ChatGPT

Consolidated CRM Change Request (Domain-by-Domain)

This document merges the requirements from the three source change requests into one authoritative specification. It defines the purpose, scope, schema changes, behavioural rules, eventing, automation integration and acceptance criteria for each domain object in the CRM. The order of sections reflects the dependency chain: automation primitives first, then pipeline entities, followed by core CRM entities, list entities and the visibility subsystem.

1. Automation

1.1 `automation_action` – Definition of CRM actions

Purpose & Scope
`automation_action` stores **declarative rules** that tell the CRM what to do when certain events occur. It generalises the existing chat-agent `form_action` model to cover pipelines, stages, lists and other CRM objects. Actions can trigger webhooks, Flowable workflows, internal events or AI workers. Each action definition is tenant-scoped and does not execute itself; it only defines when and how to run.

Required Schema Changes

Create a table `automation_action` with these fields:

Column	Type/Constraint	Description
`id`	UUID **PRIMARY KEY**	Unique identifier for the action.
`tenant_id`	UUID **NOT NULL**	Tenant context; all actions are tenant scoped.
`entity_type`	VARCHAR(50) **NOT NULL**	The type of record the action targets (e.g. `deal`, `ticket`, `pipeline`, `pipeline_stage`, `list`, `activity`). Allows the same trigger definitions to be used across domains.

Column	Type/Constraint	Description
`scope_type`	ENUM(`PIPELINE`, `PIPELINE_STAGE`, `LIST`, `ENTITY`) **NOT NULL**	Determines how the action applies: to an entire pipeline, a specific stage, a list, or a generic entity (contact, deal, ticket, etc.).
`scope_id`	UUID **NULLABLE**	ID of the pipeline, stage or list, depending on `scope_type`. Must be `NULL` for actions whose scope is the entire entity type (e.g. all `deal` records), and **NOT NULL** when `scope_type` is `PIPELINE_STAGE` or `LIST`.
`trigger_event`	ENUM of events defined for the target domain	The event that causes this action to fire (see trigger semantics below).
`condition`	JSONB **NULLABLE**	Optional logical predicate evaluated against the record and context (e.g. `{ "amount": { "$gt": 5000 }, "stageId": "..." }`). If omitted, the action runs whenever its event occurs. Conditions are always tenant-safe.
`action_type`	ENUM(`WEBHOOK`, `WORKFLOW`, `EVENT`, `AIWORKER`) **NOT NULL**	Determines what the action will do: call a webhook, signal a Flowable workflow, emit an internal event, or invoke an AI worker.

Column	Type/Constraint	Description
`config`	JSONB **NOT NULL**	Configuration specific to the action type (e.g. webhook URL and headers, workflow key and variables, event type, AI worker parameters).
`priority`	INTEGER **NOT NULL DEFAULT 100**	Order of execution; smaller values run first. Actions with the same priority run in an undefined but repeatable order.
`enabled`	BOOLEAN **NOT NULL DEFAULT TRUE**	Whether the action is active. Disabled actions are ignored.
`inherit_pipeline_actions`	BOOLEAN **DEFAULT TRUE**	Only applicable when `scope_type = 'PIPELINE_STAGE'`. If `TRUE` (default), stage-level actions will run **in addition** to any applicable pipeline-level actions. If `FALSE`, stage-level actions suppress pipeline stage-triggered actions. Pipeline-level triggers such as `ON_RECORD_CREATED`, `ON_WON`, `ON_LOST` still execute.
`created_at` / `updated_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Audit timestamps.
`created_by` / `updated_by`	VARCHAR(100) **NULLABLE**	Optional audit fields referencing the user who defined/modified the action.

Indexes and Constraints

- **Primary Key:** `(id)`

- **Tenant Isolation:** Add `(tenant_id, id)` unique constraint to enforce uniqueness per tenant. Index on `(tenant_id)` for partition pruning.
- **Scope Validity:**
- When `scope_type = 'PIPELINE'`, `scope_id` must either be `NULL` (applies to all pipelines of `entity_type`) or a valid `pipeline.id` (must match `entity_type` = `deal` or `ticket` accordingly).
- When `scope_type = 'PIPELINE_STAGE'`, `scope_id` must reference a valid `pipeline_stage.id` (and the `entity_type` must be `deal` or other stage-based entity). A check constraint ensures that `inherit_pipeline_actions` is not `NULL`.
- When `scope_type = 'LIST'`, `scope_id` must reference a valid `list.id`.
- When `scope_type = 'ENTITY'`, `scope_id` must be `NULL` and the action applies to the entire `entity_type`.
- **Trigger Enforcement:** A check constraint ensures that `trigger_event` belongs to the allowed set for the entity (see trigger lists below). This may be implemented via a lookup table if the database does not support dynamic enum validations.

API Changes

- **Action Management Endpoints:** Add endpoints for creating, updating, enabling/disabling and deleting `automation_action` rows. Each call must specify `tenant_id` or derive it from the authenticated user. Attempts to create invalid combinations of `entity_type`, `scope_type`, `scope_id` or `trigger_event` should yield a `400 Bad Request`.
- **Listing and Filtering:** Provide APIs to list actions filtered by `entity_type`, `scope_type`, `trigger_event` and status. Support ordering by `priority` to show execution order.

Behavioural Rules & Lifecycle

- **Additive Composition:** By default, actions scoped to the pipeline (**scope_type = 'PIPELINE'**) and to the stage (**scope_type = 'PIPELINE_STAGE'**) both execute on a stage transition. Stage-level actions have their own `inherit_pipeline_actions` flag controlling whether pipeline actions for stage events run in addition to stage actions.
- **Override Mechanism:** Setting `inherit_pipeline_actions = false` on a stage action suppresses pipeline actions for stage triggers (`ON_STAGE_ENTER`, `ON_STAGE_EXIT`, `ON_STAGE_DWELL`). Pipeline triggers unrelated to stage transitions (e.g. `ON_RECORD_CREATED`, `ON_WON`, `ON_LOST`) still fire.
- **Phase-2 Composition Mode (Optional):** If finer control is required in a future release, add an enum `composition_mode` with values `MERGE` (pipeline + stage), `STAGE_ONLY` (only stage actions), `PIPELINE_ONLY` (only pipeline actions). Not required in v1; document as a possible extension.
- **Idempotency & Deduplication:** When an event occurs, each qualifying action is invoked exactly once. Downstream execution uses a deterministic `execution_key` derived from `tenant_id`, `action_id`, `trigger_event`, `entity_id` and, for stage transitions, the stage transition identifier. This prevents duplicate executions due to retries.
- **Disabled Actions:** When `enabled = false`, the action is ignored entirely. The record remains for audit/history purposes.

Trigger Definitions

`trigger_event` values depend on the `entity_type` and `scope_type`:

- **Pipeline-level triggers:**
- `ON_RECORD_CREATED` – fires when a new record (deal or ticket) is created in the pipeline.
- `ON_STAGE_CHANGED` – fires whenever the stage ID of a record changes within the pipeline.
- `ON_WON` – fires when a deal or ticket enters a terminal "won" stage.
- `ON_LOST` – fires when a deal or ticket enters a terminal "lost" stage.
- `ON_DWELL` – fires when a record exceeds a configured dwell time in any stage (requires stage dwell timers from Flowable; see Stage Automation below).

- **Stage-level triggers:**

- `ON_STAGE_ENTER` – fires when a record enters this specific stage.
- `ON_STAGE_EXIT` – fires when a record leaves this stage.
- `ON_STAGE_DWELL` – fires when the dwell timer for this specific stage expires.

- **List triggers:**

- `ON_MEMBER_ADDED` – fires when a contact/company/deal is added to the list (via `list_membership` insert).
- `ON_MEMBER_REMOVED` – fires when a contact/company/deal is removed from the list.
- `ON_LIST_CREATED` and `ON_LIST_ARCHIVED` – optional triggers; rarely used in the MVP but defined for completeness.

Automation Integration

- Actions are evaluated by an automation engine (e.g. a service or Flowable listener) when events occur. The engine must:
- Gather all enabled actions for the `tenant_id` where `entity_type` matches, and `scope` matches the record and event.
- Sort actions by `priority` (ascending). Execute pipeline-level actions, then stage-level actions.
- Evaluate `condition` JSONB using the event payload. The condition must evaluate to true for the action to fire. Conditions are expressed in a simple JSON predicate language (AND/OR, comparison operators) and must be supported by the automation executor.
- For each qualifying action, emit an `automation_action_execution` record and call the appropriate execution type (webhook, workflow, event or AI worker). Propagate the execution key to support idempotency.
- Support **long-running or time-based triggers** via Flowable timers. For `ON_DWELL` events, when a record enters a stage that has a dwell threshold, Flowable schedules a timer job. The timer emits an `ON_DWELL` stage or pipeline event if not cancelled by a stage change. The automation engine must receive and process these events in the same way as other events.

