ChatGPT

Consolidated CRM Change Request – File Impact Guide

This guide lists the **code artifacts** that must be updated or created to align the CRM service with the schema changes. Each entry is grouped by the impacted domain object and provides traceability to both the SQL change and the current implementation. New files are marked with **[NEW FILE]**. Only files evidenced in the existing repository are referenced; no new layers or patterns are introduced.

Record watcher (new domain object)

The SQL introduces a new `record_watcher` table to allow users or groups to subscribe to changes on any record. There is no existing watcher implementation, so analogous files from the list-membership domain are used as templates.

File path	Responsibility (current or template)	Required update / creation rationale	SQL
`app/domain/models/record_watcher.py` **[NEW FILE]**	Define the SQLAlchemy model for watchers. The template is the `ListMembership` model, which links lists to CRM records and declares columns for `list_id`, `member_id`, `member_type` and `created_by`.	Create a model with fields `tenant_id`, `record_type` (`crm_record_type`), `record_id`, `principal_type` (`principal_type`), `principal_id`, `created_at` and `created_by_user_id`. Use a composite primary key on (tenant_id, record_type, record_id, principal_type, principal_id) and indexes mirroring those defined in SQL.	Crea `rec` tabl inde

File path	Responsibility (current or template)	Required update / creation rationale	SQL
`app/domain/schemas/record_watcher.py` **[NEW FILE]**	Pydantic schemas define create/read models for list memberships. The `ListMembershipRead` schema (in the existing code) exposes `id`, `list_id`, `member_id`, `member_type` and timestamps.	Create `RecordWatcherCreate` and `RecordWatcherRead` schemas with fields corresponding to the new model. The read schema should include the composite key fields, `created_at` and optionally `created_by_user_id`.	Alig stru `rec` tabl
`app/domain/services/record_watcher_service.py` **[NEW FILE]**	Service layers manage list memberships with functions to list, create and delete memberships.	Implement analogous service functions: list watchers by record or principal, create a watcher (ensuring the referenced record exists and inserting into the new table) and delete a watcher. Use `commit_or_raise` and publish events via a producer.	Enal ope reco tabl
`app/domain/schemas/events/record_watcher_event.py` **[NEW FILE]**	The list membership domain defines event schemas for creation and deletion.	Create a delta or snapshot event schema for watcher events to convey created or deleted watchers.	Req ever inte wat add
`app/messaging/producers/record_watcher_producer.py` **[NEW FILE]**	Producers publish domain events; `ListMembershipMessageProducer` demonstrates this pattern.	Create a Kafka/AMQP producer to send `record_watcher` created and deleted events after successful commits.	Allo serv wat
`app/api/routes/record_watchers_admin_route.py` **[NEW FILE]** and `app/api/routes/record_watchers_tenant_route.py` **[NEW FILE]**	Admin routes for list memberships expose collection and singleton endpoints under `/admin/lists/{list_id}/memberships` and `/admin/memberships/{membership_id}`.	Provide REST endpoints to list, create and delete watchers. Admin routes should support cross-tenant queries with optional `tenant_id`. Tenant routes should scope by tenant and record. Use `record_watcher_service` for business logic and Pydantic schemas for validation.	Expo func new `rec` tabl

Automation action (new domain object)

Declarative automation rules require a new domain object. The design mirrors the **support macro** domain, which stores a set of actions and an `is_active` flag and uses a dedicated service and producer.

File path	Responsibility (current or template)	Required update / creation rationale	SQL drivers
`app/domain/models/automation_action.py` **[NEW FILE]**	The `SupportMacro` model includes fields like `id`, `tenant_id`, `name`, `description`, `is_active` and `actions`.	Create a model matching the `automation_action` table: fields for `id`, `tenant_id`, `entity_type` (`crm_record_type`), `scope_type` (`automation_scope_type`), optional `record_type` / `record_id`, optional `pipeline_id`, `pipeline_stage_id`, `list_id`, `trigger_event`, `condition_json`, `action_type` (`automation_action_type`), `config_json`, `priority`, `enabled`, `inherit_pipeline_actions`, timestamps and audit fields. Apply the unique constraint on `(tenant_id, id)` and the scope check constraint; define foreign keys to pipeline, pipeline stage and list.	`automation_a` table definition a constraints.
`app/domain/schemas/automation_action.py` **[NEW FILE]**	Pydantic schemas for support macros define create, update and read representations with validation.	Provide `AutomationActionCreate`, `AutomationActionUpdate` and `AutomationActionRead` schemas. The create/update schemas should enforce scope rules at the application layer (exactly one target column set) and default values for `priority`, `enabled` and `inherit_pipeline_actions`.	Schemas map th structure of the automation_acti table.

