ChatGPT

Support Domain Implementation Guide (Consolidated)

This consolidated guide provides a complete blueprint for implementing the Dyno CRM support/helpdesk domain. It synthesizes the patterns extracted from the Company domain (see `domain_object_implementation_guide.md`) and applies them uniformly to every table introduced in the `002_support_domain_schema` migration. The goal is to produce a consistent, repeatable implementation across models, schemas, services, routes, events, and messaging.

Shared Patterns and Conventions

All support domain objects follow the architectural patterns established in the Company domain. The key conventions are:

- **ORM Models**: Each table corresponds to a SQLAlchemy model with columns mirroring the DDL. Use `UUID` for identifiers, include `tenant_id` for tenant scoping, define audit fields (`created_at`, `updated_at`, `created_by`, `updated_by`), and enforce unique and foreign key constraints. Relationships use explicit `primaryjoin` and `foreign_keys` declarations to avoid ambiguity.
- **Pydantic Schemas**: Each domain has request and response models defined in `app/domain/schemas/<domain>.py`. Create and update request schemas derive from a common base; response models set `model_config = ConfigDict(from_attributes=True, extra="ignore")` to allow ORM objects directly. Validate inputs (e.g. enumeration values, JSON shape) and provide defaults.
- **Service Layer**: Business logic lives in `app/domain/services/<domain>_service.py`. Services encapsulate list/get/create/update/delete operations, enforce tenant scoping, set audit fields from the `X-User` header, validate relationships, and call `commit_or_raise` to ensure transactional safety. Services emit events via producers after successful commits.
- **Events and Producers**: Event payloads are defined in `app/domain/schemas/events/<domain>_event.py`. Each mutation publishes a corresponding event using a producer in `app/messaging/producers/<domain>_producer.py`. Event names follow the convention `<exchange>.domain.action` (e.g. `dyno-crm.ticket.created`). Deltas capture changed fields on update events.
- **API Routes**: FastAPI routers expose CRUD endpoints. Tenant routes live under `/tenants/{tenant_id}` and include nested resources (e.g. `/tickets/{ticket_id}/messages`). Admin routes live under `/admin` and accept `tenant_id` as a query parameter. Separate routers are created for top-level and nested resources, mirroring the Company pattern. Use dependency injection for database sessions and user context.
- **Nested Resources**: Child tables (e.g. `ticket_message`, `kb_article_revision`) are exposed via nested routes under their parent. Services manage nested object lifecycles and propagate events. Deleting a parent cascades to its children via ORM relationships.

- **Tenant Safety**: All foreign keys include `tenant_id` in the composite key to enforce that associations do not cross tenants. Services always filter by `tenant_id`, and routes require it.
- **Audit and Validation**: Every mutation sets `created_at` / `updated_at` automatically and populates `created_by` / `updated_by` from the `X-User` header (defaulting to `"anonymous"`). Check constraints and validators ensure enumerations and numeric ranges are respected. JSON fields (`JSONB`) are validated for shape and allowed keys.

These conventions apply uniformly across the support domain. The following sections describe each domain object, summarizing its responsibility and pointing to the file-level implementation guides.

Domain Object Summary

Domain Object	Responsibility	Nested Under
tenant_user_shadow	Read-only projection of users from the Tenant Service. Provides names and emails for assignment and display.	Top-level
tenant_group_shadow	Read-only projection of groups from the Tenant Service. Used for queue assignment and group names.	Top-level
group_profile	CRM-local metadata indicating whether a group is a support queue and defining default SLA policies, routing config, and AI work mode.	Top-level (links to `tenant_group_shadow`)
inbound_channel	Defines entry points (email inbox, chat widget, SMS number, voice line, API integration) that create or update tickets.	Top-level
ticket	Core support case containing subject, description, status, priority, type, assignments, timestamps, custom fields, orchestration linkage, and AI posture.	Top-level
ticket_participant	Additional participants (requester, CCs, followers) on a ticket, including contacts and agents.	Nested under `ticket`

Domain Object	Responsibility	Nested Under
ticket_tag	Simple tags attached to tickets for flexible categorization and filtering.	Nested under `ticket`
ticket_message	Append-only conversation thread of messages and notes on a ticket. Supports author types and distinguishes public vs internal messages.	Nested under `ticket`
ticket_attachment	File attachments linked to tickets or specific messages. Stores storage provider and key but not the file itself.	Nested under `ticket` (optionally `ticket_message`)
ticket_assignment	History of ticket assignments to groups and users, including who made the assignment and why.	Nested under `ticket`
ticket_audit	Append-only audit timeline capturing discrete events and before/after snapshots for a ticket.	Nested under `ticket`
ticket_form, **ticket_field_def**, **ticket_form_field**, **ticket_field_value**	Custom ticket intake forms and fields. Definitions are top-level; values are stored per ticket.	`ticket_field_value` nested under `ticket`; others top-level
sla_policy, **sla_target**, **ticket_sla_state**	SLA definitions and computed state. Policies and targets are top-level; states are computed per ticket.	`ticket_sla_state` nested under `ticket`
ticket_task_mirror	Mirror of orchestration tasks for a ticket, allowing the UI to display tasks without querying Flowable.	Nested under `ticket`
ticket_ai_work_ref	Records AI Workforce sessions acting on a ticket, including agent key, purpose, status, outcome, and confidence.	Nested under `ticket`
support_view	Saved filters and view configurations for tickets. Allows agents to quickly load predefined lists.	Top-level