Acceptance Criteria

1. A tenant admin can create pipeline actions and stage actions specifying conditions and configuration using the new API. Attempts to create invalid combinations (e.g. `scope_type='PIPELINE_STAGE'` without `scope_id`) are rejected.
2. When a record is created, the automation engine evaluates all enabled `ON_RECORD_CREATED` pipeline actions for the correct `entity_type` and tenant, honouring conditions, priority and idempotency rules.
3. When a record enters or exits a stage, both pipeline-level and stage-level actions execute in the correct order unless `inherit_pipeline_actions=false` suppresses them.
4. When membership is added to a list, only list actions with `ON_MEMBER_ADDED` for that list (or for all lists of that object type) fire; if there are no list actions defined, nothing executes.
5. Idempotency ensures that repeating the same event (e.g. due to retry) does not trigger duplicate calls downstream.
6. The `automation_action` table supports soft deletion via `enabled=false`, and disabled actions do not execute.

1.2 `automation_action_execution` – Tracking action runs

Purpose & Scope
`automation_action_execution` stores a historical record of each attempt to execute an automation action. It captures timing, status and results for audit and troubleshooting, similar to `form_action_execution` in the chat agent system. Each execution is tied to a specific `automation_action` and event context.

Required Schema Changes

Create `automation_action_execution` with these fields:

Column	Type/Constraint	Description
`id`	UUID **PRIMARY KEY**	Unique execution identifier.
`tenant_id`	UUID **NOT NULL**	Tenant context.
`action_id`	UUID **NOT NULL**	Foreign key referencing `automation_action.id`. Enforces tenant consistency via `(action_id, tenant_id)` composite reference.
`entity_type`	VARCHAR(50) **NOT NULL**	The record type on which the action fired.
`entity_id`	UUID **NOT NULL**	ID of the record associated with the event.

Column	Type/Constraint	Description
`pipeline_id`	UUID **NULLABLE**	ID of the pipeline if the event relates to a pipeline or stage.
`from_stage_id`	UUID **NULLABLE**	Previous stage ID for `ON_STAGE_CHANGED`, `ON_STAGE_EXIT` events.
`to_stage_id`	UUID **NULLABLE**	New stage ID for `ON_STAGE_CHANGED`, `ON_STAGE_ENTER` events.
`list_id`	UUID **NULLABLE**	ID of the list if the event is `ON_MEMBER_ADDED` or `ON_MEMBER_REMOVED`.
`trigger_event`	VARCHAR(50) **NOT NULL**	Name of the event that caused this execution.
`execution_key`	VARCHAR(255) **NOT NULL**	Deterministic key used to prevent duplicate execution. Unique per `(tenant_id, execution_key)`.
`status`	ENUM(`PENDING`, `IN_PROGRESS`, `SUCCEEDED`, `FAILED`) **NOT NULL DEFAULT 'PENDING'**	Execution state.
`response_code`	INTEGER **NULLABLE**	HTTP status code from a webhook or analogous code for workflows/events.
`response_body`	JSONB **NULLABLE**	Raw response body or structured output.
`error_message`	TEXT **NULLABLE**	Error description for failed actions.
`triggered_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Timestamp when the event was processed.
`started_at`	TIMESTAMPTZ **NULLABLE**	Timestamp when the action began executing.
`completed_at`	TIMESTAMPTZ **NULLABLE**	Timestamp when the action finished.
`created_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Record creation timestamp.

Indexes and Constraints

- **Primary Key:** `(id)`
- **Tenant Isolation:** Unique constraint on `(tenant_id, execution_key)` to guarantee idempotent execution.
- **Foreign Key:** `(tenant_id, action_id)` references `(tenant_id, id)` in `automation_action` to enforce that the execution belongs to the same tenant as the action definition.
- **Indexes for Analytics:** Add indexes on `(tenant_id, action_id, status)` and `(tenant_id, status)` to support dashboards (e.g. number of failed actions, average duration).

API Changes

- Provide read-only endpoints to query the execution history by `action_id`, `entity_id`, `status` or date range. These are used by admins to view logs and debug errors.
- Expose an endpoint to re-trigger a failed execution manually (with safeguards to avoid double side effects). It must check idempotency keys.

Behavioural Rules & Lifecycle

- **Execution Status:**
- `PENDING` – row inserted but execution not started.
- `IN_PROGRESS` – action is being performed (webhook call in flight, workflow started).
- `SUCCEEDED` – action completed successfully; downstream system confirmed success.
- `FAILED` – action failed after all retries. `error_message` must be populated.
- **Retries & Idempotency:** The automation engine may retry failed executions. It must set `execution_key` such that duplicate deliveries are suppressed by the receiver or by the CRM via the unique constraint.
- **Audit & SLA:** All execution rows are retained for a minimum retention period (e.g. 90 days) to allow auditing. Time to start and time to complete can be derived from `triggered_at`, `started_at` and `completed_at` for SLA reporting.

Acceptance Criteria

1. Each action execution creates an `automation_action_execution` record with status `PENDING` and a unique `execution_key`.
2. When the action starts, the status transitions to `IN_PROGRESS`; when it ends, the record is updated to `SUCCEEDED` or `FAILED` accordingly.
3. Duplicate execution attempts with the same `execution_key` fail with a uniqueness violation or are ignored, preventing side effects.
4. Admins can list executions by tenant, action or status and can see timestamps and responses for debugging.
5. If a webhook returns a non-2xx code or a workflow returns an error, the status becomes `FAILED` and `error_message` is recorded.

---

2. Pipelines

2.1 `pipeline` – Multi-pipeline support and lifecycle

Purpose & Scope
The `pipeline` table represents a workflow for stage-based records (deals, tickets, leads, etc.). Each pipeline defines a set of ordered stages and is scoped to a specific object type. This change introduces multi-pipeline support, ordering, stable keys and archiving, ensuring that pipelines can be managed safely and referenced reliably in APIs and automation.

Required Schema Changes

Alter the existing `pipeline` table (or create if absent) to include:

Column	Type/Constraint	Description
`object_type`	ENUM(`deal`, `ticket`, `lead`, ...)**NOT NULL**	Indicates which record type the pipeline belongs to. Required to support multiple pipelines per object.
`display_order`	INTEGER **NOT NULL**	Order in which pipelines are presented in the UI. Lower numbers appear first. Unique per `(tenant_id, object_type)`.
`is_active`	BOOLEAN **NOT NULL DEFAULT TRUE**	Whether the pipeline is active. Inactive pipelines remain in the database but do not accept new records. Soft deletion uses this flag rather than dropping the row.
`key`	VARCHAR(100) **NOT NULL**	Stable, user-defined slug used for API calls and configuration. Unique per `(tenant_id, object_type)`. Once assigned, `key` cannot be changed without migrating downstream references.

Existing columns (`id`, `tenant_id`, `name`, `created_at`, `updated_at`, `created_by`, `updated_by`) remain unchanged.

Indexes and Constraints

- **Unique Keys:** Ensure `(tenant_id, object_type, key)` is unique; ensure `(tenant_id, object_type, display_order)` is unique to prevent ordering collisions.
- **Tenant Isolation:** Primary key `(id)` remains unique; maintain `(id, tenant_id)` unique constraint to enforce composite foreign keys from child tables.

- **Name Uniqueness:** Continue to enforce `(tenant_id, name)` uniqueness via existing index. Names may change, but `key` does not.

API Changes

- **Create Pipeline:** Accept `name`, `object_type`, `key` and `display_order` in the payload. The API must reject attempts to create a pipeline with a duplicate `key` or `display_order` within the same object type.
- **Update Pipeline:** Allow updating `name`, `display_order` and `is_active`. Changing `object_type` or `key` is not allowed because it would break references. Attempting to deactivate (`is_active=false`) a pipeline with active records must require a `force` flag and explicit reassignment (see behavioural rules below).
- **List Pipelines:** Support filtering by `object_type` and optionally by `is_active`, returning pipelines ordered by `display_order`.

