ChatGPT

FLM Trip Logic in Shipsy (First/Last Mile)

1. FLM Trip Validations and Business Logic

- **Trip Creation (Backend):** Creating a First/Last Mile (FLM) trip requires a valid hub and trip type. The system asserts that `hubId` and `tripType` are provided [1]. If a `trip_reference_number` is given and already exists for that hub, the same Trip is returned (preventing duplicate creation) [2] [3]. Otherwise, a new Trip record is created with status `"created"` and the given type (e.g. `"FLM"`) [1] [4]. After creation, **pre-action validations** run: the Trip data is formatted and checked via `TRIP_CREATE_VALIDATION` rules [5] [6]. (These rules ensure business constraints, e.g. valid hub, unique reference, etc., are satisfied before finalizing the trip.) If any rule fails, the creation is aborted with an error.

- **Trip Update (Backend):** Updates to an FLM trip also undergo validation. For example, attempting to update the planned start time on a trip that has already started will throw an error ("Cannot update planned_start_time of a trip that has started" [7]). The new planned time must be a valid epoch timestamp; an invalid date triggers an `"Invalid Date for planned_start_time"` error [8]. The system also prevents setting a scheduled start in the past (this check is enforced to ensure scheduled trips are not back-dated – earlier code versions included this logic) [9]. Other updates are validated similarly: e.g. updating `vehicle_make_code` checks that the specified vehicle make exists in the org, otherwise "Vehicle Make not found" is thrown [10]. If changing a linked trip (coupling two trips), the system ensures the referenced trip exists; if found, it will **couple** the trips or decouple any previous pairing as needed [11] [12]. All update operations run through a `TRIP_UPDATE_VALIDATION` step (pre-transformation) to enforce business rules [13] [14].

- **Assigning a Driver (Worker) to Trip:** When assigning a rider/driver to an FLM trip, the system ensures the worker is eligible and available:

- The backend fetches **eligible workers** for the trip's hub and type, filtering by organization config flags. For example, if the org config `flm_trip_config.allow_available_for_trip_action` is true, only workers flagged as "available_for_trip_action" are considered [15]. If `allow_available_for_pickup_worker` is true, the worker must be marked active for pickup [16]. Configured valid vehicle statuses (`vehicle_valid_status_for_trip`) also filter out workers whose vehicles are not in an allowed status (e.g. only vehicles in service) [17]. Additionally, if `assign_helper_to_trip` is enabled, the system will exclude workers of type "helper" from driver assignment [18] (since helpers are assigned separately – see below).
- **Ineligibility Handling:** If no eligible worker matches, the assignment is blocked. The API returns an error "Worker is ineligible for trip assignment" [19]. The UI also prevents proceeding without a selection: if no rider is chosen and the user isn't adding a new one, it shows *"Please select a rider"* [20].
- **Task-Type Restrictions:** If configured (`allow_worker_task_type` flag), the system enforces that the driver's allowed task types cover all tasks in the trip. It loads the trip's tasks and ensures none have a type outside the worker's capability. If there's a mismatch, assignment fails (e.g. "Worker does not support task types of some tasks in this trip" [21]).

- **Preventing Conflicts:** The system checks if the worker (and their vehicle) is currently assigned to another trip. If so, logic decides whether to unassign them or block the action. For instance, if the worker's current trip is **already loaded ("is_freeze" true) and in assigned status**, and the config `restrict_vehicle_worker_change_after_trip_freeze` is true, the system refuses to reassign them: *"Cannot assign worker as worker is associated to trip X; Status: assigned & Loaded"* [22] . Likewise, if the trip is in a non-pending terminal status (e.g. completed), re-use is blocked [22] . If the current trip is just a default/placeholder trip and config allows, the system may automatically remove the worker from it and proceed (the `remove_default_trip_in_create_route` flag controls this) [23] .
- **Automatic Unassignment:** If the conditions allow reassigning a busy worker, the system will unassign them from their current trip behind the scenes. It calls an internal `__unassignWorkerFromTrip` to remove the worker (and their vehicle) from the old trip before assigning them to the new one [24] [25] . Similarly, if the FLM trip already had a different driver/medium assigned, that old assignment is removed (`extraDetails.is_worker_changed` is set true) before the new worker is attached [26] [27] .
- **Assignment Event & Status:** Upon successful assignment, the trip's record is updated: it sets `medium_id` (linking the worker's "medium"/device record), records the assignment time, and updates status to **"assigned"** (provided both a driver and a vehicle are now set) [28] [29] . An **AssignTripToWorker** event is logged with details (old/new driver IDs, etc.) [30] [31] . In the UI, a success message "Rider assigned" is shown on completion. If any error occurs, the UI displays the backend error message [32] [33] .

- **UI Validations:** The frontend modal for assigning a rider prevents submission if required fields are empty. For example, it will warn if no driver is selected ("please_select_a_rider" message) and will not call the API until that is fixed [20] . If the org requires a note on assignment (`is_notes_mandatory` flag), the UI enforces that a note is entered (otherwise "Notes is mandatory" error) [34] [35] . For reassigning a driver, if the option to disable the previous rider is selected, a reason may be required – the UI will prompt "Disable Rider Reason is required" if the configuration mandates a comment and none is provided [36] .