File path	Responsibility (current or template)	Required update / creation rationale	SQL drivers
`app/domain/services/automation_action_service.py` **[NEW FILE]**	The support macro service handles listing, creating, updating and deleting macros, publishes events and enforces uniqueness.	Implement analogous service functions: list actions with optional filters (e.g., by entity_type or scope_type), create a new action (ensuring the scope constraint), update existing actions, delete actions and publish events. Use `commit_or_raise` and return snapshots or deltas.	Provides application-layer enforcement for `automation_a` table.
`app/domain/schemas/events/automation_action_event.py` **[NEW FILE]**	Support macros define event deltas for changed base fields.	Define event payloads for `automation_action` created, updated and deleted events. Include the full snapshot and any changed fields.	Enables other se to respond to automation rule changes.
`app/messaging/producers/automation_action_producer.py` **[NEW FILE]**	Producers publish support macro events.	Create a producer class to emit `automation_action` lifecycle events after service commits.	Supports event-d automation engi
`app/api/routes/automation_actions_admin_route.py` **[NEW FILE]** and `app/api/routes/automation_actions_tenant_route.py` **[NEW FILE]**	Support macro routes expose collection and singleton endpoints under `/admin/support_macros` and `/tenant/support_macros`.	Implement REST endpoints to list, create, update and delete automation actions. Admin routes should allow cross-tenant listing; tenant routes should scope by tenant and optionally filter by entity_type or scope_type. The routes must validate that exactly one scope target is provided.	Exposes the automation_acti domain to client based on the SQ specification.

Automation action execution (new domain object)

An execution log for automation rules must be captured. There is no direct analogue, but the `ticket_audit` and `ticket_time_entry` models demonstrate patterns for append-only logging tables with composite foreign keys and status fields.

File path	Responsibility (current or template)	Required update / creation rationale	SQL d
`app/domain/models/automation_action_execution.py` **[NEW FILE]**	Ticket audit model records immutable events with composite foreign keys and a check constraint on actor type.	Define a model with fields `id`, `tenant_id`, `action_id`, `entity_type` (`crm_record_type`), `entity_id`, optional `pipeline_id`, `from_stage_id`, `to_stage_id`, `list_id`, `trigger_event`, `execution_key`, `status` (`action_execution_status`), `response_code`, `response_body`, `error_message`, `triggered_at`, `started_at`, `completed_at` and `created_at`. Add a unique constraint on `(tenant_id, execution_key)` and a foreign key to `(tenant_id, action_id)` with cascade delete. Include indexes on `(tenant_id, action_id, status)` and `(tenant_id, entity_type, entity_id)`.	`auto` table
`app/domain/schemas/automation_action_execution.py` **[NEW FILE]**	Pydantic schemas exist for `ticket_audit` and `ticket_time_entry` events.	Provide schemas for creating and reading execution logs. Execution records are generally created by the automation engine, so creation might be restricted to internal calls.	Enable logs.
`app/domain/services/automation_action_execution_service.py` **[NEW FILE]**	There is no existing equivalent, but a service is needed to write execution logs and query status.	Implement functions to create execution records (with status = `PENDING`), update status to `IN_PROGRESS`, `SUCCEEDED` or `FAILED`, and query logs by action, status or entity. Use `commit_or_raise`.	Suppo for au

File path	Responsibility (current or template)	Required update / creation rationale	SQL d
`app/domain/schemas/events/automation_action_execution_event.py` **[NEW FILE]**	Event schemas for ticket audits exist.	Define event payloads for execution status changes if downstream systems need to react (e.g., for monitoring dashboards).	Optio event
`app/messaging/producers/automation_action_execution_producer.py` **[NEW FILE]**	Producers send ticket audit events.	Create a producer to emit execution status events after status transitions.	Enable of aut
Routes	The current API does not expose ticket audit or time entry creation through routes. Execution logs are internal to the automation engine.	No external endpoints are required; the execution service will likely be called from an internal task.	Execu conce

Stage history (new domain object)

The `stage_history` table records stage changes for any stage-based entity. It resembles the `ticket_time_entry` and `ticket_audit` log tables.

