted no UI to add a Lead manually except some quick add for sales [92]. In target state, we can allow Marketing to create a lead (e.g. an "Add Lead" button on Lead Inbox that opens a form for a new inbound lead). This would call POST `/api/crm/leads`. The backend will generate a new lead_id (with proper prefix via DB trigger) and default triage_status = New [93]. This ensures marketers can log leads from events or lists without going through the import if it's one-off. - Ensure every CTA has a corresponding handler: E.g., *Add Opportunity* from Account page should open the Quick Add prospect flow or at least redirect to Pipeline with that account pre-selected. Ideally, implement a modal on Account page to create a new Opportunity under that account (owner = current user) calling POST `/api/crm/opportunities`. - **Dead code / placeholders:** remove any leftover `TODO` comments or unused components. The target state should be a clean baseline – for instance, if a `prospects-kanban.tsx` component exists but we choose not to use it (preferring a unified pipeline component), we remove it. All menu items must be functional or not present.

Separation of Folders by Domain: Notice in the blueprint structure, each top-level module has its own folder (crm, kpi, ticketing, etc.) and its own API sub-routes [94]. This avoids intermixing code from different domains. In the current repo, the CRM code is largely isolated, which is good. We will continue to ensure CRM pages only import CRM-specific components (or generic UI components). Any cross-module interactions (like KPI charts showing number of new leads) should go through well-defined boundaries (likely via database views or minimal API calls), not by directly importing code from one module into another.

By following this structure: - Development is simplified (knowing exactly where to add a new lead view or endpoint). - There is a **single canonical route** for each action (preventing, say, both an RPC and a REST endpoint doing the same thing in parallel) [16]. - Roles and RLS are respected by design – the folder `(protected)` can be dropped as long as our **auth strategy** uses Supabase SSR session and RLS to protect data [95]. (We won't use Next.js middleware to gate routes; instead, a server-side check in a parent layout can redirect unauthenticated users to login, per blueprint guidelines [95]).

Finally, we will document the folder layout in the repo README so new developers understand where everything lives (as reflected above). The end goal is a clean, **maintainable codebase** where front-end and back-end concerns are properly separated, and all expected features are accounted for by actual code (no stubs).

4. Automation & Cadence System

Automation is a critical part of the CRM, ensuring that once a lead or opportunity enters the system, a structured sequence of follow-ups (a **cadence**) is initiated without manual effort. The target-state CRM will leverage dedicated **Cadence** tables to manage these sequences and ensure that activities (tasks) are created, assigned, and scheduled properly. We identify the relevant tables and outline how the cadence system operates:

Cadence Tables: The database schema includes three tables supporting sales cadences [96] [97] [98]: - `cadences` – Defines an automation sequence template (e.g. "Standard 2-Week Follow-Up Cadence"). Fields: `cadence_id` (PK), name, description, is_active, owner_user_id (who owns/created the cadence template), etc [96]. - `cadence_steps` – Defines the individual steps of a cadence template. Each step has a foreign key to a cadence, a step number (order), an activity type (call, email, etc.), a subject template, optional description template, and a `delay_days` value [97] [99]. The `delay_days` indicates how many days after the start (or after the previous step) this action should be scheduled. - `cadence_enrollments` – Links a specific record (Account, Contact, Opportunity, or Target) to a cadence, indicating that the record is currently in that automated sequence [98]. Fields include `cadence_id` (which template), optional `account_id`, `contact_id`, `opportunity_id`, or `target_id` depending on what is being enrolled, the current step number, status of the enrollment (Active, Paused, Completed, etc.), who enrolled it, and timestamps [98] [100].

These tables were created to support a flexible automation system. In the current code, however, the usage of them is minimal; the logic is partly hardcoded. The target state will **fully utilize these tables** for consistency and configurability: - The standard sequence of follow-up activities that was previously hardcoded in the `CADENCE_TEMPLATE` array in code [101] will instead be stored as a default Cadence (say, "Default Lead Follow-Up"). For example, Cadence #1 might be "New Lead 14-day Cadence" with 5 steps (Day 0 Call, Day 2 Email, Day 5 Call, Day 10 Email, Day 14 Call – matching the template in code) [101]. - The application can seed this default cadence in the database (via migrations or initial data), so it can be managed via the app if needed (admins could adjust delays or types without code changes).

Seeding Cadence on Opportunity Creation: Whenever a new opportunity is created from a lead (whether by claim or convert), the system will automatically enroll that opportunity (and its account) into the default cadence: - In the **Claim Lead** RPC, after creating the opportunity, the code currently directly inserts one activity ("First contact" call for today) [41] [42]. In the target state, we will expand this such that: - It creates a new row in `cadence_enrollments` linking the opportunity (and account) to the chosen cadence template, with current_step = 1. - It generates the first activity from the cadence_steps definition (which should be a Call on Day 0). This first step activity is essentially the same "First contact" call that was being inserted, but now we record its `cadence_enrollment_id` and `cadence_step_number` in the activities table [102] [103]. The Activities table has columns for `cadence_enrollment_id` and `cadence_step_number` specifically for this purpose [102]. - Subsequent steps are not all inserted at once in this model. Instead, the system can insert them one at a time or on a schedule: *Option A:* Insert all cadence steps upfront as planned activities (this is what the original code did in the Convert Lead handler – it mapped through the template and inserted 5 activities at once with future due dates [104] [105]). This guarantees they're all visible on the planner. If we do this via cadence_steps, it means reading all steps for that cadence and creating corresponding activity records with appropriate due_date offsets. We would fill in each activity's `cadence_step_number` and link to the enrollment. *Option B (preferred):* Insert only the first activity initially. Then, as each activity is completed or its due date passes, automatically create the next one. This is more dynamic and can adjust if the schedule changes (e.g. if step 2 is supposed to be 2 days later but step 1 was done late, we might reschedule accordingly). - We will implement **triggered creation of next activities** using the `rpc_activity_complete_and_next` function (or a variation). The system already has `rpc_activity_complete_and_next` which marks an activity "Done" and can auto-create a follow-up [106] [107]. We can enhance this to work with cadences: if the completed activity has a `cadence_enrollment_id` and there is a next step in that cadence, the function will create the next activity. Specifically, it can increment the `current_step` on the cadence_enrollments record and insert a new activity with that next step's details (type, subject template, due date = today + delay_days of next step).

This keeps the cadence moving. If an activity was not marked done by its due date, we could still create the next one on schedule – perhaps via a daily job that checks Active enrollments. - The **owner handoff** aspect: when these activities are created, they need to be assigned to the correct owner. In our case, since an opportunity is owned by a salesperson, all cadence activities for it should have `owner_user_id = opportunity.owner_user_id` (the salesperson). The current implementation in code already sets owner to the profile user in context (which is correct) [108] . We will ensure that if marketing enrolls something (e.g. a target into a cadence), activities either get assigned to whoever is responsible (owner_user_id field in enrollment can help). In summary, cadence tasks follow the record's owner so that the accountable person always sees them in their Activities list. - Step delay logic is dictated by `delay_days` in cadence_steps. The code in Convert Lead used `dayOffset` to compute due dates [104] . In target state, we rely on the stored `delay_days`. For example, cadence step 2 might have delay_days = 2, meaning 2 days after enrollment (or after previous step) – if we generated all upfront, we simply add 2 days to today for its due_date. If generating one by one, we take the completed step's completion date and add the next step's delay to schedule the next activity.

- In the **Convert Lead** API (for a lead already owned by sales, perhaps), the current code seeds 5 activities immediately [104] [105] . We will refactor that to follow the same cadence enrollment process:
  - Create/find Account, create Opportunity (as it does).
  - Instead of mapping through a hardcoded template, enroll the new opportunity in the default cadence. Then either insert all cadence steps as activities or just the first, as per our chosen approach. (Even if we initially still insert all, we'll do it by reading from cadence_steps table for maintainability.)
  - The convert function already records how many activities were created and returns that in the response [109] . This will continue to work (just coming from DB-defined template).

Cadence Ownership and Customization: The design allows multiple cadence templates (maybe one for new leads, one for winback, etc.). In the database, each cadence has an `owner_user_id` which could denote who created it, but functionally all active cadences are available to relevant users (e.g. sales team). We can expose a UI for admins to create/edit cadence templates in the future. For now, we will focus on using the default ones: - **Lead Follow-Up Cadence** – triggered on new opportunity creation (post-claim or convert). - **Winback Cadence** – can be triggered when an account becomes inactive. This would involve enrolling an Account (or a dummy opportunity) into a special cadence designed for reactivation. Sales could initiate this from the account page ("Enroll in Winback Cadence"), which would create tasks like "Call customer – re-engage" etc. The blueprint explicitly mentions a reactivation workflow where sales enrolls an inactive account into a winback cadence [69] .

Automatic Step Creation & Delay Logic: We will implement logic such that: - When an opportunity is enrolled, an initial set of tasks is created with appropriate due dates spaced out as defined. If tasks are all created upfront, their due_date is set to today + delay_days, etc. [104] [105] . If tasks are created stepwise, then each completion triggers next. - If a salesperson doesn't complete a task by the due date, the system could either: - leave it as "Planned/Overdue" and still create the next one on schedule (this could lead to overlapping tasks, not ideal), - or hold off creating the next until the previous is done (but then you lose the strict schedule).