- **Assigning a Vehicle to Trip:** Vehicle assignment can happen alongside or after driver assignment (depending on config **worker-vehicle coupling**):

- The backend checks for **eligible vehicles** at the trip's hub. Like workers, it respects org rules: only vehicles at the hub and of the needed make/type are returned. If none qualify, it errors "Vehicle is ineligible for trip assignment" [37] [38] . The UI similarly forces a selection – submitting the Assign Vehicle form with none picked triggers a warning *"Please select a vehicle"* [39] .
- **Preventing Conflicts:** If the trip already has a vehicle or the chosen vehicle is in use, similar checks are applied. For example, if the trip currently has a vehicle and it's loaded (and `restrict_vehicle_worker_change_after_trip_freeze` is true), you cannot swap it unless marking a breakdown. The system would throw: *"Cannot assign vehicle as vehicle associated to trip X; Status: assigned"* (when trying to change a loaded vehicle without a breakdown flag) [40] . It also blocks assigning a vehicle that's part of a departed/started trip: *"The Vehicle is already part of a departed trip X"* [41] [42] . If the chosen vehicle is simply assigned to another active trip, the system will unassign it from that trip first (similar to driver logic). It invokes `__unassignVehicleFromTrip` on the vehicle's current trip (unless it's the same trip) before proceeding [43] [44] .
- **Vehicle Breakdown Scenario:** If a vehicle is being changed due to a breakdown, a special flag (`vehicleBreakdown`) is used. In this case, the system is more permissive in allowing a swap even after a trip has started. It will record the ending odometer reading of the broken vehicle (`endKMReading`) and mark that vehicle as disabled/unavailable. For instance, when

unassigning the old vehicle, if `vehicleBreakdown` is true, the old vehicle's record is updated with the odometer reading and possibly flagged as needing service [45] [46].

- **Updating Trip & Events:** On assigning a vehicle, the Trip is updated with the new `vehicle_id`, and `assigned_time` is set if not already. Extra metadata about the vehicle is stored: the `extra_details` JSON gains `vehicle_details` (vehicle code/number, fleet type, capacity, etc.) and `vehicle_make_details` [47] [48] [48]. If a vendor is associated, `vehicle_vendor_id` and vendor name/code are added too [49] [50]. The system also calculates utilization metrics (e.g., volume/weight usage) and, in a breakdown, accumulates the distance traveled by the first vehicle (adds to `previous_vehicle_distance`) [51] [52]. An **AssignTripToVehicle** event is logged, and if it was a breakdown, a **VehicleBreakdown** event is also logged with the old and new vehicle info and odometer readings [53] [54]. The trip status is set to `"assigned"` once both a driver (medium) and vehicle are attached (for coupled mode) [55] [56].

- **UI Considerations:** The Assign Vehicle modal on the frontend populates a list of free vehicles at the hub (or all vehicles if scheduling) and highlights if a vehicle is currently on a trip (showing the trip # and status) in its details. The user must pick one and optionally add a note. The UI prevents submission if none is selected ("please_select_vehicle" warning) [39]. For reassigning a vehicle, the UI uses the same endpoint as driver reassign (the `reassignTrip` API) with a `reassign_vehicle` flag, and allows marking the old vehicle as disabled (e.g., if it broke down) [57] [58].

- **Assigning a Helper (Optional):** If the operation is configured to allow helpers (assistant riders) on FLM trips (`assign_helper_to_trip`), the system supports assigning a helper to ride along. The helper must be free (not currently assigned as a helper elsewhere) and of the helper worker type. The `getEligibleHelpersForHub` query ensures the selected helper isn't already tied to a medium in use [59] [60]. If no eligible helper is found, "Helper is ineligible for trip assignment" is raised [61] [62]. When assigning, any previously assigned helper on this trip is unassigned (their `current_helper_trip_id` cleared) and similarly the new helper is removed from any prior trip [63] [64]. The trip's `extra_details.current_helper_id` is then set to the helper's ID. (UI for helper assignment would be analogous, but this is a less common flow.)

- **Trip Loading & Freezing Validations:** "Freezing" an FLM trip refers to finalizing the loading of all packages and sealing the trip for dispatch. The **freeze** action has several safeguards:

- It requires a valid trip ID, hub ID, and a list of loaded items ("loading_draft"). If the draft (list of consignments/tasks to load) is not an array, it throws "Invalid trip draft" [65]. It also ensures the hubId matches the trip's hub, etc.
- Before freezing, if e-waybill details are required, the system validates that all consignments have EWB data. For example, if `enable_consolidated_ewb_validation` is true, it checks the trip's consignments EWB status; if any are missing or failed, it aborts: *"EWB details missing for some of the CNs"* [66].
- The `allow_partial_loading` flag (sent as `allow_partial_freeze`) dictates whether a trip can be frozen if not all planned stops are loaded. By default partial is false unless explicitly allowed (the API sets `allowPartialFreezeTrip` based on input) [67]. If partial loading is not allowed and some packages are not scanned, the freeze action would error (the backend validation catches mismatched counts).
- **Excess Loading:** If the org allows loading extra consignments beyond the plan (`flm_trip_config.allow_excess_loading`), the freeze process permits additional tasks not originally in the trip plan. This flag is read at freeze time [68]. Without it, any consignment in the draft that wasn't part of the trip's plan would cause an error.

- When freezeTrip is called, the system wraps it in a database transaction and uses a **TMS Handler** to perform the operation. It finalizes loading all consignments: the Trip's internal state `extra_details.is_freeze` is set true (indicating the trip is sealed/loaded) [69]. It also records driver details if provided (e.g., manual entry of driver name/number) [70].