File path	Responsibility (current or template)	Required update / creation rationale	SQL drivers
`app/domain/models/stage_history.py` **[NEW FILE]**	The `TicketTimeEntry` model logs work done on a ticket and defines fields such as `tenant_id`, `ticket_id`, `minutes_spent`, `started_at` and `created_at`.	Create a model with fields `id`, `tenant_id`, `entity_type`, `entity_id`, optional `pipeline_id`, `from_stage_id`, `to_stage_id`, `changed_at`, `changed_by_user_id` and `source`. Define foreign keys to pipeline and pipeline stages with `SET NULL` on delete and indexes on `(tenant_id, entity_type, entity_id)` and `(tenant_id, pipeline_id)`.	`stage_history` table definition.
`app/domain/schemas/stage_history.py` **[NEW FILE]**	Pydantic schemas exist for `ticket_time_entry` and `ticket_audit` events.	Provide create and read schemas for stage history records. The read schema should expose all columns.	Exposes stage history to consumers.
`app/domain/services/stage_history_service.py` **[NEW FILE]**	There is no existing equivalent; a service is needed to insert history rows and query them.	Implement functions to record stage transitions (called by pipeline stage update logic) and to list history by entity or pipeline. Use `commit_or_raise`.	Supports writing to and reading from the stage_history table.
`app/domain/schemas/events/stage_history_event.py` **[NEW FILE]**	Event schemas exist for ticket_time_entry and ticket_audit changes.	Define event payloads for new stage history entries.	Enables event-driven consumers to react to stage changes.
`app/messaging/producers/stage_history_producer.py` **[NEW FILE]**	Producers publish ticket_time_entry events.	Create a producer to emit stage history events after successful inserts.	Supports asynchronous handling of stage transitions.

File path	Responsibility (current or template)	Required update / creation rationale	SQL drivers
`app/api/routes/stage_history_tenant_route.py` **[NEW FILE]**	There is no current route for stage history; however, listing logs is analogous to ticket audit or time entry retrieval.	Provide a read-only route for tenants to fetch stage history for a given entity (e.g., `/tenant/deals/{deal_id}/stage-history`). Only GET operations are needed; inserts happen through business logic in the service layer.	Exposes stage change history to API clients.

Pipeline (existing domain object)

The SQL adds multiple fields to the `pipeline` table and new constraints. Existing files handle only the `name` field; they must be extended.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/pipeline.py`	Defines the `Pipeline` ORM with columns `id`, `tenant_id`, `name`, `created_at`, `updated_at`, `created_by` and `updated_by`. It currently omits all new columns.	Add mapped columns for `object_type` (`pipeline_object_type`), `display_order` (integer), `is_active` (boolean default `True`), `pipeline_key` (string), and `movement_mode` (`pipeline_movement_mode`) with corresponding defaults. Update `__table_args__` to include the unique constraints on `(tenant_id, object_type, pipeline_key)` and `(tenant_id, object_type, display_order)` and the index on `(tenant_id, object_type)`.	Schema alterations to `pipeline`.

File path	Current responsibility	Required update	SQL drivers
`app/domain/schemas/pipeline.py`	Pydantic schemas define `PipelineBase` with only a `name` field and create/update/read models that expose `id`, `tenant_id` and timestamps.	Extend the base, create and update schemas to include the new columns (`object_type`, `display_order`, `is_active`, `pipeline_key`, `movement_mode`). The read schema should expose these fields as read-only.	New columns require schema support.
`app/domain/services/pipeline_service.py`	Provides CRUD operations for pipelines. `create_pipeline` builds a `Pipeline` with only a `name` and emits events; `update_pipeline` updates only the `name`.	Update creation and update functions to accept and persist the new fields. When creating a pipeline, assign defaults for `is_active` and compute `pipeline_key` if not supplied. Ensure that `display_order` is set in a tenant-safe manner (e.g., by counting existing pipelines of the same object_type). Include the new fields in snapshots and deltas.	Pipeline enhancements.
`app/api/routes/pipelines_admin_route.py` and `app/api/routes/pipelines_tenant_route.py`	Expose endpoints to list, create, update and delete pipelines. They currently rely on schemas that accept only `name`.	Update request models used in POST and PATCH endpoints to include the new fields. Adjust query parameters to allow filtering by `object_type` and `is_active`. Ensure that the response model exposes the new fields.	New pipeline properties and filtering capabilities.
`app/domain/schemas/events/pipeline_event.py` and `app/messaging/producers/pipeline_producer.py`	Define and publish pipeline created/updated/deleted events. Existing events include only `id`, `tenant_id` and `name`.	Extend the event payload to include `object_type`, `display_order`, `is_active`, `pipeline_key` and `movement_mode` so downstream services receive full context.	New columns must be propagated to events.