Domain Object	Responsibility	Nested Under
support_macro	Predefined sequences of ticket actions that agents can apply with one click. Defines actions but does not execute them.	Top-level
ticket_time_entry	Records time spent by agents on tickets for time tracking and billing.	Nested under `ticket`
csat_survey, **csat_response**	CSAT questionnaires (`csat_survey`) and customer ratings and comments (`csat_response`).	Surveys are top-level; responses nested under `ticket`
kb_category, **kb_section**, **kb_article**, **kb_article_revision**, **kb_article_feedback**	Hierarchical knowledge base: categories contain sections; sections contain articles; articles have revisions and collect feedback.	Nested hierarchy (sections under categories, articles under sections, revisions and feedback under articles)
ticket_metrics, **ticket_status_duration**	Reporting primitives: aggregate counters per ticket and time-in-status records. Populated by background jobs.	Nested under `ticket` (read-only)

Each domain object is fully documented in its own implementation guide file (e.g. `ticket_implementation_guide.md`, `kb_article_implementation_guide.md`). Below is a high-level summary of the behaviour, file structure, and special considerations for each group of objects.

Tenant Projections and Queue Profiles

- **tenant_user_shadow** and **tenant_group_shadow** are read-only projections from the Tenant Service. Implement only models, read-only schemas, list/get services, and read-only routes. No events or producers are needed. See `tenant_user_shadow_implementation_guide.md` and `tenant_group_shadow_implementation_guide.md`.
- **group_profile** attaches support-specific metadata to mirrored groups. Provide full CRUD via model, schemas, service, events, and routes. Admins use this to mark groups as support queues and to assign default SLA policies and AI modes. See `group_profile_implementation_guide.md`.

Channels and Entry Points

- **inbound_channel** defines email inboxes, chat widgets, SMS numbers, voice lines, API integrations, and other entry points that create or update tickets. CRUD operations allow admins to configure channels. The `external_ref` identifies the provider resource. See `inbound_channel_implementation_guide.md`.

Ticket Core and Related Entities

- **ticket** is the central support case. Implement it as a top-level model with a rich schema capturing status, priority, type, assignments, timestamps, custom fields, orchestration linkage, and AI posture. Provide full CRUD operations with JSON Patch support for updates, nested resource management for messages and attachments, and event emission for creation, updates, and deletion. See `ticket_implementation_guide.md`.
- **ticket_participant** records additional participants on a ticket (requester, CCs, followers). It is a nested resource under tickets with create/delete operations. It enforces unique participants per role and distinguishes contacts from agents. See `ticket_participant_implementation_guide.md`.
- **ticket_tag** stores tags attached to tickets. Implement it as a nested resource with simple create and delete operations. Tags are case-insensitive and unique per ticket. See `ticket_tag_implementation_guide.md`.
- **ticket_message** is an append-only log of messages and notes. Messages support different author types (`contact`, `agent`, `system`, `ai`), channels, public vs internal flags, and metadata for deduplication. Implement create and list endpoints; do not support updates or deletions except for administrative corrections. See `ticket_message_implementation_guide.md`.
- **ticket_attachment** stores metadata about file uploads associated with tickets or messages. Provide endpoints to list and create attachments; actual file storage is handled externally. See `ticket_attachment_implementation_guide.md`.
- **ticket_assignment** logs each assignment or reassignment of a ticket. Entries include the target group/user, the user who performed the assignment, the reason, and a reference to an AI session if applicable. Implement this as a nested resource with create operations triggered by assignment actions. See `ticket_assignment_implementation_guide.md`.
- **ticket_audit** captures all discrete ticket events (status changes, priority changes, tag additions, field updates, message additions, etc.) along with before/after state snapshots. It is an append-only log used for timeline views and compliance. Implement read-only access and append operations when ticket state changes. See `ticket_audit_implementation_guide.md`.

Custom Forms and Fields

- **ticket_form**, **ticket_field_def**, **ticket_form_field**, and **ticket_field_value** support custom ticket intake forms. Admins define forms and fields; forms specify which fields and in what order; values are stored per ticket. Implement full CRUD for form and field definitions. `ticket_field_value` is a nested resource under tickets and should be created and updated when tickets are created or edited using a form. See `ticket_form_implementation_guide.md`, `ticket_field_def_implementation_guide.md`, `ticket_form_field_implementation_guide.md`, and `ticket_field_value_implementation_guide.md`.