- After a successful freeze, the trip is effectively ready to depart. The API returns status "OK" on success [71]. (If any validation fails during freeze – e.g., no hub, trip not found, etc. – it returns a `wrongInputError` explaining the issue.)

- **Trip Start & Completion:** Changing the state of an FLM trip (e.g., "start trip", "end trip") also has safety checks:

- A trip cannot be started (departed) until it's been frozen/loaded. If somehow an attempt is made to start a trip that isn't loaded, validations on the backend would prevent it (the Trip's status/flags are checked – an unfrozen trip should not move to "in transit"). In practice, the UI only shows a "Depart" action after loading is completed.
- Once a trip is en route, the system tracks its progress. If SIM/GPS-based tracking is enabled for the org, the rider app will send location updates – the backend has hooks for sim-based tracking (config `isSimBasedTrackingEnabled`) and will only consider a trip "on-time" departure if driver consent was obtained when required [72]. (These tracking details are internal and ensure compliance with any opt-in requirements for tracking the driver's device.)
- **Unloading and Closure:** Upon reaching the destination hub, the trip must be unloaded and closed. The **start of unloading** is recorded via a similar call (`startUnloading`), which asserts tripId and hubId, and then marks the trip as unloading in progress. This triggers business logic: the Trip's state changes (e.g., status might change to "unloading"), and a **Vehicle** handler marks the vehicle as "start unloading" at that hub (for operational tracking) [73] [74]. There are typically UI scans for each package unloaded (each stop's consignment is scanned off the vehicle).
- **Finish Unloading:** When all packages are offloaded, the **finishUnloading** action finalizes the trip. It requires the trip draft of unloading (what was actually unloaded) and can optionally skip certain scans (e.g., `skipConsignmentInscanAtHub` if the workflow doesn't require inscanning at the hub) [75] [76]. The backend TripHandler performs validations here too – ensuring all expected consignments are accounted for unless skip-scan is true. Once finished, the trip is closed out: the Trip's status likely moves to **"completed"**, and `end_time` is recorded. The vehicle is marked as unloaded (freeing it for new trips) [77] [78]. Any pending reconciliation tasks (cash on delivery reconciliation, etc.) are flagged for completion.
- After completion, no further modifications to the trip are allowed (the system will treat it as a historical record). The UI reflects the trip as completed (and it may disappear from active trip lists to a completed trips view). Any attempt via API to alter a completed trip would be rejected (as the status would not be in allowed set for updates – e.g., backend checks that status is in "pending" states before allowing edits [22]).

2. End-to-End FLM Trip Flow

Below is a step-by-step outline of the FLM trip lifecycle, integrating frontend and backend logic, including config flags and inter-module dependencies:

- **Trip Planning/Creation:** An FLM trip can be created via the Shipsy dashboard or via integration. On the dashboard (CRM-frontend), users may create a trip by selecting a set of orders or consignments for first-mile pickup or last-mile delivery. When the user triggers **"Create Trip"**, the backend `TripHandler.__createTrip` is invoked within a transaction [1] [79]. Key fields (hub, type, etc.) are set and the trip starts in `created` status. The system also generates a unique trip reference number if not provided [80] [4]. The new Trip is saved to the database and a TripCreate event is emitted [81]. At this stage, the trip has no driver or vehicle assigned and no stops attached; it essentially reserves a “trip shell” that will be populated with tasks (stops).

- **Adding Stops (Tasks) to Trip:** After creation, consignments or tasks are added to the FLM trip (either immediately during creation for a planned trip, or later via a planning UI). Each “stop” is represented by a Task (pickup or delivery task). In the UI, this might be done by selecting orders and adding them to the trip, or by scanning waybills into a loading app. The backend ensures each consignment can only belong to one active trip. If the same consignment is attempted on multiple trips, validation will reject it. (There are also org settings like `allow_rescheduled_cn_addition_to_trip` that control if a rescheduled package can be added to a trip [82].) Once tasks are associated, the trip’s `planned_task_list` and `incomplete_tasks` fields are updated (these hold arrays of task IDs). The **number of stops** is essentially the number of tasks added. The system will later compute `no_of_stops` based on the trip’s stop sequence length [83]. At this point, the trip can be seen in a planning view with a list of all stops (addresses/pincodes can be derived from each task’s location, and a tentative route may be shown).

- **Assigning Driver and Vehicle:** Before execution, the trip must be assigned a driver (rider) and usually a vehicle. In the TMS dashboard UI, the user opens an **Assign Rider** modal for the trip. The frontend fetches a list of free workers for that hub via `GET_FREE_WORKERS` API, which returns all currently available riders with their status (e.g., available or already on trip) [84] [85]. The list indicates if a rider is free (*green “available” tag in the UI*) or shows an associated trip and tasks count if currently busy [86] [87]. The planner selects a rider. If the rider is already on another trip, the system (as described above) will handle unassigning them from the old trip if allowed. On clicking **“Assign”**, the frontend calls the `assignRider` API with the `tripId` and chosen `workerId` (and a note if provided) [88]. The backend then performs the assignment through `tripHandler.assignTripToWorkerAndVehicle` (or separate assignWorker if decoupled) which encapsulates the logic described: eligibility check, automatic unassign from other trip if needed, linking the worker to the trip, etc. After successful assignment, the trip’s status might remain `"created"` or become `"assigned"` depending on vehicle assignment. The UI will now show the driver’s name against the trip.