A balanced approach: we keep the tasks scheduled at certain calendar dates regardless, so the salesperson sees them all (like a calendar of upcoming touches). If one is not done in time, it's marked overdue but the next still appears when its time comes. The **Sales Inbox** shows overdue tasks anyway, alerting the

salesperson [3]. This was partially implemented: the Sales Inbox page fetches activities with `view=inbox` which presumably filters to overdue tasks for that user [3]. We will maintain that function. Therefore, creating all tasks upfront with due dates might be simplest and aligns with having visibility into the schedule. We just need to ensure that if a lead is unclaimed or an opp is closed early, we handle those leftover planned activities (e.g. cancel them if opp is Closed Won before some steps occur). - Each activity has an `activity_type` and `subject`. The cadence_steps templates provide a "subject_template" – e.g. "Follow-up email with proposal". When creating the actual activity record, we substitute placeholders if needed (like company name) to generate the final subject ("Follow-up email with proposal: [CompanyXYZ]") [110]. The current code already did string concatenation for subjects [111]. We will formalize that using the template fields. - **Owner handoff edge-case:** Suppose Marketing had a cadence for Nurture leads (if implemented) – those tasks would have marketing as owner. When a lead eventually gets qualified and handed to sales, any open nurture tasks should probably be canceled or transferred. Our system will likely treat Nurture leads separately (maybe a different cadence system for marketing follow-ups). For now, we note that as a consideration: if a lead moves from Nurture to Qualified, we should remove it from the nurture cadence (mark that cadence_enrollment as completed/stopped). The act of handover effectively cancels marketing's cadence in favor of the sales cadence.

Cadence and Activity Logging: Each created activity from a cadence will reference the cadence. We can easily query how many steps have been completed via the enrollment record (current_step updates, or status becomes "Completed" when current_step exceeds total steps). Sales managers could report on adherence to cadence (e.g. did the rep complete all steps? If an enrollment is paused or stopped, that could indicate a change in plan, etc.).

Summary of Required Rules: - **Auto-Enrollment:** Whenever an opportunity is created from a lead (either by claim or convert), auto-enroll it into the default follow-up cadence (no user action needed). Similarly, allow enrolling any account or target into a cadence via explicit user action (e.g. a button "Start Cadence" for winback). - **Auto Activity Creation:** Immediately create the first step's Activity (or all steps) for the new enrollment. Ensure each Activity has `owner_user_id` set to the appropriate user (sales owner for opps, etc.), status = "Planned", and due_date computed. - **Step Delay Logic:** Each step's due date = enrollment start date + cumulative delay_days (if all upfront) [112]. The delay schedule is defined in the cadence_steps. For example, if step1 delay=0, step2 delay=2, step3 delay=5... relative to start: due_dates at D0, D+2, D+5... - **Owner Handoff in Activities:** If ownership of the record changes mid-cadence (unlikely for opps, but e.g. if account is reassigned, or if lead claimed by someone else?), we might want to reassign future activities to the new owner. To keep it simple, we can say once a cadence is enrolled, the tasks stick with the owner at time of enrollment. Account reassignments should perhaps trigger re-enrollment or manual adjustment of tasks. - **Pausing/Stopping:** Users should have the option to pause or stop a cadence enrollment (e.g. if the customer responded or plans changed). The cadence_enrollments status field can be updated to "Paused" or "Stopped". In paused, no new tasks should generate until resumed; in stopped or completed, the cadence ends. We will provide a way (maybe in UI later) to stop enrollment (for now, completing all steps or converting/winning might implicitly complete it). - **Concurrent Cadences:** A record generally should not be in two active cadences of the same type. Our design doesn't explicitly forbid multiple enrollments (the table allows it), but as a rule, enrolling an opportunity in the default cadence while it's already Active would be redundant. We can enforce unique (opportunity_id, status='Active') perhaps. - **Cadence for Targets vs Leads:** The **Prospecting Targets** module might use cadences for outreach before they become leads. For example, a sales rep could enroll a Target in an "Outbound Prospecting Cadence" (calls/emails to try to engage them). Our cadence tables already allow `target_id` in enrollments [98]. In target state, we plan to utilize that: when a target is imported, the system could auto-enroll it into a cadence if desired (or the user

can manually do it). If the target responds and is Qualified (status → qualified), the rep converts it to a Lead/Opportunity, at which point that cadence might end (target converted) and a new one (the sales follow-up for the opportunity) begins. We should ensure converting a target marks its cadence_enrollment as completed to avoid hanging tasks on a now-converted target. - **System Cron for Cadence:** If we choose not to create all tasks upfront, we might need a scheduled job (cron or Supabase scheduled function) to check daily for cadence_enrollments that are Active and create the next activity when its due. However, an easier route is to create tasks upfront. Given our limited scope, we'll likely implement upfront creation of all steps to guarantee tasks show up (with the downside that if some steps become irrelevant, they'd need manual deletion or ignoring).

Example Flow (Lead Claim): Marketing qualifies a lead, it auto-handover to sales pool. Salesperson clicks **"Claim"**: - RPC `sales_claim_lead` runs: finds lead, creates account if needed, creates opportunity, assigns lead to sales [38] [37]. - After opportunity creation, system enrolls that opportunity in cadence #1 (Default Lead Follow-Up). Inserts cadence_enrollment row (enrolled_by = sales user). - Fetch cadence_steps for cadence #1: - Insert Activity for step 1 (e.g. Call today). - Insert Activity for step 2 (Email 2 days later), step 3, etc. All these have `related_opportunity_id` and `related_account_id` set, owner = sales user, status = Planned. - All activities also store cadence_enrollment_id and step_number. - Return response to client with success, plus perhaps the new opportunity_id and first activity_id (the RPC already returns opp_id, etc.) [113]. - Sales Inbox UI navigates the user to either the new Opportunity in Pipeline or at least confirms the claim. In target state, we will improve UX to **navigate directly to the Opportunity 360 or Pipeline** when a lead is claimed, so the sales user can start working it immediately (with a toast or link) [114]. - The Activities page for that sales user now shows a "Call [Company] (Planned, due today)" along with future tasks dated accordingly. If the user completes the call, they mark it done (triggering `rpc_activity_complete_and_next` via the /activities/[id]/complete route) which could, if we choose to, create the next activity (though if we pre-created all, this RPC might simply mark complete and perhaps offer to schedule an additional follow-up if `p_create_next` flag is true – this RPC might be more relevant if tasks weren't already created). - If tasks are all pre-created, marking one as done won't auto-create another (since it's already there), unless we decide to always create one more beyond the last (some cadences might be open-ended until stopped). But default cadence is finite steps.

Activity Auto-creation on Stage Change: Another automation outside cadences: When an opportunity stage changes, we might want to generate tasks. For example, moving to "Quote Sent" might automatically generate an Activity "Follow up on quote in 3 days". The current RPC for stage change checks that a Quote exists but doesn't create tasks [21]. In the target state, we could extend automation such that certain stage transitions prompt adding a follow-up activity (or even enrolling in a micro-cadence for that stage). This is a nice-to-have and could be defined in a config (for now we note it as a potential future addition).

In summary, the **Automation & Cadence System** in target state will ensure that: - No new lead/opportunity falls through without scheduled follow-ups. The mantra is every active item must always have a next action and owner [7]. - The cadence templates provide a standardized workflow which can be adjusted without code changes. - Salespeople are guided through consistent touchpoints, improving conversion rates and providing metrics (we can track completion rates of cadence tasks as part of KPI). - The implementation leverages the existing cadence DB schema (cadences, steps, enrollments) rather than ad-hoc hardcoding, aligning with the SSOT principle (single source of truth for process logic).

5. Mapping of Tabs to Queries, Views, and Filters

Each CRM page and tab corresponds to a specific data query (often hitting either a Postgres **view** or using query parameters on a REST endpoint) and enforces certain filters (usually based on record status/owner). The table below maps each page or tab to its underlying query logic and highlights any potential overlaps or gaps in the filtering (to avoid collisions or orphan records):

- **Lead Inbox (Marketing Leads)** – **Query:** GET `/api/crm/leads?view=inbox`. This API will fetch from the `leads` table (or a dedicated SQL view `v_lead_inbox` if defined) all leads where `triage_status IN ('New','In Review')` and that are owned by or created by the marketing team user (RLS ensures a marketing user only sees their leads or team's leads) [1] . In the current code, the `view=inbox` parameter is handled by filtering statuses on the server side [27] . We will implement a Postgres **view** called `v_lead_inbox` to encapsulate `WHERE triage_status IN ('New','In Review')` for clarity. This ensures **New** and **In Review** leads show up here and only here. No lead with another status will appear in this view.
- *Filters:* This page might further allow filtering by assigned marketing user (if that becomes relevant) or date, but by default it shows all new leads visible to that marketer. Orphan check: Once a lead's triage_status moves out of these values, it disappears from this list (which is correct, and other views catch it).
- *Evidence:* The audit confirms that the Lead Inbox query was only showing New/In Review due to a filter, which caused Qualified leads to drop off [27] [1] . We fix that by moving Qualified leads to another view (Sales pool) immediately.
- **Qualified Leads (Marketing)** – (This was missing in current system). If we did not auto-handover, we'd need a view here for triage_status = Qualified without handover. But in target design we choose auto-handover, so there is no persistent "qualified but not handed over" state. Thus, **no separate Qualified Leads page** is needed; those records go straight to Sales.
- **Handover Pool (Sales Inbox)** – **Query:** GET `/api/crm/leads?view=handover_pool`. This returns leads handed over but not yet claimed. Implementation: either a view or a join on the `lead_handover_pool` table where `claimed_by IS NULL`. We can create a Postgres view `v_sales_inbox` that selects from `lead_handover_pool` join leads to get lead details where claimed_by is null [115] . Alternatively, since the handover pool essentially holds lead_id with that flag, we could simply filter leads by `handover_eligible = true AND sales_owner_user_id IS NULL` (since unclaimed leads have no owner). However, using the pool table is more definitive. This view shows the **unassigned leads that sales can claim** [50] .
- Additionally, the Sales Inbox page in UI also pulls **Overdue Activities** for the logged-in salesperson. That likely is a separate call: GET `/api/crm/activities?view=overdue` or similar. If implemented, `view=overdue` would filter `activities` where status = 'Planned' and due_date < today and owner_user_id = current user (via RLS or a filter param). In the audit, they mention the sales inbox uses two data sources – leads handover and activities inbox [3] . We will maintain that, possibly combining them in one request if we create a special endpoint, but likely the frontend just makes two calls.
- *Filters:* The handover pool view inherently filters out any lead that's been claimed (claimed_by not null) [116] . Also RLS will ensure only sales team (or admin) can select from it [117] [118] . No collisions: a lead in this pool is by definition not visible in Lead Inbox anymore (triage_status likely "Handed Over" which isn't included in inbox filter).
- *Orphan check:* Once claimed, the lead leaves this view – where does it go? It should appear in My Leads/Pipeline for the owner.