Behavioural Rules & Lifecycle Constraints

- **Multi-pipeline per Object:** Tenants may define multiple pipelines for each object type (e.g. “New Sales”, “Renewals”, “Support Tickets”). A record must belong to exactly one pipeline at any time. `pipeline_id` on the record is required.
- **Activation & Deactivation:** Setting `is_active=false` archives the pipeline. New records cannot be assigned to inactive pipelines unless `force` is used via an admin operation. Existing records remain and can finish their lifecycle. A pipeline may be deleted only when no records remain (except via a `force=true` admin API that triggers reassignment or deletion).
- **Reordering:** Administrators may change `display_order`. The CRM must update the `display_order` fields atomically to maintain uniqueness. Reordering triggers UI updates but does not affect existing records.
- **Enforcement Rules:**
- Deleting a pipeline with active records must fail unless `force=true`. In a forced deletion, all records must be reassigned to another pipeline or archived via automation.
- Changing `object_type` of a pipeline is not permitted.
- Changing `key` after creation is not permitted; to rename a pipeline’s API slug, create a new pipeline and migrate records.

Event Emission Requirements

- **Pipeline Created:** Emit `pipeline_created` event with payload `{ pipelineId, tenantId, objectType, key }`.
- **Pipeline Updated:** Emit `pipeline_updated` event when `name`, `display_order` or `is_active` changes. Payload must include the changed fields.
- **Pipeline Deleted or Archived:** Emit `pipeline_deleted` or `pipeline_archived` event with payload `{ pipelineId, tenantId, objectType }` when a pipeline is hard-deleted or archived.
- **Pipeline Reordered:** Emit `pipeline_reordered` with payload `{ tenantId, objectType, orderedPipelineIds }` when multiple pipelines are reordered; used to synchronise caches.

Automation Integration

- Pipeline-level actions apply whenever a stage change occurs in any stage of this pipeline (subject to conditions). Stage-level actions apply only to specific stages.
- `ON_RECORD_CREATED`, `ON_STAGE_CHANGED`, `ON_WON`, `ON_LOST` and `ON_DWELL` triggers rely on the pipeline ID and object type. When a pipeline is deactivated, actions configured for that pipeline still fire for existing records but new records should not trigger actions unless `force=true`.

Acceptance Criteria

1. A tenant can create multiple pipelines for deals and tickets; each has a unique `key` and `display_order`.
2. Deleting or archiving a pipeline with active records is prevented by default, requiring explicit reassignment or `force` deletion.
3. Updating `display_order` properly reorders pipelines without duplication; API returns pipelines sorted by `display_order`.
4. Pipeline events are emitted on create, update, deletion/archival and reorder, with payloads containing stable identifiers.

2.2 `pipeline_stage` – Stage semantics and forecasting fields

Purpose & Scope

`pipeline_stage` defines the ordered steps within a pipeline for stage-based records. Stages drive forecasting, automation and reporting. This change adds probability, stage type and terminal flags, enabling weighted pipeline analysis and clear lifecycle boundaries. It also introduces ordering and optional override semantics for automation.

Required Schema Changes

Modify `pipeline_stage` as follows:

Column	Type/Constraint	Description
`display_order`	INTEGER **NOT NULL**	Order of stages within a pipeline. Renamed from `stage_order` for clarity. Must be unique per `(pipeline_id)`.
`probability`	NUMERIC(5,2) **NULLABLE**	Stage-level probability (0.00–1.00) for forecasting. Null means “use default probability” or not set.

Column	Type/Constraint	Description
`stage_type`	ENUM(`open` , `won` , `lost`) **NOT NULL DEFAULT 'open'**	Semantic classification of the stage. `won` and `lost` imply terminal outcomes; `open` indicates ongoing.
`is_terminal`	BOOLEAN **GENERATED ALWAYS AS** (`stage_type IN ('won','lost')`) **STORED**	Flag for downstream logic. Derived from `stage_type` to prevent conflicting values.
`inherit_pipeline_actions`	BOOLEAN **NOT NULL DEFAULT TRUE**	Whether stage-level actions inherit pipeline actions for stage triggers. See automation section.

Existing columns (`id` , `tenant_id` , `pipeline_id` , `name` , `created_at` , `updated_at` , `created_by` , `updated_by`) remain unchanged. Existing `stage_order` should be migrated to `display_order` . Existing `probability` (NUMERIC(5,2)) remains; rename if necessary.

Indexes and Constraints

- Unique constraint on `(pipeline_id, display_order)` and `(pipeline_id, name)` remains; update indexes accordingly.
- `probability` values must be null or between 0 and 1 inclusive. Add a check constraint `CHECK (probability IS NULL OR (probability >= 0 AND probability <= 1))` .
- Derive `is_terminal` as a stored generated column to ensure data consistency. Existing logic checks stage type at runtime; storing the flag simplifies queries and ensures one place of truth.

API Changes

- **Create Stage:** Accept `name` , `display_order` , `probability` , `stage_type` and `inherit_pipeline_actions` in the payload. Validate that `display_order` is unique for the pipeline and that `probability` and `stage_type` values are within range.
- **Update Stage:** Allow updates to `name` , `display_order` , `probability` , `stage_type` and `inherit_pipeline_actions` . Changing `stage_type` updates `is_terminal` accordingly. Do not allow changing `pipeline_id` (would require reassigning records).

Behavioural Rules & Lifecycle Constraints

- **Ordering:** Records must transition through stages in a defined sequence but the system does not enforce monotonic advancement; it permits skipping stages or moving backwards if required by business logic. However, `display_order` defines the typical progression used for forecasting and reporting.
- **Terminal Stages:** Stages with `stage_type='won'` or `stage_type='lost'` are terminal. Entering these stages marks the record as closed and triggers close logic (for deals, closing date; for tickets, resolution). Terminal stages must be explicit; they are not inferred solely by probability.

- **Probability & Forecasting:** If `probability` is null, no probability is assigned to the stage. Weighted forecasting may fall back to custom logic. `forecast_probability` on the deal can override the stage probability; if both are null, weighted forecast is not calculated.
- **Override Pipeline Actions:** `inherit_pipeline_actions` controls whether pipeline-level stage actions run. This applies only to stage-related triggers. It does not suppress pipeline-wide triggers.

Event Emission Requirements

- **Stage Created/Updated/Deleted:** Emit `pipeline_stage_created`, `pipeline_stage_updated` and `pipeline_stage_deleted` events with payload containing `pipelineId`, `stageId`, `name`, `displayOrder`, `probability`, `stageType`, `inheritPipelineActions` and `tenantId`.
- **Probability/Type Changes:** When `probability` or `stage_type` changes, include both old and new values in the event payload for audit.
- **Stage Reordered:** When stages are reordered, emit `pipeline_stage_reordered` with `pipelineId` and ordered stage IDs.

Automation Integration

- Stage triggers `ON_STAGE_ENTER`, `ON_STAGE_EXIT`, `ON_STAGE_DWELL` use `pipeline_stage.id` and `inherit_pipeline_actions` to determine which actions to fire. Stage actions run after pipeline actions by default.
- Stage dwell timers (see Section 2.3) must schedule a timer when a record enters the stage and cancel it when leaving. When the timer fires, an `ON_STAGE_DWELL` event is emitted. If `inherit_pipeline_actions=false`, pipeline `ON_DWELL` actions for stage triggers are skipped.

Acceptance Criteria

1. Creating, updating and deleting a stage updates the `display_order` uniqueness and emits the correct events.
2. Stage terminality is determined solely by `stage_type`. If an admin sets `stage_type` to `won`, the stage becomes terminal and cannot be made non-terminal without updating `stage_type`.
3. Updating `inherit_pipeline_actions` toggles whether pipeline stage triggers fire. Setting it to `false` prevents pipeline actions on stage triggers; other pipeline triggers still fire.
4. The API rejects invalid `probability` values and unknown `stage_type` values.
5. Stage reordering produces a consistent `display_order` for all stages in the pipeline.

2.3 `stage_history` (or equivalent) – Stage transition auditing

Purpose & Scope

To support forecasting, sales velocity analysis, dwell-time triggers and auditing, the CRM must record every stage transition. This table can be extended in future to cover all stage-based entities (deals, tickets, leads). Two implementation strategies were discussed: a dedicated history table or an immutable event log. **We select a dedicated history table** for immediate analytics, while still emitting events for downstream consumers.

Required Schema Changes

Create a table `stage_history` with:

Column	Type/Constraint	Description
`id`	UUID **PRIMARY KEY**	Unique identifier for the history record.
`tenant_id`	UUID **NOT NULL**	Tenant context.
`entity_type`	VARCHAR(50) **NOT NULL**	The type of record (e.g. `deal`, `ticket`, `lead`).
`entity_id`	UUID **NOT NULL**	ID of the record changing stage.
`pipeline_id`	UUID **NOT NULL**	ID of the pipeline the record belongs to. Foreign key referencing `pipeline.id` and `(tenant_id)`.
`from_stage_id`	UUID **NULLABLE**	Stage the record is leaving. Nullable for the first stage entry.
`to_stage_id`	UUID **NOT NULL**	Stage the record is entering.
`changed_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Timestamp of the stage change.
`changed_by_user_id`	UUID **NULLABLE**	ID of the user or system that caused the change. References `tenant_user_shadow.user_id` scoped by `tenant_id`.
`source`	VARCHAR(50) **NOT NULL DEFAULT 'api'**	Indicates whether the change came from API, UI, automation or import.