- Next, the user assigns a vehicle via the **Assign Vehicle** modal. A list of free vehicles is fetched by `GET_FREE_VEHICLES` API for that hub [89] [90]. The planner picks a vehicle and clicks **Assign**. The frontend posts to `assignVehicle` with the `tripId` and `vehicleId` [91]. The backend `assignTripToVehicle` logic checks that the vehicle is not already in use or handles unassignment if it is (respecting the freeze/departed restrictions). On success, the Trip now has both a driver and vehicle linked and is moved to **“assigned”** status [29]. (If the system is configured with **worker-vehicle separation** disabled, the above two steps may be combined: e.g., assigning a rider automatically attaches that rider’s dedicated vehicle, since `changeVehicle` would default to true [88] [92]. In that case, a separate vehicle assignment step is not needed – the UI can do both in one go.)

- **Trip Execution (First Mile Pickup or Last Mile Delivery):** Once assigned, the trip is ready to be executed by the rider. The trip will typically appear on the rider’s mobile app. For **first-mile trips**, the rider will go out to the origin locations (sellers) to pick up packages; for **last-mile trips**, the rider will deliver packages to end customers. The sequence of stops is determined either by an optimization algorithm or by manual planning. (If route optimization is enabled, the system might automatically order the stops and set `sequence_details` for the trip.) The route and stops can be viewed on the dashboard – e.g., a route string like "CityA–CityB–CityC" is generated by grouping stop locations [93] [94]. The planned route is often shown via the `route` field in UI, and `no_of_stops` is the count of stops in that sequence. Each stop corresponds to a Task (which contains details like address and consignee). The driver's app guides them through these stops. The system tracks completion of tasks: as each pickup/delivery is done, the task is marked complete and removed from `incomplete_tasks`. The trip's `last_main_event_time` is updated whenever a significant event occurs (pickup done, etc.) [95].

- **In-Transit Tracking:** While the trip is ongoing, Shipsy can track its movement. If SIM-based GPS tracking is configured, the driver's device provides live location. The platform can compute ETAs for remaining stops and track milestones (there are flags such as `trip.rider_consent` for allowing tracking, and `tripTypeEnum` to distinguish FLM trips for tracking logic). These details ensure that the trip's progress can be monitored on the dashboard (e.g., showing current driver location, completed stops, etc.). Internally, every time the trip's state changes (e.g., a stop completed), a TripEvent is generated (like *TripArrivedStop*, *TripDepartedStop*, etc., and finally *TripComplete*). These events record timestamps and can trigger notifications or SLA checks.

- **Trip Loading at Hub (First Mile flows):** For first-mile pickups, once the rider has collected all shipments, they return to the hub. At the hub, an operator will **"freeze"** the trip to close it. In the Shipsy Ops app (HubOps interface) or dashboard, the user selects **"Seal Trip"** or **"Freeze FLM Trip"**. This triggers the `freezeTrip` API (internal endpoint `/trips/flmTrip/freeze`), which runs the validations discussed (all packages picked, EWB compliance, etc.). The system marks the trip as frozen (`is_freeze=true` in extra_details) and seals it for further processing [69]. From this point, the first-mile trip's life ends – it has handed over shipments to the hub. (Often, a corresponding Middle Mile trip will take over for linehaul to another hub.)

- **Trip Start and Delivery (Last Mile flows):** For last-mile trips, after assignment, the next step is to **start the trip (depart the hub)**. In the HubOps app, once the rider has loaded their vehicle with all parcels, the supervisor taps "Depart" (often this is combined with sealing the bag). The system will perform an auto-freeze if not already done and then trigger `startTrip`. On start, it records the vehicle's starting odometer reading (if required by config) and sets the trip status to "started" [96] [97]. The `start_time` is saved on the Trip record. The driver then goes out to complete deliveries. As each delivery is made, the mobile app updates the task status, collects proof of delivery, etc.

- **Completing a Last-Mile Trip:** After finishing all deliveries (or pickups, in first-mile), the driver returns to the hub with any undelivered parcels or proofs of delivery. The hub team then **starts unloading** the trip. In the HubOps interface, they click "Start Unloading" when the vehicle arrives – this triggers backend `handleTripStartUnloading` to mark the trip unloading in progress and log a gate-in or docking event for the vehicle [98] [74]. The operator will scan each returned parcel (if any) or just confirm that unloading began. Finally, they click **"Finish Unloading / Complete Trip"**. This calls the `finishUnloading` API, which finalizes the trip: any remaining consignments are reconciled (possibly marked RTO or undelivered), and the trip's status is set to **"completed"** [75] [76]. The system records `end_time` for the trip and generates a TripCompletion/TripClose event. The vehicle is marked free (`markVehicleUnloaded`) so it can be assigned to new trips [77] [78]. If configured, the app might prompt to enter the vehicle's end odometer reading and capture a photo (the `use_km_reading_in_trip_actions` and `use_km_image_proof` flags in `FlmTripConfig` control this) – these would be stored as `end_km_reading` and an image URL in trip extra_details.

- **Post-Trip Reconciliation:** Once an FLM trip is completed, there may be financial reconciliation steps (e.g., cash collected on delivery or cost allocation). The system marks whether the trip's COD reconciliation is done (`is_reconciliation_complete`) and whether all delivery fees are settled. These are indicated in the trip data (for instance, the `is_reconciliation_complete` field and related flags) [99]. Trips can also have custom charges or penalties – the system provides hooks (like trip Penalties metadata) to add additional fields for trip audits. These happen after trip completion and are beyond the core execution flow.

