ChatGPT

Carrier/Vendor Payout Business Logic in Shipsy

Overview

Vendor Payouts in Shipsy handle the calculation and disbursement of payments to delivery partners (carriers or vendors) for completed logistics services. The process automates computing payables based on predefined rate cards and recorded shipment data, allowing operations teams to review and adjust payouts before approval and invoicing. It spans multiple services: the **Finance Service** (payout calculation and records), **ProjectX** (shipment data, rate configurations, batch computation jobs), **CRM Backend/Frontend** (triggering calculations, UI for review/approval), and the **Vendor Portal** (visibility and invoice actions for vendors).

In essence, when a set of shipments or trips are completed, the system can calculate what a vendor should be paid for that period or service. This involves fetching the relevant trips/consignments, applying the vendor's rate card (e.g. per km rates, fixed charges, penalties, etc.), and generating a **Payout** record capturing the computed amount and breakdown. Users can then review the payout, apply adjustments or deductions, approve it, and finally bundle approved payouts into an **Invoice** for payment. Below, we detail each aspect of this logic – from the API inputs/outputs and underlying models to the triggers, validations, integrations, and front-end workflows governing vendor payout calculations.

Payout Calculation API (`payoutCalculate`) – Inputs & Outputs

The primary interface to calculate a payout is the Finance Service's `POST /objectPayout/<source>/payoutCalculate` endpoint. This accepts a **PayoutCalculateDto** payload defining **who, what, and when** to calculate a payout for. Key input fields include:

- **payout_type** – The payout cycle or granularity (e.g. `"triplevel"`, `"monthly"`, etc.), corresponding to enum values like `triplevel`, `daily`, `weekly`, `monthly`, etc. [1]. This influences how the rate card is applied (per trip vs. aggregated period).
- **organisation_id** – Identifier of the tenant/organisation context (required for multi-tenant data segregation).
- **courier_partner** – The vendor or carrier identifier for whom payout is being calculated (typically a vendor code or name).
- **courier_account** – The account or contract identifier under that partner (used if a vendor has multiple accounts or contract types).
- **object_type** – Type of entity that will receive the payout, e.g. `"vendor"` for third-party carriers or `"worker"` for internal riders [2].
- **object_id** – The unique ID of that vendor/worker (the payout "object"). For vendor payouts, this is the Vendor ID in the system.
- **entity_type** – Type of logistics entity the payout is based on: `"trip"`, `"consignment"` (shipment/order), or `"vehicle"` [3].
- **entity_id** – The specific entity ID (e.g. Trip ID, Consignment ID, or Vehicle ID) for which the payout is being computed.

- **start_date** and **end_date** – The date range of services covered by this payout (for period-based payouts). For example, a weekly or monthly payout would use these to denote the cycle's start and end. For single-trip or order-level payouts, these may coincide with the trip date.
- **event_date** – The date of the payout event [4], often the same as end_date for a period closing or the trip completion date for trip-level payouts. This is stored in the payout record and used for filtering.
- **event_time** – Timestamp of the payout event [5]. This, along with event_date, can be used by the pricing engine to apply time-specific rates (if any).
- **Optional dimensional fields** – A variety of additional parameters that may affect pricing:
- *Geography:* origin/destination hub codes, first/last task city, planning hub info [6], etc.
- *Vehicle details:* vehicle_number, make/code, fleet type [7] [8], reporting hub, etc.
- *Route details:* route_code, trip_leg, distances (planned vs actual) [9] [10].
- *Operational metrics:* total_trips in period, vehicle_present_days (for monthly contracts), trip_duration_in_hrs, etc. [11] [12].
- *Charges and penalties:* lists of additional charge IDs or penalty IDs (`additional_charges_detail`, `trip_penalties_details`) that should be factored in [13] [14], and any manual vendor_charge overrides.
- *Flags:* is_replacement (if a vehicle was replaced mid-trip) with its details [15].
  These optional fields are used as needed depending on the payout_type and the vendor's payout configuration. For example, a *vehicle-level monthly payout* might utilize `planned_distance` or `vehicle_present_days`, whereas a *trip-level payout* would use actual trip distance, route, etc. All inputs are validated for type and presence as required by the DTO (e.g. all the fields listed above are marked `@IsDefined()` except those explicitly optional).

Output: The `payoutCalculate` API does not directly return the calculated amount or breakdown in the response. Instead, on success it creates a new **Payout** record in the database and returns a simple status (e.g. `{ success: true, status: 'OK' }` on success [16], or an error status if something went wrong). If the pricing calculation fails or returns no data, the API returns an error object with `success: false` and an error status/message (for example, `status: 'INVALID_PAYOUT_DATA'` if inputs are not sufficient [17], or it may propagate a `statusCode: 500` error from the pricing engine [18]).