Indexes and Constraints

- **Tenant Isolation:** Composite primary key `(tenant_id, id)` or unique constraint ensures tenant separation.
- **Foreign Keys:**
- `(pipeline_id, tenant_id)` references `pipeline(id, tenant_id)`.
- `(from_stage_id, tenant_id)` and `(to_stage_id, tenant_id)` reference `pipeline_stage(id, tenant_id)`.
- `(changed_by_user_id, tenant_id)` references `tenant_user_shadow(user_id, tenant_id)`.
- **Indexes:**
- `(tenant_id, entity_type, entity_id)` for retrieving history of a record.
- `(tenant_id, pipeline_id, changed_at)` for reporting stage transitions by pipeline.

Event Emission Requirements

- On every insertion into `stage_history`, emit a `stage_changed` event with payload:

```
{
  "tenantId": "…",
  "entityType": "deal",
  "entityId": "…",
  "pipelineId": "…",
  "fromStageId": "…",
  "toStageId": "…",
  "changedAt": "…",
  "changedByUserId": "…",
  "source": "api"
}
```

Downstream consumers derive `ON_STAGE_ENTER` and `ON_STAGE_EXIT` triggers from this event. Do **not** emit separate enter/exit events in the database. The automation engine listens to `stage_changed` events and evaluates actions accordingly.

Automation Integration

- When a record enters a stage, the `stage_changed` event triggers evaluation of `ON_STAGE_ENTER` actions for the new stage and `ON_STAGE_EXIT` actions for the previous stage. The engine uses `inherit_pipeline_actions` and stage composition rules described earlier.
- When a record remains in a stage beyond a configured dwell time, Flowable schedules a dwell timer. If the record is still in the stage when the timer fires, an `ON_STAGE_DWELL` event is emitted. This event is treated similarly to `stage_changed` for triggering actions but indicates a time-based escalation rather than an immediate transition.

Acceptance Criteria

1. Every time a record's `stage_id` changes, a `stage_history` row is created and a `stage_changed` event is emitted with all required fields.
2. History records accurately reflect the previous stage and next stage, including cases where the previous stage is `NULL` (first entry). The event payload matches the inserted row.
3. Stage dwell timers schedule and emit `ON_STAGE_DWELL` events only when a record remains in a stage beyond the configured time. These events include the same payload structure as `stage_changed` events, with `fromStageId` and `toStageId` equal to the current stage.
4. Deleting a record does not delete its stage history; history is retained for at least the configured retention period (e.g. 7 years for auditing).

---

3. CRM Core Entities

3.1 `contact` – Ownership & watchers support

Purpose & Scope

`contact` represents a person associated with a tenant. Contacts are managed by sales and support teams. They must support ownership (responsible user), optional team ownership and watchers so that

multiple reps can follow updates. Contacts are not work items and therefore do not require assignment fields.

Required Schema Changes

Add the following columns to `contact`:

Column	Type/ Constraint	Description
`owned_by_user_id`	UUID **NULLABLE**	ID of the shadow tenant user who owns the contact (responsible for the relationship). References `tenant_user_shadow(user_id)` scoped by `tenant_id`.
`owned_by_group_id`	UUID **NULLABLE**	ID of the tenant group (shadow) that owns the contact. When set, indicates team ownership or a queue. References `tenant_group_shadow(id)` scoped by `tenant_id`.

These fields are optional but strongly recommended. If both are null, the contact is considered unowned. Only one of the two should be non-null at a time; a check constraint enforces `NOT (owned_by_user_id IS NOT NULL AND owned_by_group_id IS NOT NULL)`.

Indexes and Constraints

- Composite foreign keys `(owned_by_user_id, tenant_id)` to `tenant_user_shadow` and `(owned_by_group_id, tenant_id)` to `tenant_group_shadow` enforce tenant consistency.
- Optional index on `(tenant_id, owned_by_user_id)` to query contacts by owner.
- Optional index on `(tenant_id, owned_by_group_id)` to query contacts by owning group.

API Changes

- **Set/Change Owner:** Provide endpoints to set or change the `owned_by_user_id` or `owned_by_group_id` of a contact. When changing owners, ensure the new owner belongs to the same tenant.
- **Get Contact:** Include owner information in the contact payload and list watchers via `record_watcher` (see visibility section). Only users with appropriate permissions can view ownership.

Behavioural Rules & Lifecycle Constraints

- **Single Owner:** A contact can be owned either by a user or by a group, but not both. The application must enforce this when creating or updating a contact.
- **Ownership Transfer:** When a user is deactivated or leaves the tenant, automation should reassign owned contacts to another user or to a default group (e.g. an unassigned queue). This logic may be implemented in Flowable or in a dedicated reassignment service.

- **Watchers:** Contacts do not store watchers directly. Watchers are managed via the `record_watcher` table (see Section 5). When a user views or follows a contact, a row is added to `record_watcher`. Automatic watchers: the owner (user or group) is automatically added as a watcher upon creation or reassignment; watchers can be removed manually without affecting ownership.

Event Emission Requirements

- Emit `ownership_changed` event when `owned_by_user_id` or `owned_by_group_id` changes. Payload includes `tenantId`, `entityType = 'contact'`, `entityId`, `previousOwnerUserId`, `previousOwnerGroupId`, `newOwnerUserId`, `newOwnerGroupId`, and `changedByUserId`.
- Emit `watcher_added` and `watcher_removed` events when a `record_watcher` row is inserted or deleted for a contact. See Section 5.

Automation Integration

- Actions may be defined for `contact` entity via `automation_action` with scope `ENTITY`. Example: send a welcome email (`WEBHOOK`) when a contact is created.
- Ownership change events can trigger automation (e.g. notify the new owner or reassign associated deals). These events map to `ON_FIELD_CHANGE` triggers on `contact` (not explicitly defined, but can be added as a new trigger type if needed).

Acceptance Criteria

1. Contacts may be created without an owner, but setting an owner (user or group) attaches the record to that person or team. Attempting to set both owner fields returns an error.
2. The API returns the correct owner in the contact's data and watchers list via `record_watcher`.
3. Changing a contact's owner updates `owned_by_user_id` / `owned_by_group_id` and emits an `ownership_changed` event.
4. Watchers for contacts are managed via the generic `record_watcher` table; owners automatically become watchers on creation or reassignment.

3.2 `company` – Account-equivalent with ownership, assignment and watchers

Purpose & Scope

`company` represents an organisation (account) associated with the tenant. It aggregates contacts, deals, tickets and other records. The CRM already supports contact-to-company relationships and company hierarchies (`contact_company_relationship`, `company_relationship`). This change keeps `company` as the account entity, adds ownership and optional assignment, and supports watchers. It does not introduce a separate `account` object.

Required Schema Changes

Add the following columns to `company`:

Column	Type/ Constraint	Description
`owned_by_user_id`	UUID **NULLABLE**	Primary owner (account manager). References `tenant_user_shadow(user_id)`.
`owned_by_group_id`	UUID **NULLABLE**	Team owner. References `tenant_group_shadow(id)`. Only one of the owner fields may be non-null.
`assigned_to_user_id`	UUID **NULLABLE**	For specific work items such as onboarding tasks or account reviews. References `tenant_user_shadow`.
`assigned_to_group_id`	UUID **NULLABLE**	For queue-based handling (e.g. support teams). References `tenant_group_shadow`. Only one of the assignment fields may be non-null at a time.

Existing company properties and relationship tables remain unchanged. A check constraint enforces `NOT ((owned_by_user_id IS NOT NULL AND owned_by_group_id IS NOT NULL) OR (assigned_to_user_id IS NOT NULL AND assigned_to_group_id IS NOT NULL))`.

Indexes and Constraints

- Composite foreign keys `(tenant_id, owned_by_user_id)` and `(tenant_id, owned_by_group_id)` ensure tenant-safe references to `tenant_user_shadow` and `tenant_group_shadow`.
- Optional indexes on `(tenant_id, owned_by_user_id)`, `(tenant_id, owned_by_group_id)`, `(tenant_id, assigned_to_user_id)` and `(tenant_id, assigned_to_group_id)` support filtering and reporting.