- **My Leads (Claimed by Me)** – **Query:** GET `/api/crm/leads?view=my_claimed` (or implement as part of leads view). This is a new view to introduce. It will fetch leads where `sales_owner_user_id = auth.uid()` and `triage_status = 'Handed Over'` (or generally leads that have been claimed by the user and not converted yet). However, since claiming an unqualified lead automatically creates an opp and effectively moves it to pipeline, we might instead surface these via the opportunities list. There are two possible implementations:
- **Option 1:** Use the leads table: define a view `v_my_leads` = leads where sales_owner_user_id = current user AND handover_eligible = true AND (maybe status not converted). Those would be leads currently assigned to the salesperson that haven't been converted (which is essentially a narrow window, since conversion ideally happens immediately or soon after claim).
- **Option 2:** Use opportunities: once claimed, an opp exists in Prospecting stage with owner = user. We could just instruct sales to look at Pipeline (filtered by My deals).
- We will include My Leads page as a safety net listing of leads with triage_status 'Handed Over' assigned to the user. This ensures any lead record in limbo can be seen. In practice, if our claim flow immediately sets lead.status = "Contacted" and they might convert it later via `/convert`, there is a period where the lead is being worked but not closed. My Leads would cover that scenario.
- *Filters:* By owner = current user (enforced in query or RLS). It excludes any that have `opportunity_id` but we might still show it until fully closed.
- *Collisions:* An item in My Leads would also appear in Pipeline (as an opp in Prospecting stage). This is a potential duplication. To minimize confusion, we might design the UI such that My Leads is more of a checklist for the salesperson ("these are leads you claimed recently, ensure to convert them"). Or we might decide to drop My Leads entirely and rely on Pipeline. But since the prompt explicitly mentions it, we include it with understanding it might overlap.
- *Orphan check:* If a salesperson claims a lead but doesn't convert it (maybe waiting for more info), it lives here so it's not lost. The pipeline also shows it (so it's doubly covered).
- **Nurture Leads** – **Query:** GET `/api/crm/leads?view=nurture`. We will create a view that selects leads with `triage_status = 'Nurture'` (and possibly also any with an explicit Nurture owner if that concept exists). These are accessible to marketing team only. This page was missing before, causing nurture leads to effectively vanish from UI. Now they will be listed in a dedicated tab/page so marketing can continue to follow up periodically [24] [51].
- *Filters:* triage_status = Nurture. Optionally, could filter by who is responsible for nurturing (in some orgs, inside sales might handle it). For now, all nurture leads appear for marketing.
- *No collisions:* No other page shows nurture leads (they're not New/InReview, not in sales pool, etc.).
- **Disqualified Leads** – **Query:** GET `/api/crm/leads?view=disqualified`. A view selecting `triage_status = 'Disqualified'`. This is like an archive list. Only accessible to appropriate roles (marketing or admins). It ensures even disqualified leads are accounted for somewhere (for reporting or potential undo).
- We might combine Nurture and Disqualified into one "Closed Leads" page with sub-tabs, or separate as needed. In any case, both statuses will be covered.
- *No overlap:* These are final states not shown elsewhere.
- **Pipeline (All Opportunities)** – **Query:** GET `/api/crm/opportunities?view=pipeline`. This returns all open opportunities (stage not in Closed Won/Lost). The current code likely filters in the SQL or API for active stages [4]. We will implement a Postgres view `v_pipeline_active` = opportunities where stage not in ('Closed Won','Closed Lost','On Hold' maybe). The API can simply select from that view for `view=pipeline`. It might also join account name, etc., or the front-end might call an extended view that has account and owner names.

- *Filters:* stage filter as above. Could also accept query params like `?owner={id}` or `?account={id}` to further filter. We will support an `owner` param so that on the Pipeline page the user can toggle "My Deals" which effectively adds `.eq('owner_user_id', currentUserId)` on the API call.
- *Overdue view:* We plan a `view=overdue` which returns opportunities where `next_step_due_date < current_date` and stage is still open and owner = current user. This can be used for a "My Overdue" tab or to highlight overdue deals.
- *Collisions:* An opportunity could technically appear in multiple filters (e.g. an opp that is overdue and also in active pipeline; but that's fine as those are just different lenses on the same list). No duplication in UI simultaneously.
- *Orphan check:* Opportunities that are Closed (Won/Lost) are not shown here – but they could be shown in an archive or in the Account's history. We might later add a Pipeline filter for "All/Closed" or just rely on Accounts page to see closed opps for that account. For now, closed opps are hidden from main pipeline (which is expected).
- **Pipeline (My Opportunities)** – **Query:** GET `/api/crm/opportunities?view=pipeline&owner=me` (or a specific endpoint `/api/crm/opportunities?view=my_pipeline`). This would filter the pipeline to only those where `owner_user_id = auth.uid()`. We can implement it either in the same route (just filter on query if owner param is provided) or as part of RLS by logging in as a sales user (though RLS as defined allows marketing to see all opps too which is interesting [119], but likely sales might want to filter anyway). The UI "My Deals" tab will use this. It prevents one salesperson from seeing others' pipelines unless they deliberately switch (if we allow).
- This is essentially a subset of pipeline view, so no unique orphan issues.
- **Activities (Planner)** – **Query:** GET `/api/crm/activities?view=planner`. In current code, it likely returns all "Planned" activities for the user grouped by date [5]. Possibly implemented with a filter `status = 'Planned' AND owner_user_id = auth.uid()`, and maybe `due_date >= current_date` (to show upcoming and today). The UI might fetch all and do grouping client-side. We could use a view or just query directly with supabase client and let RLS filter by user. According to RLS, marketing can see all activities (owner or not) [118], which might mean in a marketing planner view they see not only their tasks but possibly tasks of leads they own? This could be adjusted if needed.
- *Filters:* status = Planned by default (the UI might have a toggle to see Done or All). Also perhaps excluding canceled. The audit noted no link back to related lead/opp from an activity [5] – we will add such links in UI (e.g. clicking an activity can open the related opportunity or account).
- *Overdue:* The Sales Inbox page specifically pulled overdue tasks separately, but on the Activities page, overdue tasks would just appear under past dates or marked overdue. We can highlight them.
- *No collisions:* Activities are either Planned (in this list) or Done (which could be in a history if implemented). No activity should appear in two statuses at once.
- *Orphan check:* If an activity is tied to a record the user no longer owns (say account transferred), RLS might hide it from the old owner and show it to new. That's handled by policies likely using owner_user_id. Completed activities not shown can still exist in DB for logs.
- **Accounts (Customers List)** – **Query:** GET `/api/crm/accounts?view=enriched`. This likely maps to a view `v_accounts_enriched` which the migrations mention [120] [121]. That view probably joins accounts with some aggregated info (like number of open opportunities, perhaps tenure and activity status calculations). The accounts page uses it to display badges and counts. We will ensure that view includes the computed `tenure_status` and `activity_status` for each account (these could be computed via SQL from invoice_ledger as per the rules) [66] [67]. If not already, we will

create such a view. Alternatively, the front-end might retrieve accounts and compute statuses client-side, but better to do in DB for consistency.

- *Filters:* The accounts page provides a search by company_name or domain (the audit says search works) [8] . We can implement this by allowing a query param (e.g. `?q=Acme`) that our API uses in a `.ilike('company_name', '%Acme%')` or similar. Also, perhaps filter by owner (to see one's own accounts vs all). Current RLS allows sales to select accounts if they are admin or sales (no owner restriction to view) [122] , which implies all sales can see all accounts – maybe acceptable since accounts aren't secret, or maybe they intended only sales owner and marketing can view (this is unclear; we might refine RLS if needed).
- *No collisions/orphans:* It's just listing accounts. Inactive ones are still accounts and will show up (with statuses indicating inactive). The only accounts that might not show are perhaps those marked is_active=false (if we implement archiving), but likely they still show unless filtered out intentionally.
- **Account 360 (Account Details)** – **Query:** GET `/api/crm/accounts/[id]`. This should return a comprehensive dataset: the account record, plus maybe an array of related contacts, open opportunities, recent activities, etc. Possibly the route handler does multiple selects or one big select with embedded sub-objects (Supabase allows queries like `select=*, opportunities(*), contacts(*), activities(*)` with appropriate foreign keys). We have to be mindful of RLS: likely the route uses service role (since it's server side) and then filters data manually to only include what the user should see. Or we craft supabase RPC to get account 360 info if needed. In target state, this endpoint will ensure:
- Contacts: list contacts where account_id = given (all visible to any user who can view the account).
- Opportunities: maybe open opps for that account (or all opp history, maybe separate by stage).
- Recent Activities: could filter last N activities (both done and planned) for that account.
- We will implement what is feasible and needed; at minimum contacts and open opps.
- *Filters:* The account itself might require the user to have access (if RLS restricts accounts by owner, but as noted currently any sales/marketing can see all accounts by policy [123]).
- *No overlaps:* This is a unique view of one account's data.
- **Targets (Prospecting Targets)** – **Query:** GET `/api/crm/targets` with various params. The UI shows counts by status, implying either:
- The front-end calls GET without a filter to get all targets then counts statuses, or
- There are pre-defined views or aggregate endpoints.