SLA Management

- **sla_policy** defines named SLA policies with match rules (e.g. channel, priority). **sla_target** expresses per-priority thresholds for first/next response and resolution times. Provide full CRUD operations and emit events. See `sla_policy_implementation_guide.md` and `sla_target_implementation_guide.md`.
- **ticket_sla_state** stores computed deadlines and breach flags for each ticket. It is updated by the orchestration service; tenants and admins can only read it. Provide read-only endpoints for tenants

and optional update endpoints for orchestration via admin routes. See `ticket_sla_state_implementation_guide.md`.

Orchestration and AI Integration

- **ticket_task_mirror** mirrors tasks created by the orchestration engine (Flowable) for a ticket. The mirror allows the UI to display tasks without querying Flowable directly. It is updated via events from the orchestration service and provides read-only endpoints for agents. See `ticket_task_mirror_implementation_guide.md`.
- **ticket_ai_work_ref** records AI Workforce sessions executed on a ticket, including agent key, purpose, status, outcome, and confidence. It is upserted by the AI integration layer and exposed via read-only endpoints. See `ticket_ai_work_ref_implementation_guide.md`.

Views and Macros

- **support_view** stores saved filters and column configurations that define "views" of tickets. Agents can create, update, delete, and list views. Implement full CRUD operations, validate filter and sort definitions, and emit events. See `support_view_implementation_guide.md`.
- **support_macro** defines pre-configured sequences of ticket actions that agents can apply with one click. Implement full CRUD operations to manage macro definitions, validate the actions JSON, and emit events. Execution of macros is handled elsewhere in the application. See `support_macro_implementation_guide.md`.

Time Tracking and Satisfaction

- **ticket_time_entry** records time spent by agents on tickets. Provide nested routes under tickets to list, create, update, and delete time entries. Validate that `minutes_spent` is non-negative. Emit events on creation, update, and deletion. See `ticket_time_entry_implementation_guide.md`.
- **csat_survey** defines CSAT questionnaires. Admins manage surveys via CRUD endpoints. See `csat_survey_implementation_guide.md`.
- **csat_response** stores customer ratings and comments. Responses are nested under tickets. Provide create and read endpoints; updates and deletes are restricted to admins for corrections. See `csat_response_implementation_guide.md`.

Knowledge Base

- **kb_category**, **kb_section**, **kb_article**, **kb_article_revision**, and **kb_article_feedback** implement a hierarchical knowledge base. Categories contain sections; sections contain articles; articles have revisions and collect feedback. Provide full CRUD for categories, sections, and articles; revisions are append-only; feedback is append-only with optional deletion by admins. Use nested routes to reflect the hierarchy. See `kb_category_implementation_guide.md`, `kb_section_implementation_guide.md`, `kb_article_implementation_guide.md`, `kb_article_revision_implementation_guide.md`, and `kb_article_feedback_implementation_guide.md`.

Reporting Primitives

- **ticket_metrics** stores aggregate counters per ticket (reply count, reopen count). Maintain metrics via internal jobs; provide read-only endpoints for tenants and optional update endpoints for admins. Emit update events so dashboards can refresh. See `ticket_metrics_implementation_guide.md`.
- **ticket_status_duration** records time-in-status records for tickets. Create and update these records when tickets change status, typically via background jobs processing ticket audits. Expose read-only endpoints and emit events. See `ticket_status_duration_implementation_guide.md`.

Implementing the Support Domain

To implement the support domain:

1. **Follow the patterns** documented in `domain_object_implementation_guide.md` for each domain object. For every table, create the corresponding model, schemas, event payloads, producer, service, and routes (tenant and admin). Align names and file structures with the examples provided for the Company domain.
2. **Respect nesting and scoping**. Child entities live under their parents in the URL hierarchy. Admin routes accept `tenant_id` as a query parameter rather than as part of the path.
3. **Validate and enforce constraints**. Ensure that enumerations match the DDL check constraints, that JSON fields adhere to expected shapes, and that unique constraints are enforced at both the database and service layers.
4. **Emit events** for every mutation. After committing a transaction, call the appropriate producer to publish an event with the full snapshot and, for updates, a delta capturing changed fields. This enables downstream services (search indexes, analytics, orchestration) to react to changes.
5. **Use audit fields**. Populate `created_at` and `updated_at` automatically and set `created_by` and `updated_by` from the `X-User` header or system context. These fields support compliance and traceability.
6. **Provide read-only endpoints** for objects that are maintained by internal processes (e.g. shadows, SLA state, metrics, status durations). Do not expose update or delete operations for these objects to end users.

By following this consolidated guide and the per-object implementation guides, a developer can implement the entire support/helpdesk domain in Dyno CRM with confidence and consistency. Each domain object will adhere to the same patterns, enabling maintainable, predictable behavior across the system.