API Changes

- **Create/Update Company:** Accept `owned_by_user_id`, `owned_by_group_id`, `assigned_to_user_id`, `assigned_to_group_id`. Validate that only one owner field and one assignment field are set. Changing an owner triggers an `ownership_changed` event; changing an assignee triggers an `assignment_changed` event.
- **Get Company:** Include owner and assignment information, and list watchers via `record_watcher`.

Behavioural Rules & Lifecycle Constraints

- **Single Owner & Assignee:** A company may have one owner (user or group) and zero or one active assignee (user or group). Owners represent stewardship; assignees represent current work responsibility (e.g. a support agent working an open ticket for the account). Owners and assignees may be the same or different. Changing owner or assignee must respect single-field rules.
- **Automatic Watchers:** The owner and assignee of a company automatically become watchers via `record_watcher`. When ownership or assignment changes, watchers are updated accordingly. Additional watchers may be added manually for collaboration.

- **Queues and Escalation:** If `assigned_to_group_id` is set and `assigned_to_user_id` is null, the company is in a queue. Users in the group can pick up the work and set themselves as `assigned_to_user_id`, clearing the group assignment. Flowable may automate this based on business rules.
- **Hierarchy & Roll-up:** Existing relationships (`contact_company_relationship` and `company_relationship`) remain; owners and assignees on parent companies do not automatically propagate to children. Hierarchical roll-up of metrics is performed in reporting, not by automatic field propagation.

Event Emission Requirements

- `ownership_changed` events when `owned_by_user_id` or `owned_by_group_id` changes (similar payload to contact events but `entityType='company'`).
- `assignment_changed` events when `assigned_to_user_id` or `assigned_to_group_id` changes. Payload includes previous and new assignments.
- `watcher_added` and `watcher_removed` events for `record_watcher` changes (Section 5). Owners and assignees automatically generate watcher events.

Automation Integration

- **Entity-level actions:** `automation_action` can target the `company` entity type with scope `ENTITY`. Example triggers: `ON_RECORD_CREATED` (new company is added) or `ON_FIELD_CHANGE` (ownership changed), although `ON_FIELD_CHANGE` is not formally defined yet; if needed, define new trigger events for ownership and assignment changes.
- **Assignment & SLA:** Company assignments may drive Flowable workflows for onboarding or support. Example: when `assigned_to_user_id` changes, start a review workflow; when `assigned_to_group_id` is set, signal a queue workflow.

Acceptance Criteria

1. Companies can be created with or without owners and assignees. Setting both owner fields or both assignment fields returns a validation error.
2. Changing owner or assignee updates the record, inserts a watcher row and emits the appropriate events.
3. Owners and assignees are automatically watchers; removing an owner or assignee removes them from watchers. Deactivating a user triggers reassignment logic (external to this schema) and updates watchers accordingly.
4. Listing companies returns the correct owner, assignee and watchers for each record.

3.3 `deal` – Forecasting, ownership, assignment, stage linkage & watchers

Purpose & Scope

`deal` represents a sales opportunity. Deals must be tied to a pipeline and stage for forecasting; they support ownership, assignment, close dates, deal type and forecast probability override. Deals may relate to multiple contacts and a primary company via association tables. Automation and eventing revolve around stage changes and win/loss outcomes.

Required Schema Changes

Alter `deal` to include:

Column	Type/Constraint	Description
`owned_by_user_id`	UUID **NULLABLE**	Primary sales owner. References `tenant_user_shadow(user_id)`.
`owned_by_group_id`	UUID **NULLABLE**	Sales team owner. References `tenant_group_shadow(id)`.
`assigned_to_user_id`	UUID **NULLABLE**	Current assignee (e.g. sales rep actively working the deal). References `tenant_user_shadow(user_id)`.
`assigned_to_group_id`	UUID **NULLABLE**	Queue assignment. References `tenant_group_shadow(id)`.
`pipeline_id`	UUID **NOT NULL**	Foreign key to `pipeline(id)`. Required – deals cannot exist outside a pipeline.
`stage_id`	UUID **NOT NULL**	Foreign key to `pipeline_stage(id)`. Must belong to the same pipeline.
`deal_type`	ENUM(`new`, `renewal`, `upsell`, `other`) **NOT NULL DEFAULT 'new'**	Categorises the opportunity for revenue reporting. Enumerations may be extended.
`close_date`	DATE **NULLABLE**	Target or actual close date. When entering a terminal stage, this defaults to the current date if not provided.
`forecast_probability`	NUMERIC(5,2) **NULLABLE**	Overrides stage probability for this deal. Must be between 0 and 1.
`loss_reason`	VARCHAR(255) **NULLABLE**	Optional reason for lost deals. Required if entering a `lost` stage when configured.

Existing fields (id, tenant_id, name, amount, currency, created_at, updated_at, etc.) remain. Include a check constraint to ensure only one of `(owned_by_user_id, owned_by_group_id)` and only one of `(assigned_to_user_id, assigned_to_group_id)` are set.

Indexes and Constraints

- Composite foreign keys `(pipeline_id, tenant_id)` and `(stage_id, tenant_id)` enforce tenant consistency and correct pipeline-stage pairing.

- Foreign key constraints on `(owned_by_user_id, tenant_id)` to `tenant_user_shadow` and on `(owned_by_group_id, tenant_id)` to `tenant_group_shadow` ensure valid owners.
- Similar constraints for assignment fields.
- Indexes on `(tenant_id, owned_by_user_id)`, `(tenant_id, assigned_to_user_id)`, `(tenant_id, pipeline_id, stage_id)` to support filtering and reporting. Consider a partial index on `(tenant_id, stage_id)` where `stage_type='open'` for active pipeline views.

API Changes

- **Create Deal:** Payload must include `pipeline_id`, `stage_id`, `name` and optionally `deal_type`, `close_date`, owner and assignee. Validate that `stage_id` belongs to `pipeline_id` and that the user or group IDs exist for the tenant.
- **Update Deal:** Allow updating any field except `id` and `tenant_id`. Changing `pipeline_id` requires also updating `stage_id` to a valid stage of the new pipeline.
- **Close Deal:** Provide an explicit endpoint or require that entering a terminal stage sets the `close_date` and `forecast_probability` as needed. A lost deal must supply `loss_reason` if configured as required.

Behavioural Rules & Lifecycle Constraints

- **Pipeline & Stage:** Deals cannot exist outside a pipeline. When a deal is created, `pipeline_id` and `stage_id` must be provided. When the stage changes, update `stage_history` and emit a `stage_changed` event.
- **Ownership vs Assignment:** Owners represent long-term account managers; assignees represent the current worker (e.g. a sales development rep). Assignment can be null while ownership is set. Changing either field triggers events and watcher updates.
- **Terminal Stages & Close Date:** Entering a stage with `stage_type='won'` or `'lost'` marks the deal as closed. If `close_date` is null, set it to the current date. If `stage_type='lost'` and a loss reason is required by configuration, the API must enforce that `loss_reason` is provided. Terminal stages must not be changed back to `open` without explicit override.
- **Forecasting:** Weighted revenue forecasts are computed as `amount * (forecast_probability OR stage.probability)`. If both are null, the forecast for that deal is zero. Reports must respect this logic.
- **Watchers:** Owners and assignees automatically become watchers via `record_watcher`. Additional watchers may subscribe manually. When ownership or assignment changes, watchers are updated accordingly.

Event Emission Requirements

- `ownership_changed` and `assignment_changed` events when the respective fields change. Include old and new values in the payload.
- `stage_changed` events via `stage_history` (see Section 2.3).
- `deal_created` and `deal_updated` events for other field changes (e.g. amount, closeDate). Payload includes changed fields only.
- `deal_closed` event when entering a terminal stage (won or lost), including close date and outcome.

Automation Integration

- **Stage Triggers:** Deals use pipeline and stage actions defined in `automation_action`. `ON_RECORD_CREATED`, `ON_STAGE_CHANGED`, `ON_WON`, `ON_LOST` and `ON_DWELL` triggers drive CRM automation.
- **Ownership & Assignment Triggers:** If needed, define new trigger events such as `ON_OWNERSHIP_CHANGED` and `ON_ASSIGNMENT_CHANGED` to allow actions when a deal is reassigned.
- **Forecasting & AI:** AI workers may suggest next actions (e.g. propose meeting times) based on stage and probability; such actions are triggered by stage events and processed via `AIWORKER` actions.