Pipeline stage (existing domain object)

The `pipeline_stage` table adds new columns and renames an existing column. Current code still uses `stage_order` and lacks the new fields.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/pipeline_stage.py`	Defines the `PipelineStage` ORM with `id`, `tenant_id`, `pipeline_id`, `name`, `stage_order` and `probability`. It enforces uniqueness on `(pipeline_id, stage_order)`.	Rename the `stage_order` attribute and database column to `display_order`. Add `stage_state` (enum) with default `NOT_STARTED` and `inherit_pipeline_actions` (boolean) default `True`. Add a `CheckConstraint` to ensure that `probability` is either NULL or between 0 and 1. Replace the old unique constraint with one on `(pipeline_id, display_order)`.	Stage enhancements.
`app/domain/schemas/pipeline_stage.py`	Pydantic models describe pipeline stage create, update and read DTOs; they currently accept `name`, `stage_order` and `probability`.	Update schemas to use `display_order` instead of `stage_order`. Add fields for `stage_state` and `inherit_pipeline_actions` with appropriate defaults.	Schema must match the updated table definition.
`app/domain/services/pipeline_stage_service.py`	Provides functions to list, create, update and delete stages. It currently passes `stage_order` and `probability` to the ORM.	Modify creation and update functions to accept `display_order`, `stage_state` and `inherit_pipeline_actions`. Remove references to `stage_order`. Validate the probability range in service logic or rely on DB constraint.	Pipeline stage enhancements.

File path	Current responsibility	Required update	SQL drivers
`app/api/routes/pipeline_stages_admin_route.py` and `app/api/routes/pipeline_stages_tenant_route.py`	Admin routes expose endpoints nested under pipelines for listing, creating, updating and deleting stages.	Update request schemas for stage creation and update to include the new fields. Ensure that the API descriptions mention `display_order`, `stage_state` and `inherit_pipeline_actions`.	Exposes updated stage semantics to clients.
`app/domain/schemas/events/pipeline_stage_event.py` and `app/messaging/producers/pipeline_stage_producer.py`	Publish stage lifecycle events.	Include the new fields in the event payloads.	Stage enhancements require downstream awareness.

List (existing domain object)

The `list` table gains `processing_type` and `is_archived` columns. Lists currently only support `name`, `object_type`, `list_type` and `filter_definition`.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/list.py`	Defines the `List` ORM with fields `id`, `tenant_id`, `name`, `object_type`, `list_type` and `filter_definition`.	Add `processing_type` (`list_processing_type`) with default `STATIC` and `is_archived` (boolean) default `False` as mapped columns. These columns should be included in `__table_args__` if any new constraints apply.	List enhancements.
`app/domain/schemas/list.py`	Pydantic schemas expose existing list fields for create, update and read.	Update create and update schemas to accept `processing_type` and `is_archived`. The read schema should expose these fields.	New list properties need to be reflected in the API.

File path	Current responsibility	Required update	SQL drivers
`app/domain/services/list_service.py`	Service functions create and update lists by populating `name`, `object_type`, `list_type` and `filter_definition`.	Modify create and update functions to accept and persist `processing_type` and `is_archived`. Snapshot/delta logic must include these new fields so events contain them.	List enhancements.
`app/api/routes/lists_admin_route.py` and `app/api/routes/lists_tenant_route.py`	Routes expose endpoints for listing, creating, updating and deleting lists. They currently accept schemas without `processing_type` or `is_archived`.	Update request models to include the new fields and add optional query parameters to filter by `processing_type` and `is_archived`.	Exposes new list semantics to clients.
`app/domain/schemas/events/list_event.py` and `app/messaging/producers/list_producer.py`	Publish list lifecycle events.	Include the new fields in event payloads so consumers are aware of list processing mode and archiving status.	Events must reflect the updated list model.

List membership (existing domain object)

The change set normalizes `member_type` to the new `list_object_type` enum and adds indexes for typed queries. Current code treats `member_type` as a simple string.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/list_membership.py`	Defines memberships with `id`, `list_id`, `member_id`, `member_type` (string) and `created_by`.	Change the type of `member_type` to use the `list_object_type` enum. No additional columns are needed.	Enum conversion for member_type.
`app/domain/schemas/list_membership.py`	Schemas use string fields for `member_type`.	Update schemas to use an enum or restrict values to those defined in `list_object_type`.	Aligns with the enum conversion.