Once created, the **Payout record** contains all the details of the calculation. Key fields in the Payout model include:

- **id** – Unique identifier of the payout record (UUID).
- **object_type** and **object_id** – The payout recipient type (`vendor` or `worker`) and ID, as provided in input [19] [20].
- **entity_type** and **entity_id** – The entity on which this payout is based (e.g. a trip or consignment) [21].
- **entity_reference_number** – A human-readable reference for the entity, fetched from the core system for convenience. For example, if `entity_type` is *trip*, this is the Trip Reference Number [22] [23]. (The Finance service calls an internal API to fetch this using the entity_id).
- **payout_type** – The payout cycle type (as in input, stored as an enum) [24].
- **start_date, end_date, event_date** – The date range and event date, stored as Date fields [25].
- **status** – The current status of the payout (an enum). New payouts are inserted as `pending` [26] [27], meaning they await review. Other statuses include `approved`, `rejected`, and `invoice_generated` (when tied to an invoice) [28].
- **calculated_payout** – The original computed amount (numeric). On creation this equals the full calculated amount [29].

- **final_payout** – The amount that will actually be paid out. Initially this is set equal to the calculated_payout [29]. If any manual adjustments are made, final_payout may differ (while calculated_payout preserves the original for reference).
- **payout (breakdown)** – A JSON array detailing the breakdown of charges that sum up to the payout. Each element has a `chargeId` and `payout` (amount) [30]. For example, for a trip-level payout, there might be charges like base_fare, fuel_surcharge, additional_trip_penalties, etc., each with an amount. This breakdown is obtained from the pricing engine's response (excluding the final total). Any charge with a negative impact (e.g. a penalty) is represented as a negative value. The system automatically negates certain charges like "additional_trip_penalties" to ensure they subtract from the payout [31].
- **computation** – A JSON blob of the input computation parameters used. Essentially, the entire request payload is stored here for audit/debug purposes [32]. This helps trace how the payout was calculated (including all optional fields provided).
- **payout_hub_id** – If applicable, the hub/location associated with this payout. For example, if the vendor is region-specific or if the payout was filtered by a planning hub, that hub's ID is stored [23] [32].
- **invoice_id** – If this payout gets linked to an invoice later, this will hold the Invoice's ID. Initially it is null.
- **created_at/updated_at** – Timestamps for record creation and last update.

The payout record effectively represents the output of the calculation. To retrieve detailed outputs (like the per-charge breakdown, reference numbers, etc.), one would use the **fetchPayoutDetail** API, which returns the payout record data including the breakdown and also enriches it with related info (e.g. vendor name/code and any invoice number if invoiced) [33] [34].

Example: A trip-level payout calculation might be requested with `payout_type = "triplevel"`, `object_type = "vendor"`, `entity_type = "trip"`, and the specific trip ID and vendor ID. The pricing engine might return a final price of, say, ₹5000 consisting of base fare ₹4500 and fuel surcharge ₹500. The finance service will create a Payout record with final_payout = 5000, calculated_payout = 5000, status = pending, and a payout breakdown array like `[ {chargeId: "base_fare", payout: 4500}, {chargeId: "fuel_surcharge", payout: 500} ]`. The response to the API call would be `{ success: true, status: 'OK' }`, and the UI can then query the payout ID via `fetchPayoutDetail` to display these details to the user.

Related Models and Configuration (Rate Cards, Shipment Data, Adjustments)

The payout calculation logic relies on several supporting models and configurations across the system:

- **Vendor Rate Cards & Payout Configuration:** The rules for calculating payouts are defined in **rate cards/configurations** associated with the vendor (carrier). These define how charges are computed – e.g. rate per kilometer, fixed charges per trip, penalties for delays, minimum guarantees per period, etc. In Shipsy, these configurations are part of the pricing metadata (often referred to as "courier pricing" or vendor payout dimensions). The Finance service itself does **not** hard-code rate logic; instead, it delegates pricing to a dedicated pricing module (internally called **OBB V2**). When `payoutCalculate` is called, it bundles the input data into a `consignmentArray` and calls `UtilService.calculatePrice(..., metadataType: 'courierPricing')` [35]. This makes an internal API call to the pricing service (via `obbv2NetworkService.post` to a `/price/get` endpoint [36] [37]). The pricing service uses the vendor's configured rate card and the data provided (distance, route, etc.) to compute the

payout charges. In other words, **the payouts are calculated based on the vendor rate cards in the client's system** [38] . If the vendor's payout configuration is not set up, the calculation will fail – indeed, the ProjectX logic explicitly checks that `vendorPayoutCalculationDimensions` are configured, otherwise it throws an error [39] .