Acceptance Criteria

1. Creating a deal requires a valid `pipeline_id` and `stage_id`. A validation error is returned if the stage does not belong to the pipeline.
2. Changing the deal's stage updates `stage_id`, inserts into `stage_history` and emits a `stage_changed` event. For terminal stages, `close_date` is set if absent and a `deal_closed` event is emitted.
3. Setting both owner fields or both assignment fields is not allowed. Valid updates update watchers and emit `ownership_changed` or `assignment_changed` events.
4. Changing the pipeline on an existing deal requires updating `stage_id` to a valid stage of the new pipeline; otherwise the API rejects the request.
5. Loss reasons are enforced when configured. Attempting to close a deal in a `lost` stage without `loss_reason` (when required) yields a validation error.
6. Reports can compute weighted forecast using `forecast_probability` or `stage.probability` based on the rules above.

3.4 `lead` – Ownership & basic lifecycle

Purpose & Scope
`lead` represents an unqualified prospect. Leads are similar to contacts but may become contacts or deals upon conversion. For simplicity, leads do not live in pipelines or stages. However, they require ownership for routing, watchers for collaboration and assignment when tasks are attached.

Required Schema Changes

Add the following columns to `lead` (similar to `contact`):

Column	Type/ Constraint	Description
`owned_by_user_id`	UUID **NULLABLE**	Lead owner. References `tenant_user_shadow(user_id)`.
`owned_by_group_id`	UUID **NULLABLE**	Lead owner group (queue). References `tenant_group_shadow(id)`. Only one owner field may be set at a time.

Constraints: only one of these may be non-null at a time.

Indexes and Constraints

- Composite foreign keys `(owned_by_user_id, tenant_id)` and `(owned_by_group_id, tenant_id)` referencing the respective shadow tables.
- Indexes on `(tenant_id, owned_by_user_id)` and `(tenant_id, owned_by_group_id)` support filtering leads by owner.

API Changes

- **Create/Update Lead:** Accept owner fields. Validate that only one is set. Changing owner triggers an `ownership_changed` event.
- **Convert Lead:** Provide an endpoint to convert a lead into a contact and optionally a deal. On conversion, ownership may be mapped to the resulting contact/deal; watchers from the lead should be copied to the contact/deal.

Behavioural Rules & Lifecycle Constraints

- **Ownership:** Leads must have an owner (user or group) for routing. Setting an owner to null indicates unowned leads and may place them in a default queue for triage.
- **Watchers:** Use `record_watcher` for watchers. Owners automatically become watchers. Additional watchers may subscribe. When converting a lead, watchers transfer to the new contact or deal unless explicitly cleared.
- **Conversion:** Converting a lead deletes or archives the lead and creates a contact and/or deal with the lead's data. Ownership and watchers propagate. Emit conversion events for auditing.

Event Emission Requirements

- `ownership_changed` events when owner changes. `lead_created`, `lead_updated`, `lead_converted` for create/update/convert operations.
- `watcher_added` and `watcher_removed` events for watcher changes via `record_watcher`.

Automation Integration

- Actions targeting `lead` entity type can be defined for events such as `ON_RECORD_CREATED` (new lead created) or conversion events. For example, assign a lead from a default queue to an SDR when it meets certain criteria via AI.

Acceptance Criteria

1. Leads can be created with or without owners; setting both owner fields is invalid.
2. Changing a lead's owner updates the record and emits an `ownership_changed` event.
3. Converting a lead to a contact (and optionally a deal) transfers ownership, watchers and relevant data and emits a `lead_converted` event.

3.5 `ticket` – Support work item with assignment & watchers

Purpose & Scope
`ticket` represents a customer support case. Tickets are stage-agnostic in this CRM but live in a **support queue** defined by `tenant_group_shadow` and can be assigned to agents. Tickets already have participants (requester, CC, follower) via `ticket_participant`. This change ensures alignment with the ownership/assignment/watchers model and introduces watchers via `record_watcher` for unified visibility.

Required Schema Changes

The existing `ticket` table (defined in `002_support_domain_schema.sql`) already includes:

- `assigned_user_id`, `assigned_group_id` – for assignment.
- `requester_contact_id` – the customer raising the ticket.
- `status`, `priority`, `ai_status`, etc.

To align with the generic model:

- Rename `assigned_user_id` / `assigned_group_id` fields (if names differ) to `assigned_to_user_id` / `assigned_to_group_id` for consistency across domains. Add a constraint to ensure only one is non-null at a time.
- Add `owned_by_user_id` and `owned_by_group_id` (both nullable) if tickets need a long-term steward distinct from the current assignee. In many support contexts, the owning group is the queue (`support_queue`), and assignment reflects the current agent. If ownership is not required, these fields may remain null.

Indexes and Constraints

- Ensure `(tenant_id, assigned_to_user_id)` and `(tenant_id, assigned_to_group_id)` indices exist for queue and agent views. A check constraint prevents both assignment fields being non-null simultaneously.
- If ownership fields are added, mirror constraints for ownership similar to other entities.

API Changes

- **Create Ticket:** Must specify at least an assignee group or user; if both are null, the ticket goes to a default queue or is considered unassigned. Optionally set owner fields if ownership is used.
- **Update Ticket:** Changing assignment updates watchers and emits `assignment_changed` events. If ownership fields exist, provide API calls to set them.
- **Participants (Contact/Agent Watchers):** Continue to use `ticket_participant` for requester/CC/follower roles. Additionally, unify watchers by adding entries to `record_watcher` when participants are added and removing them when participants are removed.

Behavioural Rules & Lifecycle Constraints

- **Assignee vs Owner:** An owner (user or group) may represent the long-term responsibility for a ticket; the assignee is the current agent or AI worker working the case. Ownership may remain at

the queue level (support queue) while assignments change. Only one assignee field may be non-null at a time.

- **Watchers and Participants:** `ticket_participant` remains the source of truth for contact and agent participants. The `record_watcher` table should mirror participants for unified notifications. When a participant is added, insert a `record_watcher` row; when removed, remove the watcher row.
- **AI Integration:** Tickets have `ai_status` fields indicating AI posture. When an AI worker takes a ticket, set `assigned_to_user_id` to null and `assigned_to_group_id` to the AI group or leave assignment fields null while the AI status indicates AI ownership. Once the AI completes, assignment may revert to a user.

Event Emission Requirements

- `assignment_changed` events when `assigned_to_user_id` or `assigned_to_group_id` changes.
- `watcher_added` and `watcher_removed` events when participants or watchers change.
- Other existing ticket events (creation, update, resolution) remain. Include AI status changes when applicable.

Automation Integration

- Actions for tickets can be scoped by pipeline (if future support pipelines exist) or by ticket entity. Use triggers such as `ON_RECORD_CREATED`, `ON_ASSIGNMENT_CHANGED`, `ON_TICKET_RESOLVED`.
- Dwell timers can be implemented using Flowable to manage SLAs (time in queue, time since last response). When a timer expires, emit an `ON_DWELL` event that triggers escalation actions.

Acceptance Criteria

1. Tickets must always have an assignee (user or group) or be explicitly marked as unassigned. Attempting to set both assignment fields returns an error.
2. If ownership fields are used, they follow the same single-field rules. Otherwise, they remain null and the owning group is inferred from the queue.
3. Adding a participant (requester, CC or follower) inserts a watcher row and emits a `watcher_added` event. Removing a participant emits a `watcher_removed` event.
4. Changing assignment updates watchers and emits an `assignment_changed` event.

3.6 `activity` – Unified interactions and tasks with ownership, assignment & details

Purpose & Scope

`activity` captures both logged interactions (calls, emails, meetings, notes) and actionable tasks (to-dos, reminders) across CRM entities. It must support owner/creator, assignment, due dates, status and type-specific metadata without introducing a separate table per activity type. Activities can be associated with any record (contact, company, deal, lead, ticket) via existing association tables.

Required Schema Changes

Add or modify the `activity` table to include:

Column	Type/Constraint	Description
`activity_type`	ENUM (e.g. `call`, `email`, `meeting`, `note`, `task`) **NOT NULL**	Identifies the category activity. Use a controlle
`activity_at`	TIMESTAMPTZ **NOT NULL**	The actual date/time th occurred (for logs) or is (for tasks).
`created_by_user_id`	UUID **NOT NULL**	User who created the a References `tenant_user_shado`
`assigned_to_user_id`	UUID **NULLABLE**	For task-type activities, responsible for comple
`assigned_to_group_id`	UUID **NULLABLE**	For queue-based task a Only one of the assign may be non-null.
`status`	ENUM (`pending`, `in_progress`, `completed`, `overdue`, `cancelled`) **NOT NULL DEFAULT 'pending' `| Execution status for tasks. For logged interactions, status can remain` 'completed'`.	
`details_json`	JSONB **NULLABLE**	Stores type-specific de duration, outcome, me location, task due date

Existing fields (id, tenant_id, type/subject, content/body, created_at, updated_at) remain; rename or adjust as needed to align with these fields.