File path	Current responsibility	Required update	SQL drivers
`app/domain/services/list_membership_service.py`	Manages memberships by inserting `list_id`, `member_id` and `member_type`.	Ensure that `member_type` values correspond to the enum; validation may be added in the service layer.	Enum conversion requires validation.
`app/api/routes/list_memberships_admin_route.py` and `app/api/routes/list_memberships_tenant_route.py`	Routes expose membership management.	Adjust request schemas to enforce the enum type for `member_type`. Response models automatically include the enum value via the updated schemas.	Clients must use the new enumeration.
Indexes	No code changes but relevant for performance.	The migration adds an index on `(list_id, member_type)` and on `(tenant_id, object_type)` for lists; ensure that query patterns in services leverage these indexes where appropriate.	Enhances query performance.

Contact (existing domain object)

Contacts gain ownership fields referencing tenant users and groups. The model currently lacks these columns.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/contact.py`	Defines a contact with fields such as `id`, `tenant_id`, `first_name`, `middle_name`, `last_name`, `job_title`, `created_at` and `updated_at`.	Add `owned_by_user_id` and `owned_by_group_id` as nullable UUID columns. Define foreign keys to `(tenant_id, user_id)` and `(tenant_id, id)` in the `tenant_user_shadow` and `tenant_group_shadow` tables with `SET NULL` on delete and indexes on the owner fields.	Ownership enhancements for contact.
`app/domain/schemas/contact.py`	Pydantic schemas expose existing fields.	Add owner fields to create, update and read schemas. The fields should be optional on create/update and included in read responses.	Expose new ownership information to API clients.
`app/domain/services/contact_service.py`	Provides CRUD and JSON patch operations for contacts; currently does not handle ownership. It constructs a new `Contact` from fields like `first_name`, `middle_name`, `last_name` and timestamps.	Update creation and update logic to accept `owned_by_user_id` and `owned_by_group_id` and persist them on the ORM model. Index these fields when querying if necessary.	SQL adds ownership columns.
`app/api/routes/contacts_admin_route.py` / `contacts_tenant_route.py` and nested routes	Expose endpoints for contact operations.	Update request models to include owner fields and allow clients to set or clear ownership. Response models should return ownership information.	Ownership enhancements.

File path	Current responsibility	Required update	SQL drivers
`app/domain/schemas/events/contact_event.py` and `app/messaging/producers/contact_producer.py`	Publish contact lifecycle events.	Include owner fields in event payloads so consumers receive ownership updates.	New columns must be propagated to events.

Company (existing domain object)

Companies also gain ownership fields. The model currently defines company details but not ownership.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/company.py`	Defines company fields such as `id`, `tenant_id`, `name`, `domain`, `created_at` and `updated_at`.	Add `owned_by_user_id` and `owned_by_group_id` columns with the same foreign keys and indexes as for contacts.	Company ownership enhancements.
`app/domain/schemas/company.py`	Schemas expose existing company fields.	Add owner fields to create, update and read schemas.	Exposes ownership in API.
`app/domain/services/company_service.py`	Manages companies and nested resources; currently handles phone numbers, emails, addresses and relationships.	Update create and update functions to accept and persist `owned_by_user_id` and `owned_by_group_id`. Include these fields in snapshots and deltas so events reflect ownership.	New columns for company.
Routes and events	Companies are managed via admin and tenant routes; events are published via `CompanyMessageProducer`.	Adjust request/response models to include ownership and ensure events include these fields.	Ownership enhancements.

Deal (existing domain object)