Examples of payout rules: A vendor's rate card might specify a base rate for a trip (e.g. ₹X per km for a certain vehicle type), extra waiting charge per hour, a minimum monthly payout guarantee (e.g. ₹Y if total falls short), etc. The system supports these via the metadata. During calculation, it will compute each applicable charge. For instance, if a **minimum guarantee** applies on a monthly payout, the pricing logic would compare the computed total vs. the guaranteed amount and ensure the final payout is at least the minimum (potentially reflected as a special charge in the breakdown, like "MinimumGuaranteeAdjustment"). If there are **deductions or penalties**, the pricing engine will include them as charges with negative values. The code above shows that if a chargeId "additional_trip_penalties" is present, its amount is multiplied by -1 to record it as a deduction [31] . (This ensures, for example, a ₹100 penalty is stored as -100 in the payout breakdown.)

- **Shipment/Trip Data:** The actual data of completed shipments or trips is crucial for calculating payouts. ProjectX (the core TMS) stores all trip details (distance traveled, route taken, vehicle used, timestamps, etc.). The payout calculation will fetch and use this data either directly or via aggregated inputs:
- In the **Finance Service's single payout API** (used for individual calculations), it expects the caller to supply necessary metrics (distance, etc.) in the PayoutCalculateDto if needed. For example, fields like `distance_in_km`, `trip_duration_in_hrs` or `vehicle_number` can be provided. If the caller only knows an entity ID (say a Trip ID) and not the metrics, the Finance service can fetch some info: it uses a helper to retrieve basic entity data by ID (via ProjectX). For instance, it maps `entity_type` to a data type (e.g. "trip" or "consignment") and calls `FinanceServiceIntegration.fetchMultipleDataFromReferenceKey` to get properties like the reference number and hub for that trip [40] . This returns a map of data which is used to populate `entity_reference_number` and `payout_hub_id` in the payout record [22] [41] . However, detailed metrics (like distance) would typically need to be passed in or computed by the pricing engine. In practice, the **OBB pricing call** might itself fetch required shipment data if given an entity reference – but here it's passed a fully formed consignment object, so presumably the data in the DTO (like `distance_in_km`) should be accurate.

- In the **ProjectX batch flow** (described later), the system automatically gathers all completed trips for a vendor in the given date range from the database [42] . It filters by the specified status (typically "completed_at" for completed trips) and date range, and then aggregates or groups them as needed for the rate calculations [43] . For example, if a vendor's contract has different payout components (say separate calculations for different vehicle types or lanes), the logic iterates through configured dimensions (each dimension might correspond to a sheet in the output, like "FTL Trips", "LTL Trips", "Penalties") [43] [44] . It computes the payout for each subset and then totals them. The end result is similar (a total payout amount and breakdown by charge), but here ProjectX produces a detailed Excel of all trips and charges.

- **Adjustments & Additional Charges:** During calculation, the system can include extra charge lines. For instance, `additional_charges_detail` in the input can carry a list of charge item IDs that should be applied (perhaps pre-recorded extra services). The pricing engine will include those in the breakdown. Penalties (like late delivery fines, etc.) are included similarly via `trip_penalties_details`. The Finance Service doesn't interpret these itself but passes them through to pricing. Once the payout is recorded, **manual adjustments** can be made prior to approval (see next section). These manual adjustments are effectively *post-calculation*

modifications to the payout breakdown or final amount, usually to account for edge cases or corrections not captured by automation.

- **Payout Config Master Data:** There are also supporting models for payout master data, such as **VehicleVendor (the vendor master)** which stores vendor-specific settings. For example, a vendor record may contain tax information (GST tax rate, tax type, GSTIN, etc.) in `extra_details`. We see that when generating an invoice, the system fetches the vendor's tax rate and type [45] to compute taxes, and checks a flag for reverse charge applicability [46]. Vendor master also likely holds configuration like `vendor_payout_charges_distribution_logic` (how to distribute charges across trips) [47], or concurrency limits for calculation [48], and possibly a toggle for allowing automatic invoice PDF generation [49]. These settings influence the payout process (e.g., whether to auto-generate a PDF of the invoice after calculation, whether to split trip-level charges among consignments, etc.).

In summary, the **payout calculation is highly data-driven**: it pulls in trip data, applies vendor-specific rate rules, and prepares a structured payout record ready for review.