We likely implement: - `view=all` or default: all targets the user owns (or all if admin). RLS on prospecting_targets already restricts select to owner or unassigned or admin [124] , meaning a salesperson sees the targets they imported or those not yet assigned. - We add a query param `status=<value>` to filter by a specific target status (`new_target, contacted, engaged, qualified, dropped, converted`). The front-end can use this to only fetch one category at a time if desired. - We might also provide an aggregate (or simply count on client side). - Search param `q` to match company or contact name, etc. - *Filters:* status filter as above. By default, exclude status = converted or dropped in the main working list (converted targets have been turned into leads/opps, so those might be hidden unless looking at archives – we could show them in an archive tab in Targets page if needed). - *No collisions:* A target that is converted is no longer in the active target list; it has an account_id or lead_id linking to CRM proper. We should perhaps exclude converted ones from default view and show them in a "Converted Targets" sub-view for reference (to avoid confusion). Also "dropped" (not interested/invalid) targets can be hidden or in a separate filter (similar to disqualified leads). - *Orphan check:* When a target status changes to converted, we populate `converted_to_account_id` (and possibly converted_to_lead_id) on it [125] . That target won't be worked further, and since it's in Converted status we handle it as essentially archived (the new lead/

opportunity takes over). It will not appear in open target list, but an admin could query converted ones if needed. So no issue as long as we don't lose the link (we have the account_id reference stored).

Filter Collisions and Missing States: The above mapping fixes the known missing views (Qualified leads, Nurture, Disqualified were missing – we added those). **Filter collision** would mean a record accidentally appears in two lists at once: - In our design, a lead could theoretically appear in both My Leads and Pipeline (as discussed) once claimed. This is a conscious overlap for usability – My Leads is essentially a subset view. It's not a logic error, but we should avoid confusion (perhaps by clarifying My Leads shows "your leads in pipeline"). - Another potential collision: if a lead's triage_status isn't updated properly. For example, suppose a bug left a lead as triage_status = In Review but also handover_eligible=true. That lead might erroneously appear in both Lead Inbox and Sales Inbox. Our backend should prevent that state from happening (handover RPC sets triage_status to 'Handed Over' [35]). We will enforce consistency: once handover_eligible = true, triage_status should be considered 'Handed Over' (we might explicitly set that enum value if we extend lead_triage_status to include "Handed Over" state, which migration did [126]). This way, marketing inbox query (New/InReview) will exclude it, and it solely appears in sales view. - If a salesperson converts a lead via the convert API without using claim (maybe for a lead already assigned to them), that lead's triage_status might not have gone through Handed Over. The convert API currently doesn't update triage_status at all (it just sets lead.status Closed Won) [59]. In target state, we should set triage_status to something (maybe leave as Handed Over or Qualified) and probably `handover_eligible=false` because it's now handled. In any case, after conversion the lead is out of lists. - **Orphan states:** We particularly ensure: - Qualified leads – now immediately handed over, so none remain orphan. - Nurture leads – have a page. - Disqualified leads – have a page. - Claimed leads – either in My Leads or appear in Pipeline for that user, so they are visible somewhere to sales. - Converted leads – moved to Pipeline (as opp) and removed from Leads listing. - Converted targets – become leads/opps, so they leave the target list (status Converted) which we filter out from active view. They can be seen in accounts or pipeline now.

The invariant is that every record is either in one of these views or is truly closed out. We also log anything that is closed for audit.

The queries above will be implemented in the API route handlers using either direct `.from(...).select()` calls with filters or calling RPCs that encapsulate the view logic. We will preserve usage of RLS where possible (especially for simple filters by status and owner, since RLS already restricts by role and ownership). For example, listing leads for marketing might simply do `.eq('triage_status', 'New').or('triage_status.eq.In Review')` and rely on RLS to limit to that marketer's dept. Sales listing pool leads will query the pool table (which via policy only returns anything to sales roles) [127].

In summary, the tab-to-query mapping ensures **clear source of data for each UI element**, and resolves the prior system's blind spots. We have introduced new views for every lead status category and validated that a lead/opportunity in any given state will appear in the correct one: - New/In Review → Lead Inbox, - Qualified (immediately moves to pool, so transient), - Handover (unclaimed) → Sales Inbox (Leads section), - Claimed (not closed) → My Leads (and Pipeline), - Converted (open opp) → Pipeline, - Nurture → Nurture Leads list, - Disqualified → Disqualified list, - Closed opp (won/lost) → not in pipeline, but visible under Account's history or separate report.

These mappings will be documented for the team and reflected in the code (perhaps via constants or comments in the API handlers to avoid any confusion when modifying filters).

6. Database and Schema Validation

We have audited the `database_ugc-bcp.txt` schema to ensure all relevant tables, foreign keys, constraints, and enum types are properly defined for the CRM module. Overall, the schema aligns well with the target design, but we identified a few inconsistencies and areas to tighten to fully support the new flows:

Core Entities Schema: - **Leads Table (`public.leads`):** The leads table contains all necessary fields for the new workflow: - Keys: `lead_id` as primary key (text). There is a trigger function (not shown in snippet) presumably to auto-generate IDs with a prefix like "LEAD" (similar to accounts) – if not present, we should add one to ensure consistent ID format [128]. - Triage fields: `triage_status` enum of type `lead_triage_status` with values ('New','In Review','Qualified','Nurture','Disqualified','Handed Over') [126]. This column was added in the CRM rebuild migration and has a default 'New' [129]. **Issue:** It is not marked NOT NULL in the schema (likely an oversight). We will alter this column to `NOT NULL` because every lead should always have a valid triage_status [130]. Also, we will ensure the value 'Handed Over' is used when a lead enters the pool (the migration enumerated it, and the RPC sets it) [35]. - Legacy status: `status` of type `lead_status` enum (['New','Contacted','Qualified','Proposal',...]) [131]. This was the old pipeline stage field. In the new model, we are relying on triage_status for marketing progress and converting leads to opportunities for sales stages. The code, however, still uses `status` at least in converting leads (setting it to 'Closed Won') [132] and in claiming (setting to 'Contacted') [40]. This dual usage is confusing. **Recommendation:** We will repurpose the `status` field purely for legacy compatibility or eventually remove it. For now, perhaps treat `lead.status` as a mirror of important events: e.g., when a lead is claimed, set status to "Contacted", when disqualified set "Disqualified", when converted set "Closed Won" – basically marking the outcome. But the authoritative workflow state is in triage_status (plus the presence of opportunity_id). To reduce confusion, developers should prefer triage_status, and we will document that `status` is deprecated for leads (except for possibly one final state flag like 'converted'). - Foreign keys: `sales_owner_user_id` (UUID) references profiles(user_id) [133]. This is set when sales claims. **Check:** The foreign key exists (`leads_sales_owner_user_id_fkey`) [133]. Good. `customer_id` references accounts(account_id) [134] – this is a legacy field from when leads had a direct customer link. Now that we have opportunity->account, this might not be needed. We will likely drop or ignore `customer_id` going forward (convert processes already create account and link via opportunity). If it remains, it should at least be kept in sync (the convert lead function currently inserts an account and could set leads.customer_id = that account_id if we wanted). The audit noted this legacy usage as well. We'll plan to remove it in a future migration to avoid confusion (and use `opportunity_id` for linking). - `opportunity_id` references opportunities(opportunity_id) [135]. FK exists (leads_opportunity_id_fkey) [136]. This gets set when converted/claimed. This ensures referential integrity – e.g., cannot set an opp ID that doesn't exist. (We should confirm that if an opportunity is deleted, perhaps if lead is converted in error, the lead.opportunity_id is set to null via ON DELETE action or prevented by app logic. In general, we won't delete opps). - Other fields like `created_by`, `created_at`, `updated_at` have proper defaults and FKs (created_by -> profiles) [133]. - Fields for timestamps: `qualified_at`, `disqualified_at` – these were added to record when those events happen [137]. We need to ensure our code sets these. E.g., when triage_status changes to Qualified, set qualified_at = now(); when Disqualified, set disqualified_at = now() and disqualified_reason accordingly [138] [47]. Currently, the RPC handover sets updated_at but not qualified_at; we will update the triage API to handle these timestamps. - `handover_eligible` boolean – default false [129]. This is toggled by RPC when handing over. Good. - **Indexes:** We see index on leads.triage_status (idx_leads_triage) was created [139], and on dedupe_key. We should also add an index on `sales_owner_user_id` because queries like "my leads"

or filtering by owner will be common. The migration 13 mentions adding constraints and indexes, perhaps it added one (check supabase migration 13).

- **Lead Handover Pool:** Table `lead_handover_pool` has proper structure:
- `lead_id` with UNIQUE constraint [140] (so a lead can only be in pool once),
- refs lead_id -> leads(lead_id) not explicitly shown, but we should add a foreign key for data integrity. (It might be missing; we should create `FOREIGN KEY (lead_id) REFERENCES leads(lead_id) ON DELETE CASCADE` so if a lead is deleted or converted we can remove pool entry).
- Fields for handed_over_by, claimed_by (both UUID refs profiles), timestamps, notes, priority, etc. These all look correct. Index exists for unclaimed leads (claimed_by IS NULL) [115] for fast query of the pool list.
- We will ensure that when a lead is claimed and an opportunity created, we remove the pool entry or mark it claimed (the RPC does mark claimed_by and claimed_at) [141] . Claimed entries are effectively inactive.

- Potential improvement: If a lead is unclaimed for too long (maybe we had an expires_at field), but it's optional. There is an expires_at column in the schema (perhaps to auto-release leads if not claimed) [142] , though not heavily used now.