Throughout these flows, several **internal toggles and configs** influence behavior: - *Org-level FLM settings:* e.g. `allow_partial_loading` (partial freeze), `allow_excess_loading`, `restrict_vehicle_worker_change_after_trip_freeze`, `allow_worker_task_type`, `remove_default_trip_in_create_route`, `enable_worker_vehicle_separation` (decouples assignment steps), `send_vehicle_requisition_request` (enables on-the-fly creation of ad-hoc driver/vehicle if needed) [100] [101], etc. These flags are checked at appropriate steps to alter validations and flow. For example, if `send_vehicle_requisition_request` and the user chooses **"Add New Rider"**, the system will create a new worker and vehicle for the trip via an integration handler [102] [103]. Similarly, `flm_trip_config.is_notes_mandatory` forces a note input on certain actions [34]. - *Inter-module Dependencies:* The FLM trip logic ties into multiple modules: - **Worker/Vehicle Management:** Uses the Medium/Vehicle models. When assigning/unassigning, it updates `Medium.current_trip_id` and `Vehicle.current_trip_id` to reflect resource usage [104] [105] and clears them on unassign. Also integrates with the **VehicleRequest** subsystem if applicable (for outsourced vehicles, linking trip to vehicle requisition orders) [106] [107]. - **Task/Consignment System:** FLM trips are essentially containers for tasks. The Trip–Task relationship is maintained via fields like `planned_task_list` and `incomplete_tasks` on the Trip (arrays of task IDs). These tasks refer to **Consignments** (shipments) and have linkages to locations. The route and stop calculations depend on tasks' node locations and an Allowed Manifest Master (to map pincodes to city names for route strings) [108] [109]. During unloading, the system can optionally skip creating individual Inbound Scan records if `skipConsignmentInscanAtHub=true` (a performance optimization toggle for certain workflows) [110] [76]. - **Events & Notifications:** The Trip logic emits standardized events (creation, assignment, start, end, etc.). These events can drive notifications (e.g., SMS to customer when driver starts). One example is the *rider_consent* flow – if a trip requires driver consent for tracking, the system can send an SMS and record whether the driver consented (this is stored in trip extra_details and shown as "rider_consent" status) [111] [112]. - **External Integrations:** For some clients, FLM trips integrate with external systems (e.g., posting trip start/stop to an ERP or updating an external vendor's system if it's a 3PL run). The code contains hooks like `VendorVehicleRequisitionHandler.assignVehicleAndWorker` for vendor-provided resources [113] [114], and fields like `third_party_vendor_id` in trip extra_details to track an outside vendor association [115].

By design, FLM trip processing is robust: each stage (assignment, loading, departure, delivery, closure) has explicit checks to prevent invalid transitions and ensure data integrity. The frontend workflow guides the user sequentially, and the backend enforces the business rules at each API call to maintain consistency.

3. FLM Trip Data Model and Fields

An FLM trip is represented in the database by the **Trip** model. Important fields and their usage include:

- **Primary Identifiers:** Each trip has a unique internal `id` (primary key) and an **Organisation Reference Number** (`organisation_reference_number`) which is the human-readable Trip# shown in the UI [116] [117]. Often this is the same as the `trip_reference_number` (which may be auto-generated) [4]. Both identify the trip; the org ref is used in multi-tenant scenarios and external references.

- **Type:** `type` – The category of trip, e.g. `"FLM"` for first/last mile (and possibly `"MM"` for mid-mile) [118] [119]. This field distinguishes FLM trips so that certain logic (like route optimization or tracking) applies only to them.

- **Status:** `status` – The current state of the trip, stored as a string. Typical statuses for FLM trips include **created**, **assigned**, **started** (in transit), **completed**, and possibly **cancelled**. The status is updated as the trip progresses [120] [121]. For example: after driver assignment it becomes “assigned” [29], after departure “started”, and after finish unloading “completed”. Business rules ensure illegal status changes are prevented (e.g., cannot revert a completed trip to started).

- **Hub and Route Info:**

- `hub_id` – The hub (station) responsible for the trip’s start. For a first-mile trip this is the pickup hub (destination hub for the freight), for a last-mile it’s the origin delivery hub. A trip is always associated with one hub [122].
- `route` – Not stored as a column but computed for display. It represents the trajectory of the trip, typically concatenating city codes or hub codes of the stops (e.g., "BLR-DEL-AGR"). The route is derived from the sequence of tasks’ destination locations [109] [94]. This gives a quick view of the geographic path.

- `no_of_stops` – The number of stops on the trip. This is calculated based on the trip’s stop sequence. In practice, the system computes this as the length of the trip’s `sequence_details` array [83]. Each stop in the sequence corresponds to one task (pickup or drop). For example, if a trip has 5 delivery tasks, `no_of_stops` will be 5. (If multiple tasks are at the exact same location, they might still be counted separately unless the system groups them in sequence_details.)