Triggers for Payout Calculation

Payout calculations can be initiated in a few ways, ensuring flexibility in different operational setups:

- **On-Demand via Operations UI:** The most common trigger is a user (finance or operations staff) initiating a payout run for a vendor through the Shipsy CRM (operations portal). In the CRM frontend, a **Vendor Payout module** allows selecting one or multiple vendors and a date range, then invoking calculation. This corresponds to an API in ProjectX (`POST /dashboard/vendor-payout/v1/calculate`) which handles the request [50]. The user provides parameters like `vendor_id_list` (one or more vendor IDs), a date range (`from_date_string`, `to_date_string`), and a trip status filter. Typically, the status filter is set to "completed_at" (meaning consider completed trips). The backend will iterate each vendor and queue up a payout calculation job. For example, in code, for each vendor it instantiates a `VendorPayoutHandler` and calls `createCalculateVendorPayoutRequest` with the vendor details and date range [51]. It logs a request (in a `VendorPayoutLog`) and then triggers an **async job** to perform the heavy calculation in the background [52] [53]. This on-demand trigger is often used at the end of a payout cycle (e.g. end of week or month) or after a set of trips is done, to compute what is owed to the vendor.

- **Scheduled/Bulk Jobs:** The system can be configured to calculate payouts on a schedule. While we did not see a fixed cron in the code, the multi-vendor support in the Dashboard API suggests that an admin could trigger all due vendors at once (by passing multiple IDs). Organizations might script this or use an automation to call the API periodically. For example, on the 1st of each month, call the API for all vendors for the previous month's date range. The ProjectX async job queue architecture would handle these in the background. Additionally, one could conceive a nightly job for daily payouts or a weekly job, though exact schedulers would be configured outside the code (or via external orchestrations calling the API).

- **Automated on Shipment Completion:** In the current design, payouts are not calculated **immediately** upon each trip completion (to avoid heavy sync processing for every trip). Instead, completed trips are marked and picked up in batch as described. However, the system does record necessary data at completion that feeds the payout. For instance, when a trip is

completed, all its chargeable components are finalized (distance traveled, any tolls, etc.). There is logic to update each trip and consignment with the "transporter charges" once a payout is calculated. In the ProjectX flow, after computing the payout for all trips, it calls `asyncUpdateChargesOnTripAndConsignments` for each trip to tag the trip records with the payout charges [54] [55]. This ensures that a trip's financials are updated (preventing it from being included in another payout or duplicating charges). In essence, the *event* of trip completion makes that trip eligible for payout, and the next payout run will scoop it up.

- **Vendor Self-Service:** Typically, vendors do not trigger the calculation themselves (they instead *raise invoices* or view payouts after calculation). The vendor portal is more about reviewing what was calculated and then accepting or disputing it. So the primary triggers are on the ops side.

In all cases, there are checks to ensure the inputs for calculation are valid. For example, the system will not start a calculation if required parameters are missing. The ProjectX handler will assert that organisationId, vendorCode, tripStatus, and date range are provided [56] (throwing an error if not). It also restricts the tripStatus to allowed values (`created_at`, `assigned_at`, `started_at`, `completed_at`) [57] – in practice, "completed_at" is used for payout (some clients might hypothetically calculate interim payouts at different stages, but generally payout is on completed shipments). If the date range is missing or invalid, it errors out as well [58].

Once triggered, a payout calculation job moves through the following **phases**: 1. **Data Gathering:** Fetch all relevant trips/shipments for the vendor and time window. (Batch mode: query DB for trips; Single calc mode: use provided entity ID or fetch via API if needed.) 2. **Computation via Pricing Engine:** Prepare input payload(s) and call the pricing service which applies the rate card. This yields computed charges and totals. 3. **Record Creation:** Create Payout record(s) in the finance database. (In single mode, one Payout is created per request. In batch mode, the logic might create one Payout per trip or one aggregated per vendor per period – currently, the batch uses an Excel/PDF output rather than multiple DB records. With the finance service introduction, a design could be to create a payout entry per trip even in batch, but the present ProjectX code doesn't explicitly do so; it operates more on generating files and an invoice PDF.) 4. **Post-Processing:** Mark source data to prevent re-use (e.g. flag those trips as "payout calculated"). Possibly send notifications or update logs.

The result is that the payouts are ready for review.