- **Opportunities Table:**

- `opportunity_id` text PK, with trigger to auto-generate (likely similar to accounts, we should ensure one exists for opps, e.g. prefix "OPP"). If missing, implement a `trg_opportunity_id()` function.
- Fields include account_id (FK to accounts), owner_user_id (FK to profiles), stage (ENUM opportunity_stage) with default 'Prospecting' [60] [143] , next_step text and next_step_due_date (both NOT NULL – ensuring every opp has a next action, which is great for our invariant) [144] .
- Source references: `source_lead_id` and `source_target_id` fields are present [145] , but notably **no foreign key constraints on these**. The schema did not define FKs for source_lead_id -> leads or source_target_id -> prospecting_targets. **Recommendation:** Add foreign keys for data integrity (with ON DELETE SET NULL perhaps). This ensures if a lead were deleted (which we typically don't do, but just in case) the opp's reference doesn't break. More importantly, it enforces that any source_lead_id set actually exists in leads table, which is good.
- `lost_reason`, `competitor`, `notes` fields exist for additional info on opp.
- Timestamps and created_by are present. Possibly an `updated_at` (yes, and closed_at, outcome fields) [146] .
- **Check constraints:** None explicitly, but RPC ensures lost_reason if Closed Lost etc. [80] . We might consider adding a DB CHECK that if stage = 'Closed Lost' then lost_reason is not null – but since stage changes happen through controlled API, this might be okay without.

- Indices: presumably there's an index on owner_user_id (if not, add one for filtering pipeline by owner). There are indexes on status fields in seed.

- **Accounts Table:**

- `account_id` text PK. Trigger exists (trg_account_id) to auto-gen IDs like "ACCT20230101XYZ" [128] .

- Fields: company_name, pic_name/phone/email (contact person info), address, city, etc., owner_user_id (FK to profiles) [147] [148] . All good.
- Unique indexes on domain and npwp (tax id) to avoid duplicates [149] .
- Relationship: leads.customer_id points here (legacy). Opportunities account_id points here (enforced with FK) [150] . Contacts account_id points here (FK).
- `is_active` boolean, default true – could be used to manually mark an account inactive, but we'll rely on activity_status for real status. Still, good to have.
- `tags` text[] for any tagging, fine.

- No major issues. We might ensure every opportunity must have an account (it's NOT NULL in opp table).

- **Contacts Table:**

- Standard fields (first_name, etc.), account_id FK (ON DELETE CASCADE, so if account deleted, contacts gone, which makes sense) [151] .
- created_by FK, etc.
- Unique index on (account_id, email) and (account_id, phone) to avoid duplicate contacts at same account [152] .
- Contact ID has trigger to generate (trg_contact_id) with prefix "CONT" [153] .

- This looks solid. We will just need to implement the missing UI to create contacts.

- **Prospecting Targets Table (`prospecting_targets`):**

- `target_id` text PK, likely with trigger similar to others (e.g. prefix "TGT" or so). Migration might have added next_prefixed_id for targets as well.
- Fields: company_name, contact_name/email/phone, notes, etc., similar to leads but simpler.
- `status` is an enum `target_status` which after migration (16_target_status_ssot.sql) now has values ('new_target','contacted','engaged','qualified','dropped','converted') [154] [155] . The migration updated existing records and renamed this column from old status [156] [157] . Now `status` is NOT NULL with default 'new_target' [158] . Good.
- It also introduced a `target_status_transitions` table and a function to validate transitions [159] [160] [161] . And an RPC `rpc_target_update_status` to change target status atomically with validation and idempotency [162] [163] . We should ensure the frontend uses this RPC via `/api/crm/targets/{id}/status` or similar when implementing target status updates (like marking dropped or engaged).
- FKs: `owner_user_id` (sales owner of the target) FK to profiles [164] . `created_by` FK to profiles [165] . Good.
- `dedupe_key` UNIQUE (so that the same company/contact imported twice can be detected) [166] . In seed, dedupe_key might be something like lower(company_name) or email, etc.
- `converted_to_lead_id` and `converted_to_account_id` – these store references to what the target became when converted. **Issue:** No FK constraints on these in schema [166] . We should add:
    - FOREIGN KEY (converted_to_lead_id) REFERENCES leads(lead_id) ON DELETE SET NULL,
    - FOREIGN KEY (converted_to_account_id) REFERENCES accounts(account_id) ON DELETE SET NULL. This ensures those IDs, if present, correspond to real records. In our code, target conversion doesn't create a lead record (it goes straight to opportunity) [22] [23] , so

converted_to_lead_id likely remains NULL (we convert to opp and set converted_to_account_id). We do update target.status to 'converted' and set converted_to_account_id [125]. Possibly we should set converted_to_lead_id to some placeholder or maybe to the opportunity ID? But it's meant for lead. Perhaps originally they thought conversion would create a lead. Since it doesn't, we might leave converted_to_lead_id null and rely on converted_to_account_id and maybe add converted_to_opportunity_id (not present, but source_target_id in opp serves that).

- `converted_at` timestamp was added in migration for when target converted [167]. The RPC currently does not set converted_at explicitly (except indirectly via updated_at) – we should modify RPC to set converted_at = now() when status goes to converted (the migration did backfill it) [168].
- `drop_reason` and `dropped_at` added for when dropped [167]. The RPC validate function requires reason if dropping requires it (some transitions require reason marked by requires_reason=true in transitions table) [169] [47]. We should ensure the API passes a reason when needed. The DB function itself returns error if missing [47].
- The RLS on targets: select allowed if (admin OR owner or unowned) [124], insert if (admin or sales) [170], update if (admin or owner) [171]. This means marketing by default cannot see or create targets – which is okay since targets are a sales thing.

- All in all, target schema is robust after the SSOT alignment migration.

- **Activities Table:**

- `activity_id` text PK. A trigger likely exists to generate with prefix e.g. "ACT". If not, add one.
- Fields: `activity_type` enum (activity_type_v2, which includes Call, Email, etc) [172], `status` enum (Planned/Done/Cancelled) [173], subject, description, outcome, due_date, etc.
- The references for linking: `related_account_id`, `related_contact_id`, `related_opportunity_id`, `related_lead_id`, `related_target_id` – all FKs to respective tables (with ON DELETE as appropriate) [174]. Yes, FKs exist for each as per schema [174]. This means an activity can link to any of those entities to indicate context. In our flows:
  - Cadence tasks created on opp will have related_opportunity_id (and account).
  - A task to follow up a lead (if before conversion) might use related_lead_id.
  - Outreach to a target not yet a lead would use related_target_id.
  - Activities on an account (like a general call to the customer not tied to a specific deal) could use related_account_id (and maybe a primary contact).
- Fields: `owner_user_id` (who is responsible to do it) and `created_by` (who logged it) both FK to profiles [175]. Good.
- `cadence_enrollment_id` and `cadence_step_number` – these link the activity to a cadence. They are present (bigint and int). We should add FKs for these:
  - `cadence_enrollment_id` -> cadence_enrollments(enrollment_id) ON DELETE SET NULL (or CASCADE? Setting null might be better if an enrollment is deleted perhaps if cadence canceled).
  - There might not be an obvious need to delete cadence enrollments, but if we did, we'd orphan these fields. Better to keep enrollment until all activities done.
  - `cadence_step_number` is just an integer, likely no direct FK (since not linking to cadence_steps, though we could theoretically link (cadence_id, step_number) but that's complex). We will treat it as informational.

- Indexes: should index owner_user_id for filtering tasks by user (RLS covers but index improves sort by user maybe for admin), and due_date possibly (if querying overdue).

- RLS on activities: select if admin or owner or marketing [118], insert if admin or sales or marketing [176], update if admin or owner [177]. This is okay, though letting marketing see all sales activities might be debated – but perhaps they want marketing to have visibility on follow-up tasks for leads they generated. We could refine if needed (maybe marketing can see activities related to leads they created? But currently it's broad).

- **Cadence Tables (Cadences, Steps, Enrollments):**

- `cadences`: `cadence_id` serial PK [178], name, description, is_active, owner_user_id (who owns template), created_by, timestamps [178]. There are RLS policies allowing select for all (sales, marketing) [123] but insert/update only for admins [179]. This means normal users can't create new cadences, which is fine for now (only admin can define templates).
- `cadence_steps`: `step_id` serial PK, cadence_id FK (on delete cascade) [99], step_number, activity_type (enum), subject_template, description_template, delay_days, created_at [99]. Unique index on (cadence_id, step_number) to avoid duplicate numbering [180]. Looks good. RLS similar to cadences (select all, insert admin only) [181].
- `cadence_enrollments`: `enrollment_id` bigserial PK, cadence_id FK, account_id FK, contact_id FK (on delete set null), opportunity_id FK, target_id FK, current_step, status (text, not enum, default 'Active') [100], enrolled_by FK, enrolled_at, completed_at, stopped_at [100]. Indexes on cadence_id, account_id, opportunity_id, status for fast filtering [182]. RLS: select if admin or if enrolled_by = auth.uid (so users see the cadences they started; this might limit managers from seeing team's cadences, which we might revisit) [127], insert if admin or sales (so sales can start a cadence) [183], update if admin or enrolled_by (so only the person who enrolled can pause/stop? Possibly should allow the owner of the record too – might adjust policy to allow opp.owner to update too). We should refine that if needed so that if a sales manager enrolled something, the record owner can still mark it complete. But this is detail.
- One improvement: `status` is a text field with values like 'Active', 'Paused', 'Completed', 'Stopped' (as code comment suggests) [184]. We might turn that into an enum for consistency and to prevent typos. Not critical, but a small schema improvement (add type cadence_status).

- All FKs present except maybe none on contact (set null) and target (set null) to allow cleaning those without deleting enrollment. That's acceptable.

- **Other Tables:** The schema also includes other modules (tickets, etc.) but focusing on CRM, the above covers it. One more:

- `crm_idempotency` table: stores idempotency keys and cached responses [185]. Primary key on idempotency_key. This is used by RPCs (they call `check_idempotency()` and `store_idempotency()` inside) [70] [71]. It's working behind the scenes to ensure repeated calls don't double-process. It has an `expires_at` to purge old entries [186]. The presence of this is a positive aspect noted by audit [187]. We just ensure all our new/modified RPCs use it properly (they seem to).