- **Stop Sequence:** `sequence_details` – A JSONB field that holds an ordered list of stops (tasks) and possibly their optimized sequence info [123]. After route planning or during trip execution, this array is populated. Each element might include task ID, location coordinates, and other metadata for that stop. The length of `sequence_details` is used to derive `no_of_stops`. Initially, this may be empty or in planned order, and after optimization it’s updated to the final route order. Example structure:

```
sequence_details: [
  { "task_id": "TASK123", "address": "Location A", ... },
  { "task_id": "TASK124", "address": "Location B", ... }
]
```

- **Tasks and Consignments:**

- Trips have two array fields linking tasks: `planned_task_list` and `incomplete_tasks`. These are typically stored as JSON arrays of task IDs. `planned_task_list` contains all tasks assigned to the trip initially, in the planned order. `incomplete_tasks` tracks tasks yet to be completed. As the trip progresses, completed tasks are removed from `incomplete_tasks`. For instance, at trip start, `incomplete_tasks` equals the full task list; if one delivery is done, that task ID is removed from the array. These fields allow quick checks of remaining stops and are used in queries (e.g., for route calculation and driver eligibility as noted above).

- Each **Task** (not part of the Trip table but related) holds details of the stop: `task_type` (pickup, drop, etc.), `node_id` (link to a Location), and relations to one or more consignments. Tasks are linked via a join table to **Consignments** (shipments). In the context of FLM, each task often corresponds to one consignment or one stop with multiple pieces. The location (node) attached to the task is used to determine routing and is how the system identifies unique stops. For example, to generate the trip route string, it joins `task.node_id -> location.pincode -> city` [108].
- The **Consignment** data (like reference numbers, volumes, weights) can roll up into trip metrics. For example, the trip's total volume/weight and piece count are aggregated from its consignments and shown in the UI (fields like `volume`, `weight`, `total_pieces_count` on trip are calculated summary fields) [124] [125].

- **Driver/Vehicle Assignment Fields:**

- `medium_id` – References the **Medium** record associated with the driver assignment. In Shipsy's data model, a "Medium" can be thought of as the driver's device or a rider entity which might encapsulate the combination of a person and potentially their default vehicle. When a worker is assigned to a trip, `medium_id` is set to that worker's medium record [28]. This ties the trip to the driver in the database. (The Medium record itself has `head_worker_id` linking back to the Worker.)
- `vehicle_id` – References the **Vehicle** assigned to the trip (if any) [126] [55]. This is null until a vehicle is assigned. Once set, it remains for the trip's duration.
- `vehicle_make_id` – The make/category of the vehicle used. This can be set at trip creation (e.g., if planning requires a certain truck size) or gets filled when a specific vehicle is assigned. It's stored both as a reference ID and also duplicated in `extra_details.vehicle_make_details` for quick UI access to code/name [48] [127].
- `assigned_time` – A timestamp recorded when the trip becomes fully assigned (driver or vehicle assigned, depending on workflow) [128] [129]. This helps measure how long the trip waited before departing.
- `scheduled_worker_name` / `scheduled_vehicle_registration_number`: If the trip was **pre-scheduled** with specific resources, those appear here. For example, if a dispatcher scheduled a particular rider or vehicle in advance (without actually assigning them yet), these fields hold those planned assignments [130]. They are used in cases where `isScheduled=true` flows are utilized (the UI uses separate Schedule Rider/Vehicle APIs which populate `scheduled_*` fields instead of actual assignment).

- **Timing Fields:**

- `created_at` / `updated_at` – Timestamps for when the trip record was created and last updated [131] [132].
- `start_time` – When the trip was started (departed the hub) [128]. Set by the system at trip departure (either automatically at freeze if configured, or when "Depart" is clicked) [133].
- `end_time` – When the trip completed (finished unloading) [134]. Set when `finishUnloading` is successfully executed. These allow calculation of trip duration.

- `last_main_event_time` – The time of the last major event on the trip [135] [136]. This is updated whenever a key action happens (assignment, start, each stop completion, etc.). It's useful for operational monitoring – e.g., to see when a trip was last active.

- **Extra Details (JSON):** The Trip model has an `extra_details` JSONB column for miscellaneous attributes and flags. Many FLM-specific toggles are stored here per trip. Notable entries include:

- `is_freeze` – **Boolean**: true if the trip has been frozen (sealed after loading). This is set to true at the moment of freeze (prior to departure in last-mile, or at completion in first-mile) [69]. The presence of `extra_details.is_freeze = true` is used to prevent further modifications; for instance, once true, the UI won't allow adding more consignments, and the backend restricts driver/vehicle changes unless explicitly allowed by config [137].
- `is_manual_trip` – Boolean: true if the trip was manually created (as opposed to auto-generated by system). The code sets this at creation if a flag is passed, and also records `creation_source` as `"MANUAL"` in such cases [138] [139].
- `creation_source` – A code indicating how the trip was created (manual, via API integration, etc.) [138]. E.g., `MANUAL` for user-created trips, or other codes for system or integration.
- `movement_type` – The movement category of consignments (e.g., forward, RTO (return) etc.). If set, it's usually a lowercase string like `"forward"` or `"rto"`, derived from consignments. This helps enforce rules like disallowing RTO packages in certain FLM trips (some configs block planning RTOs) [140].
- **Freight type flags:** `is_ftl` / `is_ltl` – Booleans to mark if the trip is full-truck-load or less-than-truck-load. These can be set based on utilization thresholds. For example, if the loaded volume/weight exceeds configured percentages, the system might tag `is_ftl=true`. Conversely, default trips can be tagged LTL. These flags are used to determine `manual_freight_type` output (FTL/LTL) for reporting [141] [142]. (They are set when creating the trip or during freeze based on `flm_freight_type_config` settings [143] [143].)
- `vehicle_vendor_id` – If the vehicle comes from a vendor/3PL, this links to the Vendor. The extra_details may also include `vehicle_vendor_details` (name, code) for quick reference [49] [50].
- `vehicle_details` – An object capturing details of the assigned vehicle at the time of assignment [48]. This typically includes:
  - `vehicle_number` (registration number),
  - `vehicle_code` (internal code if any),
  - `fleet_type` (e.g., OWNED, DEDICATED, MARKET etc.),
  - `parent_vehicle_id` (if this vehicle is a child in a larger asset, like a coach in a train or container on a truck, for hierarchical tracking),
  - capacity and tags (`max_distance_per_trip`, any special `constraint_tags` the vehicle has, etc.),
  - `vehicle_hub_id` (base hub of the vehicle). These are recorded when the vehicle is assigned for auditing and analytics.
- `vehicle_make_details` – Similar object for vehicle type: includes `vehicle_make_code` and `vehicle_make_name` (like "LCV" or "Bike") [48] [127].
- `current_helper_id` – If a helper is assigned, this stores the worker ID of the helper (set when helper assignment is done).
- `is_hand_delivery_trip` – Boolean flag indicating the trip is a **hand-delivery** trip. Hand delivery means a staff carries packages without assigning to a rider (for example, in a small facility a staff member might directly deliver). If true, the system treats the trip specially: it won't allow assigning a driver (and will throw an error if you try to assign one: *"Cannot change worker for hand delivery trip – [Trip#]"* [144]). This flag is set on creation if such a mode is chosen.