Indexes and Constraints

- Check constraint enforcing that only one of `assigned_to_user_id` or `assigned_to_group_id` is non-null.
- Composite foreign keys `(created_by_user_id, tenant_id)`, `(assigned_to_user_id, tenant_id)`, `(assigned_to_group_id, tenant_id)` ensure tenant-safe references.
- Indexes on `(tenant_id, created_by_user_id)`, `(tenant_id, assigned_to_user_id)`, `(tenant_id, assigned_to_group_id)` to support lists of activities by creator or assignee.
- Index on `(tenant_id, activity_type)` to support dashboards by type.

API Changes

- **Create Activity:** Accept `activity_type`, `activity_at`, `details_json`, `assigned_to_user_id` or `assigned_to_group_id` (for tasks) and associations (e.g. `entity_type`, `entity_id`). `created_by_user_id` is derived from the authenticated user. Validate single assignment field and allowed `activity_type` values.

- **Update Activity:** Allow updating `activity_at`, `details_json`, `assigned_to_user_id`, `assigned_to_group_id` and `status`. Changing `activity_type` may be restricted depending on the type. Setting `status` to `'completed'` or `'cancelled'` closes the task.

Behavioural Rules & Lifecycle Constraints

- **Logged vs Actionable Activities:**
- For logged interactions (call, email, meeting, note), `assigned_to_*` should be null and `status` should be `'completed'`. Activities are immutable; only `details_json` may be updated for corrections.
- For tasks (task), `assigned_to_*` must be set and `status` begins as `'pending'`. The assignee or automation updates `status` to `'in_progress'` or `'completed'`. Overdue tasks can be detected by comparing `activity_at` (due date) with the current date.
- **Ownership vs Assignment:** The creator is the owner (steward) of the log; tasks may have a separate assignee. Ownership does not automatically assign the task; assignment is explicit.
- **Watchers:** Use `record_watcher` for users who should be notified of changes. The `created_by_user_id` and `assigned_to_*` fields automatically become watchers. Additional watchers may subscribe manually.
- **Associations:** Activities link to other records via an association table (not part of this change). Each association indicates which record the activity relates to (e.g. contact, deal, ticket). Multiple associations are allowed.

Event Emission Requirements

- Emit `activity_created`, `activity_updated`, `activity_assigned` and `activity_completed` events as appropriate. Payloads should include `activityId`, `activityType`, `entityType`, `entityId`, `createdByUserId`, `assignedToUserId`, `assignedToGroupId`, `status`, and `detailsJson` changes.
- Assignment changes and status changes should emit `assignment_changed` and `status_changed` events respectively.
- Watcher events are emitted via `record_watcher` as described in Section 5.

Automation Integration

- `ON_RECORD_CREATED` triggers can start workflows when new tasks are created (e.g. follow-up tasks after a call). `ON_RECORD_UPDATED` triggers can run when status changes. Additional triggers like `ON_OVERDUE` can be implemented via dwell timers similar to stage dwell, based on `activity_at` and `status`.
- AI workers may generate activities (e.g. summarise a call transcript) via `AIWORKER` actions.

Acceptance Criteria

1. Activities can be created for any supported `activity_type`; logged interactions are immutable after creation except for minor corrections, whereas tasks can have assignment and status changes.
2. Creating a task requires exactly one of `assigned_to_user_id` or `assigned_to_group_id`. Creating a logged interaction requires both assignment fields to be null.
3. Changing assignment or status updates watchers, emits events and respects idempotency.

4. `details_json` may store arbitrary key/value pairs as long as they fit JSON and remain within a configured schema (optional JSON schema validation may be added later).

---

4. Lists

4.1 `list` – Object-typed lists with lifecycle flags

Purpose & Scope
`list` stores named collections of CRM records (contacts, companies or deals) used for segmentation, marketing, workflow inputs and saved filters. Lists can be static (manually managed) or dynamic (evaluated by automation). This change introduces object typing, processing type and archival flags.

Required Schema Changes

Alter `list` as follows:

Column	Type/Constraint	Description
`object_type`	ENUM(`contact`, `company`, `deal`) **NOT NULL**	Specifies the type of record stored in this list.
`processing_type`	ENUM(`static`, `dynamic`) **NOT NULL DEFAULT 'static' `| Indicates how membership is maintained.` static `lists are manually updated;` dynamic` lists are recomputed based on criteria (phase-2).	
`is_archived`	BOOLEAN **NOT NULL DEFAULT FALSE**	Whether the list is archived. Archived lists are hidden from normal selection but preserved for history.
`criteria_json`	JSONB **NULLABLE**	For future dynamic lists: JSON representation of the criteria. Ignored when `processing_type='static'`. Not required in v1 but defined here for forward compatibility.

Existing fields (`id`, `tenant_id`, `name`, `created_at`, `updated_at`, etc.) remain. `name` remains unique per tenant.

Indexes and Constraints

- Add a unique index `(tenant_id, object_type, name)` to prevent name collisions across different object types if desired. Alternatively keep the current `(tenant_id, name)` uniqueness and handle type collisions in the application.
- Add an index on `(tenant_id, object_type, is_archived)` for list selection.
- When `processing_type='dynamic'`, require that `criteria_json` is not null (Phase 2). A check constraint can enforce this once dynamic lists are supported.

API Changes

- **Create/Update List:** Accept `object_type`, `processing_type`, `is_archived` and optionally `criteria_json`. Reject invalid combinations (e.g. dynamic list without `criteria_json`).
- **Archive/Unarchive List:** Provide endpoints to set `is_archived`. Archived lists remain in the database but are hidden from default queries. Membership remains but is no longer used in workflows unless explicitly referenced.
- **Rename & Delete:** Allow renaming lists; deleting a list with memberships should either remove memberships or archive the list. Hard deletion may only be permitted when no workflows reference the list.

Behavioural Rules & Lifecycle Constraints

- **Object Type Enforcement:** A list may only contain memberships for records of the same `object_type`. The application must enforce this when adding or removing members.
- **Static Lists:** Membership is maintained manually via `list_membership`. No automatic evaluation occurs. Users or automation may add or remove members.
- **Dynamic Lists (Phase 2):** When `processing_type='dynamic'`, membership is derived from `criteria_json`. Evaluation occurs via a scheduled or event-driven process (Flowable or dedicated worker). Dynamic lists are read-only; manual membership changes are not permitted. This functionality is deferred to a later phase; for now, dynamic lists may be created but not evaluated.

Event Emission Requirements

- `list_created`, `list_updated`, `list_archived` events containing `listId`, `tenantId`, `objectType`, `processingType`, `isArchived` and, when applicable, `criteriaJson`.
- Membership events are handled via `list_membership` (see Section 4.2).

Automation Integration

- **List Actions:** Actions scoped to `LIST` can fire on `ON_MEMBER_ADDED` and `ON_MEMBER_REMOVED` events (see triggers in Section 1). For example, when a contact is added to the “High Intent Leads” list, create a task for the sales team. For dynamic lists, `ON_MEMBER_ADDED` events are generated by the evaluation process.
- **Dynamic Evaluation (Phase 2):** When dynamic lists are evaluated, membership changes must generate `ON_MEMBER_ADDED` or `ON_MEMBER_REMOVED` events to drive automation.

Acceptance Criteria

1. Creating a list requires specifying `object_type`. Leaving `object_type` null results in a validation error.
2. Setting `processing_type` to `'dynamic'` without `criteria_json` returns a validation error. In v1, the API may reject dynamic lists entirely until evaluation is available.
3. Archiving a list via the API sets `is_archived=true`, hides the list from default queries and emits a `list_archived` event.
4. Lists enforce object type: attempts to add a membership for a record of a different type are rejected.

4.2 `list_membership` – Membership tracking with timestamps

Purpose & Scope
`list_membership` records membership of a record in a list. Membership is used for segmentation, automation triggers and analytics. This change adds a timestamp column to support membership growth and churn analysis.

Required Schema Changes

Alter `list_membership` to include:

Column	Type/Constraint	Description
`created_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Timestamp when the membership was created.
`removed_at`	TIMESTAMPTZ **NULLABLE**	Timestamp when the membership was removed. Null indicates active membership.

Existing columns (`tenant_id`, `list_id`, `record_id`) remain, where `record_id` references the appropriate object type based on `list.object_type`. Ensure `list_id` references `list(id)` and is tenant-scoped.