Deals receive owner and assignment fields plus categorization and forecasting attributes. The current model lists only basic fields such as `name`, `amount`, `expected_close_date`, `pipeline_id`, `stage_id` and `probability`.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/deal.py`	Defines deals with core fields and relationships to pipelines and stages.	Add nullable columns `owned_by_user_id`, `owned_by_group_id`, `assigned_user_id`, `assigned_group_id`, `deal_type` (`deal_type` enum), `forecast_probability` (numeric) and `close_date` (date). Define foreign keys to tenant user and group shadow tables and add indexes on owner and assignee fields.	Deal enhancements.
`app/domain/schemas/deal.py`	Schemas expose existing deal fields for create, update and read operations.	Add the new fields with appropriate optional types to create and update schemas and include them in read schemas.	Exposes new deal properties to API clients.
`app/domain/services/deal_service.py`	Service functions list, get, create and update deals. `service_create_deal` currently populates name, amount, expected_close_date, pipeline_id, stage_id and probability; `service_update_deal` updates these fields only.	Modify create and update functions to accept and persist owner fields, assignee fields, deal_type, forecast_probability and close_date. Include these properties when computing snapshots/deltas so events include them.	Deal enhancements.
Routes, events and producers	Deal endpoints and events currently reflect existing fields.	Update request and response models to include the new fields. Extend event payloads in `DealMessageProducer` and event schemas to carry ownership, assignment, type and forecasting properties.	Aligns with the extended deal model.

Lead (existing domain object)

Leads gain ownership fields similar to contacts and companies. Current models and services do not include these fields.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/lead.py`	Defines leads with fields like `id`, `tenant_id`, `first_name`, `middle_name`, `last_name`, `source` and `lead_data`.	Add `owned_by_user_id` and `owned_by_group_id` columns with foreign keys to tenant user and group shadow tables and indexes on owner fields.	Lead ownership enhancements.
`app/domain/schemas/lead.py`	Schemas expose current lead fields.	Add owner fields to create, update and read schemas.	Exposes ownership in API.
`app/domain/services/lead_service.py`	Service functions list, create and update leads; creation populates first_name, middle_name, last_name, source and lead_data.	Modify create and update functions to accept `owned_by_user_id` and `owned_by_group_id` and persist them. Update snapshot and event payloads accordingly.	Lead ownership enhancements.
Routes and events	Leads are exposed via admin and tenant routes; events are published via `LeadMessageProducer`.	Update routes to accept owner fields and events to include them.	Ownership enhancements.

Activity (existing domain object)

The `activity` table is extended and a column is renamed. The current model uses a `type` string and lacks the new fields.

File path	Current responsibility	Required update	SQL drivers
`app/domain/models/activity.py`	Defines the `Activity` ORM with fields `id`, `tenant_id`, `type`, `title`, `description`, `due_date`, `status`, `assigned_user_id`, timestamps and audit fields.	Rename the `type` column to `activity_type`. Add new columns `activity_at` (timestamp), `created_by_user_id` (foreign key to tenant_user_shadow), `assigned_group_id` (foreign key to tenant_group_shadow) and `details_json` (JSONB). Define indexes on `assigned_user_id`, `assigned_group_id` and `created_by_user_id` as per the SQL.	Activity enhancements.
`app/domain/schemas/activity.py`	Schemas expose existing activity fields for create, update and read.	Rename the `type` field to `activity_type` and add the new fields to the create/update/read schemas.	Aligns with the renamed and new columns.
`app/domain/services/activity_service.py`	Provides list, create and update functions for activities. The `create_activity` function builds an `Activity` with `type`, `title`, `description`, `due_date`, `status` and `assigned_user_id`.	Update create and update functions to use `activity_type` instead of `type` and accept `activity_at`, `created_by_user_id`, `assigned_group_id` and `details_json`. Persist these fields and update event snapshots/deltas accordingly.	Activity enhancements.
`app/api/routes/activities_admin_route.py` and `app/api/routes/activities_tenant_route.py`	Expose CRUD operations for activities.	Update request and response models to reflect the renamed and new fields. Adjust any filtering logic to include assignment group and creation user filters.	Exposes new activity attributes to clients.

File path	Current responsibility	Required update	SQL drivers
Events and producers	Activity events currently send existing fields.	Include the renamed and new fields in event payloads so downstream systems have full context.	Activity enhancements.

Additional notes

- **Enumerations:** No separate code files exist for the newly added enums. They are stored as Postgres types. However, model and schema changes must ensure that the corresponding fields accept only the allowed values (e.g., by restricting via `Enum` or validating strings). The enumerations and their values are defined in the change set.
- **Migration ordering:** For fields that are both added and converted (e.g., renaming `stage_order` to `display_order` or converting `object_type` to an enum), code changes should be coordinated with migration deployment. Ensure the application reads both column names until the migration is fully applied, or perform a coordinated deploy when the schema is updated.
- **Backward compatibility:** The `pipeline_stage_type` enum is retained for backward compatibility but is not used by the new design. No code changes are required unless legacy API endpoints expose this field.