- Other flags like `is_direct_dispatch_to_seller` (if true, the trip's loading/unloading logic ignores some dock procedures for dispatch-to-seller scenarios) [145] [146], and fields like `planned_start_time` (if the trip was scheduled for a future start, stored in extra_details until it becomes start_time) [147] [148].
- Start/End readings: If odometer capture is enabled, `start_km_reading`, `end_km_reading` and their timestamps are stored in extra_details (and also top-level fields for quick access) [149] [150]. These come from the driver's input or photo proofs at departure/arrival. The config `remove_odometer` can disable requiring these (part of `FlmTripConfig`: `use_km_reading_in_trip_actions`, `use_km_image_proof` control whether these appear) [151].

- **Custom fields:** The trip can have a `custom_fields` JSON for any client-specific data (e.g., a custom reference code, or additional checklist info). The Assign Vehicle UI, for instance, allows inputting custom fields defined by `opsDashboardVehicleChangeModalFields` and sends them with the request [152] [153]. These get saved in `trip.custom_fields`.

- **Related Entities:**

- **Worker (Rider):** Not a direct field in Trip (aside from medium_id), but commonly needed. The UI shows `worker_name` for the trip by joining through the Medium/Worker relation. In list APIs, they often join `Worker` to fetch the driver's name, phone, etc., under columns like "Driver" [154]. The trip record might also temporarily hold a `scheduled_worker_name` if a worker was scheduled in advance [155].
- **Vehicle:** Similar to worker, `vehicle_registration_number` is often shown via a join using `vehicle_id` [156]. If a vehicle was scheduled but not yet assigned, `scheduled_vehicle_registration_number` holds that plate number [155].
- **Hub:** The trip's hub_id links to a Hub entity. The UI and reports show hub names: e.g., "Hub" column comes from joining Hub.name for the given hub_id [157]. If the trip is a secondary leg (connected to a primary linehaul), it might also show the primary hub or primary trip reference.
- **Metrics:** Fields like `estimated_distance_metres` store the planned route distance (often computed via mapping APIs after optimization) [158]. After the trip, an actual distance traveled could be derived from GPS or odometer (and used for cost calculations). Utilization metrics – `vehicle_volume_utilisation`, `vehicle_weight_utilisation` (% filled) – are calculated when freezing the trip by comparing loaded volume/weight vs vehicle capacity [159] [160]. These along with `volume_capacity`/`weight_capacity` and actual `volume`/`weight` of consignments are stored for analysis [125] [161].
- **Charges and Reconciliation:** The trip may have financial fields like `total_freight_cost`, `trip_charges`, etc., especially if used in cost calculation. The `is_billed` and `is_verified` flags indicate if the trip's charges have been billed to client and verified [162]. These are filled in after trip completion during financial reconciliation workflows. (Not directly part of execution, but part of data model for FLM trips.)

Database Schema Note: The underlying database table for Trip contains many of the above as columns (as seen in the metadata). For example, `organisation_id`, `hub_id`, `type`, `status`, `trip_reference_number`, `vehicle_id`, `medium_id`, timestamps, etc., are actual columns in the `trip` table [163] [164]. Array fields like `planned_task_list` and `incomplete_tasks` are likely JSONB or array columns. Large structured data like `extra_details` and `sequence_details` are JSONB columns in Postgres [123]. This hybrid schema (structured columns plus JSON for flex fields) provides a balance between performance for key queries and flexibility for additional attributes.