Indexes and Constraints

- **Composite Primary Key:** `(list_id, record_id)` remains. Add `tenant_id` if not present to enforce tenant isolation.
- **Unique Active Membership:** A partial unique index ensures only one active membership per record per list: `(list_id, record_id)` where `removed_at IS NULL`.
- **Foreign Keys:** Ensure `record_id` references the correct table based on `list.object_type`. This may be implemented via separate membership tables per object or a polymorphic foreign key constraint at the application layer.

API Changes

- **Add Membership:** Create a row with `created_at=NOW()` and `removed_at=NULL`. If an active membership exists, the API must return a conflict error (or update `created_at` if idempotent). Trigger `ON_MEMBER_ADDED` actions.
- **Remove Membership:** Set `removed_at=NOW()` on the existing active membership row. Trigger `ON_MEMBER_REMOVED` actions. Do not delete the row so that membership history is preserved.
- **List Membership:** Provide endpoints to list current members (`removed_at IS NULL`) and membership history for a list.

Behavioural Rules & Lifecycle Constraints

- **One Active Membership:** A record may be active in a list at most once. Attempting to add an existing member must result in either a no-op or an update to `created_at` depending on idempotency policy.
- **Soft Removal:** When a record is removed from a list, mark `removed_at`. Do not delete the row; this preserves history for analytics and potential re-add logic.
- **Dynamic Lists:** For dynamic lists (Phase 2), membership rows should be inserted and removed automatically by the evaluation process. Manual updates are not allowed.

Event Emission Requirements

- Insertions with `removed_at=NULL` must emit `list_member_added` or reuse the generic `watcher_added` pattern if not kept separate. Include `listId`, `recordId`, `tenantId`, `createdAt` in the payload.
- Setting `removed_at` must emit `list_member_removed` with similar payload plus `removedAt`.

Automation Integration

- Triggers `ON_MEMBER_ADDED` and `ON_MEMBER_REMOVED` rely on `list_membership` inserts and updates. The automation engine listens for these events and executes `LIST`-scoped actions accordingly.
- When re-adding a previously removed member, treat it as a new membership and emit a new `ON_MEMBER_ADDED` event.

Acceptance Criteria

1. Adding a record to a list creates a membership row with `created_at`. If a row exists with `removed_at` set, update it by setting `removed_at=NULL` and `created_at=NOW()` to reflect the new membership.
2. Removing a record sets `removed_at` and triggers `ON_MEMBER_REMOVED`. The row remains for history.
3. Dynamic lists (when implemented) insert and remove memberships via the evaluation process, updating `created_at` and `removed_at` accordingly.

---

5. Visibility & Watchers

5.1 `record_watcher` – Unified watchers table for all domains

Purpose & Scope

`record_watcher` implements a universal mechanism for users and groups to "follow" records across all CRM entities (contact, company, deal, lead, ticket, activity and others). Watchers receive notifications and have read visibility but do not own or assign the record. This replaces per-domain watcher arrays or tables and unifies behaviour across modules. Ticket participants (requester, CC, follower) remain in `ticket_participant` but are mirrored here for unified notifications.

Schema Definition

Create `record_watcher` with these columns:

Column	Type/Constraint	De
`tenant_id`	UUID **NOT NULL**	Te
`record_type`	ENUM(`contact` , `company` , `deal` , `lead` , `ticket` , `activity` , `pipeline` , `list` , …) **NOT NULL**	Ty
`record_id`	UUID **NOT NULL**	ID Mu tab
`principal_type`	ENUM(`user` , `group`) **NOT NULL**	Ind ind
`principal_id`	UUID **NOT NULL**	ID gro `te` `te` co pr
`created_at`	TIMESTAMPTZ **NOT NULL DEFAULT NOW()**	Tir ad
`created_by_user_id`	UUID **NULLABLE**	Us be Re `te`

Indexes and Constraints

- Primary key on `(tenant_id, record_type, record_id, principal_type, principal_id)` prevents duplicate watchers.
- Composite foreign keys:

- `(tenant_id, principal_id)` referencing `tenant_user_shadow` when `principal_type='user'`.
- `(tenant_id, principal_id)` referencing `tenant_group_shadow` when `principal_type='group'`.
- Optional index on `(tenant_id, principal_type, principal_id)` to retrieve all records watched by a user or group.
- Index on `(tenant_id, record_type, record_id)` to list watchers of a record.

API Changes

- **Add Watcher:** Provide endpoints to add a user or group as a watcher to a record. Validate that the user/group belongs to the same tenant. Setting a record's owner or assignee automatically adds them as a watcher (handled in application logic).
- **Remove Watcher:** Remove the watcher row. Removing ownership or assignment must also remove the watcher relationship (owner/assignee watchers are automatic).
- **List Watchers:** List all users and groups watching a record, and optionally list all records watched by a principal.

Behavioural Rules

- **Automatic Watchers:** The following principal types automatically become watchers:
- Owners (`owned_by_user_id` or `owned_by_group_id`)
- Assignees (`assigned_to_user_id` or `assigned_to_group_id`)
- Creators (e.g. `created_by_user_id` for activities)
- Ticket participants (`ticket_participant`) for tickets. When a participant is added to a ticket, insert or update a `record_watcher` row. When removed, remove the watcher row.
- **Manual Watchers:** Any user or group may be added manually as a watcher to receive notifications or follow the record. Manual watchers may be removed without affecting ownership or assignment.
- **Propagation on Ownership/Assignment Change:** When ownership or assignment changes, update watchers accordingly: remove the old owner/assignee watcher and add the new owner/assignee watcher.
- **Visibility & Permissions:** The presence of a watcher does not grant edit rights; it only subscribes the principal to notifications and ensures the record appears in their watched list. Permissions continue to be governed by role-based access control.

Event Emission Requirements

- `watcher_added` and `watcher_removed` events are emitted whenever a row is inserted into or deleted from `record_watcher`. Payload includes `tenantId`, `recordType`, `recordId`, `principalType`, `principalId`, and `createdByUserId`. These events trigger notifications and maintain caches.

Automation Integration

- Watcher events can drive automation (e.g. send a welcome message to watchers when they subscribe). Actions targeting `watcher_added` and `watcher_removed` can be defined with scope `ENTITY` and trigger `ON_WATCHER_ADDED` or `ON_WATCHER_REMOVED` if such triggers are introduced in a later phase.

- Automatic watchers should not trigger watcher actions unless explicitly defined.

Acceptance Criteria

1. Adding a watcher inserts a new row in `record_watcher` and emits a `watcher_added` event. Adding the same watcher again yields an idempotent result (no duplicate rows).
2. Removing a watcher deletes the corresponding row and emits a `watcher_removed` event. Removing a non-existent watcher is a no-op.
3. Ownership, assignment, creation and ticket participant operations automatically add or remove watcher rows. The presence of watchers aligns with the current owner, assignee and participants at all times.
4. Listing watchers of a record returns both automatic and manually added watchers with their principal type and creation timestamp.

---

6. Summary of Decision Points & Deferred Features

This consolidated change request captures all mandatory and optional requirements from the three source documents. Key decisions include:

- **Generic Automation Framework:** A shared `automation_action` / `automation_action_execution` framework replaces per-domain action tables. It supports pipeline and list automation with additive and override behaviours.
- **Multi-pipeline Support:** Pipelines now have `object_type`, `display_order`, `is_active` and `key`. Stage definitions include probability, stage type and terminal flags. Stage history is recorded in a dedicated table and emits `stage_changed` events.
- **Ownership, Assignment & Watchers:** Relationship records (contact, company, deal, lead) gain owner and assignment fields. Work items (ticket, activity) gain assignment fields and optionally owner fields. A universal `record_watcher` table manages watchers across all entities.
- **List Enhancements:** Lists are typed, have processing and archival flags, and will support dynamic criteria in a future phase. List membership includes timestamps and triggers `ON_MEMBER_ADDED` and `ON_MEMBER_REMOVED` events.
- **Event-Driven Architecture:** Consistent event emission across ownership changes, assignments, stage transitions, list membership and watcher changes enables Flowable and AI workers to orchestrate workflows. Dwell timers are implemented via Flowable timers and triggered by `stage_changed` events.
- **Deferred Features:** Dynamic lists, advanced composition modes for stage actions, full RACI modelling, advanced permissions and live reporting engines are deferred to future phases. They are documented for context but excluded from the immediate implementation scope.

Each domain section above provides specific schema and behavioural requirements along with acceptance criteria. Engineering teams should implement these changes holistically, ensuring that APIs, database migrations, event producers and automation engines are updated accordingly.