Schema vs App Misalignments: - The **Lead vs Opportunity status** handling: As discussed, having both lead.triage_status and lead.status can confuse developers. We will clarify: *triage_status* is the primary state

for a lead's marketing journey. *status* (lead_status enum) is legacy; after our overhaul it might only be set to 'Closed Won' when converted, etc., but otherwise we could ignore it. Perhaps we use it to track intermediate sales progress on a lead if we weren't immediately converting (e.g., they set 'Contacted'). This duality should be resolved in code and possibly in DB (maybe repurpose lead.status enum to simply have values New vs Converted vs Disqualified to mirror triage? But it's currently a different set including sales pipeline stages which are irrelevant after we have opp). **Action:** Document that `lead.triage_status` drives UI, and eventually plan to drop `lead.status` or map it 1-to-1 with triage for simplicity. In any case, ensure no logic is left that only uses lead.status for things that should use triage_status. - The **customer_id in leads**: As noted, it points to accounts now (after renaming customers->accounts). It's basically the account that was created when the lead converted in older flow. We should consider removing this column to avoid confusion, since we have opportunity linking to account. If some old code populates it, it's duplicate info. We'll mark it for deprecation. - The **prospect_id in leads**: It references the old archived prospects table (now `_archived_prospects`). This is legacy from pre-rebuild (lead.prospect_id was used to link to the old prospect if I recall). We can drop this as well in future since it's no longer used (we have new prospecting_targets now). It's currently in schema with FK to archived table [188]. Removing it cleans up the model (less confusion between "prospect" and "target"). - **Enum usage updates:** The target state migration updated target_status values to new scheme [154] and dropped old type. That's good. The lead_triage_status enum includes "Handed Over" which we use. One thing: do we need a "Converted" value in lead_triage_status? It currently does *not* have "Converted" (since they maybe didn't want to mark leads as converted in triage status) [126]. Instead, they rely on lead.status = 'Converted' or `opportunity_id` not null. We might consider adding 'Converted' to lead_triage_status to explicitly mark that state (for completeness, and then we could mark triage_status = Converted when a lead is converted, rather than leaving it at Qualified or Handed Over). This wasn't done, possibly because a converted lead is effectively out of the lead lifecycle. But adding it could make queries simpler (e.g. lead list queries could explicitly exclude triage_status = Converted to avoid any edge cases). However, since converted leads have opportunity_id and status Closed Won, it's obvious. We might leave it as is to avoid confusion with the target 'Converted'. We'll just ensure converted leads are filtered out by other means. - **Foreign Key additions:** as identified: add FKs for opportunities.source_lead_id, opportunities.source_target_id, targets.converted_to_lead_id, targets.converted_to_account_id, activities.cadence_enrollment_id. These reinforce integrity. - **Constraints:** Consider adding check constraints where applicable: - e.g. `CHECK (triage_status != 'Qualified' OR handover_eligible = true)` if we auto-handover (but since we *will* auto-set handover_eligible, might not need a constraint, but could be nice to enforce that if triage_status=Qualified and not handed over, it's invalid). - Or simply ensure via code that doesn't happen. - Another: `CHECK (status != 'Closed Lost' OR lost_reason IS NOT NULL)` on opportunities (but our RPC covers it). - `CHECK (status != 'Dropped' OR drop_reason IS NOT NULL)` on targets (the transitions function covers it already at application level).

- **Row Level Security alignment:**
- We should double-check that RLS policies allow exactly the access we expect. For example, for leads:
  - Current select policy likely allows any marketing user to see leads they created or all leads? Actually, I recall an RLS function like `crm_leads_select_allowed` might exist. The migration 10 snippet in audit mention "crm_leads_update_allowed permits any creator to update their lead; API restricts triage to marketing only" [88]. That suggests:
  - SELECT policy on leads might allow sales to see leads they own as well (makes sense, sales should see leads assigned to them).

- We likely have:

```
CREATE POLICY leads_select ON leads USING (
  auth.role in (Director, admin, marketing, sales) AND
  (
    role = marketing -> maybe condition on triage_status in
(New,InReview,Nurture,Disqualified) and created_by = auth.uid
    OR role = sales -> see leads where sales_owner_user_id = auth.uid
OR maybe leads in pool? (pool is separate though)
  )
);
```

  This is speculative; the actual policy definitions might be in the SQL file which we can retrieve. We may want to refine them:
  - Marketing should not see leads once handed over (maybe RLS could hide leads with triage_status 'Handed Over' or a sales_owner set, from marketing, if we want a hard separation – or we rely on UI not to fetch them and trust people).
  - Sales should not see leads that aren't handed to them (they currently see pool via separate table, and see ones where sales_owner_user_id = their id).
  - Admin can see all.
- If any mismatches: e.g. audit said RLS allowed any lead creator to update their lead (so if a marketing created a lead, even after it's assigned to sales, they might technically still update it via service or if RLS allowed because created_by = them). That is not desired after handover. We should tighten leads_update policy: once a lead has a sales_owner, only that sales (or admin) should update, not the original creator. Possibly implement that logic in the policy or at least ensure the API forbids marketing editing after that point. Given possible complexity, simplest is to handle in API: e.g. triage endpoint can't be called if lead already handed over (the RPC probably errors if lead not found in their query). We will clarify roles allowed for each action.

- **Audit Logs:** The schema has `audit_logs` and possibly a custom `crm_audit_log`. The convert lead code attempts to insert into `crm_audit_log` and ignores errors if not exist [76]. We should confirm if `crm_audit_log` table exists (if not, remove that code or create the table). It might be they planned a separate log table. We do have the general audit_logs. The audit recommended to **log every important change** (patch lead, etc.). We should enforce via triggers or in our route handlers:

- Could add a trigger on leads, opportunities to insert into audit_logs on updates. But easier: we already have some manual inserts in code (like convert does one, stage change might not though).
- Possibly implement a generalized trigger in DB or ensure each API call writes to audit_logs (the generic audit_logs table is present with before_data, after_data fields) [189]. We may add triggers for triage_status changes etc. However, given time, we might rely on explicit logs in API for now.
- The presence of audit_logs with before/after suggests some triggers might already exist or devs planned to use Supabase's built-in replication to capture changes. But we will utilize it as needed.

Schema vs Business Logic Alignment: - The schema seems to support all "business truths": - **No duplicate accounts** via unique indices [149]. - **No orphan foreign keys** (we will add missing ones). - **No invalid state**

transitions – targets have a transitions table & function [161], opportunities have RPC checks, leads should have similar enforcement in API. We may implement a similar state transition validation for leads (like not allowing going from New directly to Disqualified without marking reason, etc.). If not using a table, at least do checks in the triage API. - **Single source of truth**: All these tables are in one Postgres schema (public), and all views draw from them. We aren't duplicating data across multiple locations. E.g., when a lead converts, we don't copy data to opportunity beyond needed fields, we reference via foreign keys. That ensures consistency – e.g., the account created is used by opp and lead references it, rather than having separate customer info on lead and opp.

- **Performance considerations:** The schema has appropriate indexes on most search/filter fields (status fields, foreign keys, etc.). We will add missing indexes as noted (sales_owner on leads, owner on opps).
- Also ensure composite index if needed: e.g. for pipeline Overdue, we might index on owner_user_id + next_step_due_date for fast querying of "my overdue opps".
- The RLS policies might need some indexes to avoid sequential scans on large data (Supabase might handle some via indexes on relevant columns with auth.uid, etc.). For instance, leads policies filtering by auth.uid in some places – index on created_by or sales_owner could help.

In conclusion, after these adjustments, the database schema will fully enforce the CRM invariants: - **Foreign keys and cascade rules** will prevent orphan records (e.g., no opportunity without account, no contact without account, etc.). - **Enum and check constraints** will enforce data validity (no invalid statuses, required reasons on drops via logic). - **RLS policies** will enforce visibility at the lowest level (even if UI had a bug, a marketing user querying sales leads would get nothing, etc.), as a defense-in-depth [25]. - The schema is in sync with how the app uses it in target state (fields like next_step on opp used, lead triage fields used, etc.), with minimal unused legacy fields (the few identified will be cleaned up).

We will perform a final **schema-validation** test by simulating typical flows: - Creating a lead -> verify triage_status default = 'New', all FKs valid. - Qualifying lead -> ensure handover pool entry requires lead has contact etc. (RPC checks) [31]. - Claiming lead -> verify account/opportunity creation, FKs link properly, lead.opportunity_id set, pool entry updated. - Converting lead -> ensure account/opportunity created, lead.status updated, no constraint issues. - Converting target -> ensure target.status goes to 'converted', account and opp created, and FKs (source_target_id on opp, converted_to_account_id on target) set correctly. - Running these operations with RLS on (using different role tokens) to ensure no unauthorized writes or reads. - Checking that attempting invalid transitions (e.g. target new_target -> qualified directly without engaged) returns errors (which the validate function does) [190].

By validating these scenarios, we confirm the schema supports the target state flows without any hidden misalignment.