To illustrate, an FLM trip record might look like (simplified):

```
{
  "id": "TRIP456",
  "organisation_id": "ORG1",
  "organisation_reference_number": "TRIP456",
  "type": "FLM",
  "hub_id": "HUB_BLR",
  "status": "completed",
  "created_at": "2025-06-01T10:00:00Z",
  "start_time": "2025-06-01T10:30:00Z",
  "end_time": "2025-06-01T15:00:00Z",
  "medium_id": "MED123",              // Driver assigned
  "vehicle_id": "VEH456",            // Vehicle assigned
  "assigned_time": "2025-06-01T10:10:00Z",
  "last_main_event_time": "2025-06-01T14:50:00Z",
  "planned_task_list": ["TASK1","TASK2","TASK3","TASK4"],
  "incomplete_tasks": [],            // empty by completion
  "sequence_details": [ ... 4 stops ... ],
  "extra_details": {
    "is_freeze": true,
    "planned_start_time": "2025-06-01T10:30:00Z",
    "is_manual_trip": true,
    "creation_source": "MANUAL",
    "movement_type": "forward",
    "is_ftl": false,
    "is_ltl": true,
    "vehicle_vendor_id": null,
    "vehicle_details": {
       "vehicle_number": "KA01ABC123",
       "fleet_type": "OWNED",
       "vehicle_hub_id": "HUB_BLR"
    },
    "vehicle_make_details": {
       "vehicle_make_code": "LCV",
       "vehicle_make_name": "Light Commercial Vehicle"
    },
    "start_km_reading": 10000,
    "end_km_reading": 10050,
    "current_helper_id": null,
    "is_hand_delivery_trip": false,
    "no_of_stops": 4        // (could be stored or computed on the fly)
  },
  "volume": 3.5,   // m³ loaded
  "weight": 120.0, // kg loaded
  "total_pieces_count": 20,
  "vehicle_volume_utilisation": 70.0,  // in %
  "vehicle_weight_utilisation": 80.0,
  "ewb_status": "COMPLETED",
  "is_reconciliation_complete": true,
  ...
}
```

In summary, an FLM trip's data encompasses identification, status/timing, linkages to driver & vehicle, the list of stops (tasks) and their route, and a variety of operational metrics and flags. The **"no. of stops"** is essentially the count of task entries in the trip's sequence – each stop is identified by a task's location (address/pincode). The number is calculated by the system when needed (for display or reports) as the length of `sequence_details` [83], ensuring it reflects the actual route stops.

Any relational aspect (e.g., tasks, consignment details) is accessible via joins or the JSON fields but kept consistent by the trip's lifecycle logic. This comprehensive data model enables the platform to enforce all the validations and processes described, and to report on FLM trip performance (timeliness, capacity usage, success rate, etc.) after completion.

Sources:

- Code excerpts from `TripHandler` (creation, update) [1] [79] [8] [7], `TmsHandler` and HubOps APIs (freeze, start/unload, finish unload) [165] [71] [75] [76]
- Assignment logic from `TripHandler.__assignTripToWorker` and UI modals [166] [167] [22] [28] [20] [39]
- Configuration flag usage and validations [168] [169] [21] [40] [34] [144]
- Trip fields from model metadata and columns definitions [162] [170] [128] [164] [123], including calculation of `no_of_stops` [83] and route derivation [109].

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 19 21 22 23 24 25 26 27 28 29 30 31 37 38 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 61 62 63 64 79 80 81 95 104 105 106 107 122 126 127 135 137 138 139 143 147 148 149 150 166 167 trip-handler.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/domain-models/trip-manager/trip-handler.js

15 16 17 18 59 60 82 83 93 94 108 109 118 140 141 142 159 160 168 169 trip-manager-utils.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/domain-models/trip-manager/trip-manager-utils.js

20 32 33 36 84 85 86 87 88 92 AssignRiderModal.tsx
https://github.com/shipsy/crm-logistics-frontend/blob/198f256803125fd84d540eb4fda7bcea0905dcb6/src/components/pages/retail/trip/AssignRiderModal.tsx

34 35 100 101 102 103 113 114 115 144 trip-manager-api.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/models/retail-dashboard-parts/trip-manager-api.js

39 57 58 152 153 AssignVehicleModal.tsx
https://github.com/shipsy/crm-logistics-frontend/blob/198f256803125fd84d540eb4fda7bcea0905dcb6/src/components/pages/retail/trip/AssignVehicleModal.tsx

65 66 67 68 71 165 trip-loading-api.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/models/hub-ops-app-parts/trip-management/trip-loading-api.js

69 70 73 74 75 76 77 78 96 97 98 110 145 146 flm-trip.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/domain-models/tms-handler/tms-handler-class/tms-handler-parts/flm-trip.js

72 tms-handler-class.js
https://github.com/shipsy/projectx/blob/4d4553cb6f672fd88d492767c666a7b24c3f868b/common/domain-models/tms-handler/tms-handler-class/tms-handler-class.js

89 90 91 trips.ts
https://github.com/shipsy/alpha-crm-logistics-frontend/blob/23c427361e8d5797ee323a1717446cd9a3a4bf6e/src/api/trips.ts

99 119 121 123 129 131 132 133 136 163 164 metadata.js
https://github.com/shipsy/projectx/blob/7d35883e52641b64b5bcb94bb39edbf835414f11/common/domain-models/object-handler/metadata/trip/metadata.js

111 112 116 117 120 124 125 128 130 134 154 155 156 157 158 161 162 170 ops-trip-columns.js
https://github.com/shipsy/alpha-projectx/blob/3ca04b50b95c6d1e5fb921fbd0a83bb41fead5d9/common/models/crm-dashboard-parts/trips/ops-trip-columns.js

151 FlmTripConfig.kt
https://github.com/shipsy/riderapp-transport/blob/d13cdc556ce6243658672d69e04fccfc751e04b2/app/src/main/java/com/shipsy/riderapp/transport/models/FlmTripConfig.kt