7. Fix Recommendations and Enhancements

Finally, we outline concrete recommendations to fix misuses or gaps in the current system and implement the target state design fully. These include database changes, RPC/API modifications, and moving certain

client-only logic to be enforced on the server/database side. All fixes are assigned as actionable changes (no TODOs left unaddressed):

- **Automatically Handover Qualified Leads:** Eliminate the bug where a lead could be marked Qualified and then disappear because it wasn't handed over. Update the lead triage API (`PATCH /api/crm/leads/{id}/triage`) such that when `triage_status` is set to "Qualified", the backend **immediately calls** `rpc_lead_handover_to_sales_pool` (or the route does equivalent) to mark it handed over [32]. This will set `handover_eligible=true`, insert into pool, and update triage_status to "Handed Over" in one go [34] [35]. Acceptance: after a marketer clicks "Qualify", the lead should show up in Sales Inbox instantly without requiring any manual pool action [191]. This fix ensures no "qualified leads vanish" scenario [27].
- **Implement Missing UI Pages/Actions:** No more placeholder buttons or dead-end links:
- **Add Account:** The "Add Account" button on Accounts page should open a working form (either as a page `/crm/accounts/new` or a modal) that collects account details and calls `POST /api/crm/accounts` to create a new account [90]. The server will create the account (using the auto ID trigger) and return the `account_id`. The UI then navigates to that account's detail page. Only authorized roles (Marketing Manager, Sales, etc.) can create accounts (enforced by RLS policy accounts_insert which currently allows sales team and admins) [122].
- **Add Contact:** On Account 360 page, the "Add Contact" button now launches a form to create a contact on that account [91]. It calls `POST /api/crm/contacts` with at least first_name and contact info. The server inserts the contact (the contacts_insert policy allows sales team which is fine as account owner will be sales) [192] [193]. After success, the contact list is refreshed. No more dummy button.
- **Import Targets:** Build out the Imports page. Allow uploading a CSV of target prospects. On submission, the UI calls a new API (e.g. `POST /api/crm/targets/import`) or possibly the same `/api/crm/targets` with a batch flag. The server will parse the file (in memory or by storing then reading) and for each row create a prospecting_target (ensuring unique dedupe_key). It will also create an entry in `import_batches` table logging the batch (source, number of rows, how many inserted). The UI should show feedback – e.g. "50 targets imported successfully" or any errors (if duplicates were skipped, etc.). Also implement listing of past import batches on this page using `import_batches` table.
- **Remove Redundant Pages:** Delete or hide the unused `/crm/customers` and `/crm/prospects` pages. Update navigation links: any "Customers" menu should now point to `/crm/accounts` (or be renamed "Accounts"). Ensure no code references the old pages (they were mostly placeholder text anyway). This avoids user confusion and makes Accounts the single page for customer list.
- **Lead Creation UI:** Optionally (P2 priority), add an "Add Lead" for marketing. This could be a small form to input lead details (company, contact, source, etc.) and calls `POST /api/crm/leads`. This helps during events when marketing wants to manually enter a lead. The API already supports POST leads (with dedupe and ID generation) [92]. This will complement the import (for one-off leads).
- **Opportunity Quick Add & Stage Modal:** These exist but ensure they function: Quick Add Prospect on Pipeline calls `rpc_sales_quick_add_prospect` (which likely creates lead+account+opp in one go) [194] – verify and adjust if needed to also seed cadence. Stage change modal calls the stage route, which uses RPC to enforce rules (works). We keep these flows and improve error handling.
- **Navigation after Actions:** Improve user flow by redirecting or updating context:
    - After **claiming a lead**, the app should navigate the user to the new Opportunity detail or pipeline board focusing on that opp. Currently, claiming just removes it from Sales Inbox but leaves user there. In target state, we can show a toast: "Lead claimed and Opportunity

created – view opportunity »" that links to Pipeline or a dedicated opp page [114]. Ideally, we implement an Opportunity detail page (or reuse account page highlighting opp) for a smooth transition.
    - After **converting a target**, currently the UI stays on targets page with no indication. We will have it either automatically open the newly created opportunity (or lead) or at least notify "Target converted to Lead/Opportunity". Perhaps route to Pipeline (since opp is created). This prevents the converted record from feeling like it vanished (earlier, after conversion the target disappeared from target list and user had to manually go find the resulting lead).

- After **closing an opportunity (Won)**, perhaps prompt to view account or next steps (not critical for functionality, but UX improvement).

- **UI Query Adjustments (No Orphan States):** Update front-end queries to use the new views/tabs:

- Marketing Lead Inbox page: ensure it requests `view=inbox` (New & InReview) – already did, but now also handle Qualified leads properly. Actually, with auto-handover, marketing won't have qualified leads lingering. If we decide to list any leads that got qualified but not yet claimed (shouldn't happen long), we could introduce a sub-tab "Qualified (pending sales)" showing those with handover_eligible=false and triage_status=Qualified. But since we auto-handover, likely this will always be empty. So maybe not needed.
- Sales Inbox page: ensure it calls both leads?view=handover_pool and activities?view=overdue. If previously it wasn't filtering by owner for activities, ensure now that overdue query is owner-specific. Possibly provide the API `GET /activities?view=inbox` which returns activities due today or overdue for auth.uid (the audit snippet suggests something like that was done [3]). We define that clearly: *overdue = all Planned activities where due_date < today and owner_user_id = current user*.
- My Leads page (new): implement API handling `view=my_claimed` as mentioned. The UI will call it and display leads (or simple message if none). This prevents claimed leads from not being visible in any list if user doesn't notice the opp in pipeline.
- Nurture/Disqualified pages: add these pages and fetch accordingly. Marketing can now click a "Nurture Leads" tab to see those leads and continue engagement or eventually re-triage them. Similarly a "Disqualified" archive tab for reference.
- Pipeline page: add filter UI for "My Deals" (toggle or tab) which calls opportunities?owner=me. Also implement an "Overdue" highlight – possibly an icon or separate view. According to spec, we add `view=my_overdue` in pipeline to list user's opps where next_step_due_date past. This can be accessible via a filter or a widget on dashboard.
- Accounts page: fix the Add Account link, implement search bar properly (likely already did).
- Account 360 page: after implementing Add Contact, also consider adding a filter on activities shown (maybe show last 5 activities done and next 5 upcoming for that account). If any sub-sections empty, maybe hide or say "No contacts yet" rather than leaving blank. Fill any dummy text with real data or remove it.
- Targets page: hook up the status filters to API calls with status param. Make sure converting or dropping a target triggers a refetch or UI update (so the converted target disappears from "Engaged" list and the counts update). Provide feedback for convert (like navigate to pipeline as said).

- Imports page: once implemented, test it thoroughly with a sample CSV.

- **Workflow Optimizations:** Ensure the transitions and handoffs are smooth:

- When a lead is claimed, as mentioned, directly navigate to the new opp (saves user time) [114] .
- When converting a target, we might consider skipping creating a lead altogether (we currently don't create a lead, going straight to opp and account). That is fine and saves a step. But to avoid confusion, perhaps rename `rpc_target_convert_to_lead` to `..._to_opportunity` (or simply document that it creates an opp). The user experience after converting a target should treat it like a new opportunity in pipeline. Possibly auto-assign it a stage (Prospecting) and an owner (the one who converted). We did that. Also auto-enroll it in cadence (the RPC currently doesn't do cadence seeding – we should add that: after creating opp in target convert, call the same cadence enrollment logic to seed follow-ups).
- Implement a **winback workflow** (if not immediate, at least plan for it): e.g., an automated job that scans for accounts with last invoice >90 days to flag them. Since this might be beyond current scope, we note it as a future feature. For now, sales can manually identify and use cadences.
- Validate the **Lead claiming race-condition**: The RPC uses SELECT … FOR UPDATE SKIP LOCKED to ensure two sales can't claim same lead [195] – that's good. We should test with concurrency.

- **Idempotency keys usage**: Make sure every route that writes uses a unique idempotency key. E.g., in the handover route, we generate `handover-{lead_id}-{uuid}` each call [84] , which is fine since you normally wouldn't double-click that fast. In claim: they include user_id and random to avoid collisions [196] . That's good. Convert lead or quick add prospect likely similarly. No changes needed but ensure new code uses `store_idempotency` consistently to not break the pattern.

- **Error Handling & UX Feedback:** The audit pointed out the UI wasn't showing errors well (e.g., silent failures) [197] . We will:

- Add toast notifications for any API error responses. The API already returns structured errors (e.g., 400 with message). We need to catch those in the front-end and display them. For example, if triage RPC returns "Lead missing required fields" [31] , surface that to the user so they know why they can't qualify yet.
- Form validation on client: e.g., require certain fields not empty (don't let user submit Add Lead form if phone/email blank, etc.), to reduce error round-trips.
- Ensure all `/api/crm/...` routes consistently return `apiSuccess` or `apiErrors` so front-end can rely on `success: false` presence or HTTP status. The code does that already. We just unify any inconsistent ones.
- For RPC outcomes that return already_converted or no available lead, handle those gracefully (the claim route returns conflict if none or if already claimed) [198] – the UI should then show "Sorry, that lead was just claimed by someone else."

- Remove console.log dev prints in production code (some route code prints errors to console but ideally we'd handle or at least not expose).

- **Security & Data Integrity Enhancements:**

- **Remove service key from client**: Double-check .env usage; ensure `NEXT_PUBLIC_SUPABASE_ANON_KEY` is used on client and the `SUPABASE_SERVICE_ROLE_KEY` is only in server env (which Next route handlers will have). No client calls should use service role. If any legacy code did (perhaps in some supabase client initialization in old code with service key), purge that. The blueprint insisted on not exposing service key ever [89] .

- **Align RLS and API checks**: As discussed, tighten the leads policies so marketing cannot update leads after handover. Possibly update leads_update policy to `(auth.uid() = sales_owner_user_id) OR (created_by = auth.uid() AND sales_owner_user_id IS NULL)` for example. And leads_select policy to `(role=Marketing -> see triage_status in (New,InReview,Nurture,Disqualified) and created_by = auth.uid()) OR (role=Sales -> see leads where sales_owner_user_id = auth.uid() OR (handover_eligible = true and claimed_by is null?) )` – well, since unclaimed leads are in separate table, sales likely only sees ones they claimed. We might not need to show sales leads that are unclaimed (they get them via pool). We will implement accordingly with help from the security team perhaps.
- **Indexing**: Add database indexes for any fields now frequently queried:
  - `leads.sales_owner_user_id` (for My Leads queries).
  - `opportunities.owner_user_id` (for pipeline filtering by owner).
  - `opportunities.next_step_due_date` (for overdue queries; maybe combined with owner).
  - `activities.owner_user_id` and `activities.status` (for listing tasks by user and status).
  - Confirm existing index on `lead_handover_pool.claimed_by` (there is one for unclaimed search) [115].
  - `cadence_enrollments.status` has an index [182] which is good for checking Active ones.
- **Audit Logging:** Implement triggers or use existing ones to ensure critical changes are recorded:
  - For example, attach a trigger on leads table AFTER UPDATE that if triage_status, sales_owner_user_id, or status changes, insert a row into audit_logs with before/after JSON. Similarly for opp stage changes, and for target status changes. We have an audit_logs table with before_data/after_data columns. We can use the generic approach (maybe an extension or a TG function to log any table changes that we specify). If not enough time, at least ensure our route handlers call audit insert. Right now:
  - Lead triage route – should call audit after change (currently, not sure if it does).
  - Handover RPC – could log an action "LEAD_HANDED_OVER".
  - Claim RPC – logs result via idempotency but not explicitly to audit_logs; we should add at least audit of lead assignment (changed owner).
  - Convert lead route – it logs to audit_logs and crm_audit_log as seen [75] [76].
  - Stage change RPC – doesn't explicitly log in audit_logs aside from stage history table. We might consider adding a log "stage changed from X to Y".
  - We may decide to treat `opportunity_stage_history` and `lead_handover_pool` as the audit trail for those events (since they record who and when).
  - For completeness, we'll add: when a lead is disqualified or nurture, maybe just rely on triage status change logged if audit trigger on leads.
  - These logs will help debugging and compliance.
- **Refactor Route Structure (Folder Layout):** As described in section 3, restructure files and paths to remove the `(protected)` group (unless needed for Next 13 auth pattern, but blueprint says avoid middleware, so likely fine). All references in code to those paths should be updated (the sidebar links etc.). After moving, test that navigation still works (Next's file system routing might handle redirects or might require adjusting imports).

- Also ensure that the `app/crm/layout.tsx` wraps all pages with the CRM-specific sidebar and topbar as needed. The current structure may have a global layout and the `(protected)` layout. We likely will have a main layout that checks auth and includes the sidebar. That might be currently `(protected)/layout.tsx`. We can rename that to something like `app/dashboardLayout` for all auth pages, or keep it as `app/(auth)/layout` etc. This is technical, but the end result should be one consistent layout for all internal pages, with the sidebar menu showing only allowed items (as blueprint required) [26] . We will verify the menu generation code (probably in `components/layout/sidebar.tsx`) uses the user's role to filter menus. If not, implement that: e.g., if user.role is Sales, hide KPI or Marketing modules accordingly.

- Remove any duplicate route definitions (like if there was both /crm/index and redirect, etc., unify to one). The blueprint wants one canonical path per page [16] .

- **Naming Consistency:** Address any inconsistent naming or overly bloated schema parts:

- Use "Account" universally instead of sometimes "Customer". We've effectively done that by removing /customers page. In code and text, ensure we stick to one term.
- Use "Opportunity" instead of "Prospect" for pipeline deals (the UI placeholder was titled "prospects" perhaps, change it to "Opportunities"). The backend table is opportunities, which is correct. In some UI text or variables maybe they said prospect. We will normalize to "Opportunity".
- "Prospecting Target" vs "Target": in UI just call them "Targets" or "Prospecting Targets" consistently. The repo uses prospecting_targets table and sometimes calls them prospects. To avoid confusion with sales "prospects" (meaning opportunities), we will consistently call them "Targets" in front-end labels.
- Field misuse: `lead.next_step` and `lead.due_date` – these were legacy fields for lead's next action. Now that we have activities, these might not be used. Indeed, when a lead becomes an opp, the opp has next_step fields. We should repurpose or stop using lead.next_step. Possibly, if a lead is in nurture, marketing might set a next_step like "Follow up in June". But better to create an Activity for that. So likely we won't use lead.next_step at all in target state. We could leave it blank or ignore. If so, we might remove it in a schema cleanup later. For now, mark it as deprecated in comments.
- Remove fields not used: leads.prospect_id (legacy, drop in migration), leads.customer_id (legacy, drop when sure not used). Also, if any columns like leads.route or timeline not used, consider if needed – if marketing fill them and want them in opp, we should map them. Actually, lead.route and lead.service_code are mapped when creating opp (we do pass lead.route, and service_code goes into service_codes array on opp) [57] [199] . Good, that uses those fields. So keep those.

- If any naming in code is inconsistent (like function names), refactor lightly (not crucial to user output though).

- **Shifting Logic to DB where possible:** We want critical business rules enforced in the database layer to guarantee consistency:

- The best example is target status transitions – implemented with a DB function and table, which is great [161] . For leads, we might do similar or at least ensure the RPC or server code enforces rules (like requiring contact for Qualified, reason for Disqualified). Possibly implement a `validate_lead_triage` plpgsql function to check transitions and call it in an RPC or in the route handler.

- Use database defaults and triggers to reduce client responsibility:
    - e.g., Use the ID generation triggers (already in place for accounts, contacts, etc.) so API doesn't need to generate IDs manually (some code used `uuidv4()` for lead id? Actually they didn't for lead; they likely rely on a trigger or use Supabase's `gen_random_uuid()` default if not prefixed. Checking leads: no trigger found, they might use some other approach or just accept a provided id. In our API, when creating lead, if no id provided, we might want to generate one. Could call the next_id function explicitly or use an extension. We should implement a `trg_lead_id` similar to accounts for consistency (prefix "LEAD").
    - Another: Instead of front-end computing due dates for cadence tasks, store delay_days in DB and let the conversion RPC compute due_date – which it did in code for convert lead using JavaScript date math [200]. We could push that logic into a SQL function if it were done via RPC as well. But since conversion is in Node code, that's fine. Just ensure it's correct.
- **Move lead conversion fully to DB?** We have a Node route doing it, but we also have `rpc_sales_claim_lead` in DB doing a very similar thing (account, opp, etc.). It might be cleaner to unify these paths. Possibly deprecate the Node convert and just have sales use claim or another RPC. But since claim assumes pool, and convert is for direct assignment case, it's fine to have both.

- If any UI-only validations (like email format) could be double-checked in DB (we can add a CHECK or use an extension, but not high priority).

- **Documentation and Team Alignment:** All these changes should be documented (this specification itself serves as documentation). We will update the project README or docs with new pages, new flows, and especially an updated **glossary** so everyone uses consistent terms [201]. For example, clarify that "Lead.status 'converted'" means Closed Won and we consider that lead finished, etc., to avoid confusion. Provide the state mapping table (from Section 2) to the team to avoid any misunderstanding of where a record should appear [33].

- Ensure developers know to remove any leftover `// TODO` and implement them now (the audit's mandate: no TODOs left; everything should have a decided action) [202]. For instance, if code said `// TODO: handle Nurture page`, now we do it.

By executing the above fixes and enhancements, we will achieve a CRM system that is robust, user-friendly, and aligned with the business requirements. Naming inconsistencies will be eliminated, the schema will be lean and precise, and all major issues from the audit (vanishing leads, dead buttons, partial workflows, security holes) will be resolved.

This target-state specification is now the **Single Source of Truth** for the CRM module design. The development team should implement all changes as described and refer to this document (along with the audit report [27] [1] and blueprint guidelines [24] [73]) when in doubt. Once completed, the UGC BCP CRM will function as a reliable, audit-proof system where every lead is tracked from creation to conversion, every user action is accounted for, and the data model supports the company's growth without technical debt from the legacy design.

Sources: This plan was derived from a detailed audit of the existing implementation [27] [1] and the UGC blueprint architecture documents [24] [33], ensuring that all critical issues (P0/P1 from the audit) are addressed and the CRM module adheres to UGC's architectural standards moving forward [203].

---

1 2 3 4 5 8 9 10 11 12 14 27 82 83 87 88 92 116 187 194 report.md
file://file-QJ1KruZunxknxmTPiBrrrD

6 7 30 63 64 66 67 68 69 73 74 81 ssot-crm-module.md
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/crm-module-enhance/ssot-crm-module.md

13 16 17 24 25 28 29 32 33 46 48 49 50 51 52 53 89 90 91 114 191 197 201 202 203 ssot_crm_module.txt
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/ssot_crm_module.txt

15 26 94 95 BLUEPRINT.txt
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/BLUEPRINT.txt

18 84 route.ts
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/app/api/crm/leads/[id]/handover/route.ts

19 85 196 198 route.ts
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/app/api/crm/leads/[id]/claim/route.ts

20 43 44 45 54 55 56 57 59 72 75 76 77 101 104 105 108 109 110 111 112 132 199 200 route.ts
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/app/api/crm/leads/[id]/convert/route.ts

21 22 23 31 34 35 36 37 38 39 40 41 42 58 61 62 70 71 78 79 80 86 106 107 113 117 118 119 122 123 124 125 127 141 170 171 176 177 179 181 183 192 193 195 10_crm_rpc_rls.sql
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/supabase/migrations/10_crm_rpc_rls.sql

47 138 154 155 156 157 158 159 160 161 162 163 167 168 169 190 16_target_status_ssot.sql
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/supabase/migrations/16_target_status_ssot.sql

60 93 99 100 115 126 128 129 137 139 140 149 151 152 153 172 173 178 180 182 184 185 186 09_crm_rebuild.sql
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/supabase/migrations/09_crm_rebuild.sql

65 96 97 98 131 142 database.ts
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/lib/types/database.ts

102 103 130 133 134 135 136 143 144 145 146 147 148 150 164 165 166 174 175 188 189 database_ugc-bcp.txt
file://file_0000000034a871fa9811d1e2041aeaff

120 121 crm-reset.md
https://github.com/assujiar/ugc-bcp-2/blob/865ffb253624b76524c0730a03aac59748ff368d/docs/crm-reset.md