Payout Calculation Flow & Logic

Within the Finance Service (`handlePayoutCalculation`): When the `payoutCalculate` API is called, the Finance microservice executes the `ObjectPayoutService.handlePayoutCalculation()` method [59] [60]. The high-level steps are:

1. **Validate basic inputs:** (The code has comments to "validate valid vendor/courier", though the implementation currently trusts the provided IDs. The DTO validation already ensures required fields are present. Any obviously invalid combo – e.g. unknown payout_type – would result in no pricing data and thus an error returned later.)

2. **Prepare pricing request:** The service constructs a parameter object for price calculation. It takes the input DTO and augments it with `payoutType` and a numeric code `payoutTypeNumber` derived from the payout_type (using a mapping, e.g. `monthly -> 16`) [61]. It sets `metadataType: 'courierPricing'` and `dateType: 'event_time'` for the pricing API, indicating that it should use the courier/vendor pricing rules and consider the `event_time` in

calculations. All input fields (like distance, etc.) are included in the `consignmentArray[0]`. Then it calls `util.calculatePrice(organisationId, params)` which internally calls the OBBv2 pricing service endpoint [62]. This returns a result object with potentially multiple calculations; here we expect one result in `data[0]` since we passed one consignment.

3. **Handle pricing response:** The returned data (let's call it `payoutData`) contains charge keys and a `finalPrice`. For example, `payoutData` might look like `{ baseFare: 4500, fuel_surcharge: 500, finalPrice: 5000 }`. The code first checks if `statusCode === 500` in the result, which indicates an internal error during pricing; in that case it simply returns that as the API response to inform the caller [18]. If no data is returned (null), it returns an `INVALID_PAYOUT_DATA` error [17]. Assuming we have a valid payoutData, it proceeds.

4. **Fetch reference info:** Before saving, the service enriches the data with some reference fields. It prepares a lookup to fetch details of the `entity_id` from ProjectX – specifically it wants the reference number and possibly the planning hub. It calls `util.fetchMultipleDataFromKey` with the entity_id and type (for example, for a Trip entity, data_type = "trip") [63]. This integration call goes to ProjectX's FinanceServiceIntegration, which in turn uses the appropriate model (Trip, Consignment, etc.) to fetch data by ID [64] [65]. The returned `dataInfoMap` might contain fields like `reference_number` and `hub_id` for that entity. These are then extracted: `entityReferenceNumber` becomes part of the payout record (for easy identification of the shipment later), and `planningHubId` (if any) is stored as `payout_hub_id` [23] [41].

5. **Begin DB transaction:** A database transaction is opened via a QueryExecutor [66]. This ensures that the payout insert and any related operations occur atomically.

6. **Build payout breakdown:** The code then iterates over each key in `payoutData` to construct the `payout` breakdown list [30]. It skips the special key `finalPrice`, and for each other key (charge), it takes the numeric value and pushes `{ chargeId: <key>, payout: <value> }` into an array. In our example, it would create two entries for baseFare and fuel_surcharge. (If any values were non-numeric, it would ignore them, but typically pricing returns numbers for defined charges.)

7. **Insert Payout record:** It then creates a new Payout entity instance and populates its fields [67]:

8. organisation_id, object_id, object_type, entity_id, entity_type from the input.
9. final_payout and calculated_payout set to the computed total (`payoutData.finalPrice`) [29].
10. status set to `PENDING` (as default or explicitly) [26] [27].
11. payout_type set to the enum value corresponding to input (converted via TypeScript enum).
12. event_date, start_date, end_date set from input (as Date objects).
13. computation field set to the input DTO object (the code spreads `body` into it).
14. payout (breakdown) field set to the array constructed above.
15. entity_reference_number and payout_hub_id set from the fetched entity data (or empty if none) [32].

This fully describes the payout. The service then calls `queryExecutor.insertSingle(payout, Payout)` to save it to the DB [68].

1. **Commit and return:** The transaction is committed and a success response is returned to the caller [16]. The new payout is now stored and can be retrieved via other endpoints for display.

Throughout this flow, certain **business rules/validations** are enforced: - The system currently allows duplicate calculations (it doesn't check if a payout for the same vendor and period already exists at this stage). That check is instead enforced at invoice creation time to prevent double-paying (see Invoice section where it checks if an invoice exists for that date range). - Date range validity is important; if the input dates are missing or invalid, the code returns an error immediately. It also imposes a limit: the Finance Service refuses to fetch payout list data for ranges over 45 days [69] (to keep queries efficient). This 45-day rule is likely also a guideline for calculation cycles (i.e. payouts are expected to be calculated for periods up to quarterly at most). - The pricing engine handles many business rules (distance rounding, minimum amounts, etc.) as per configuration, so the Finance Service code doesn't duplicate those checks.

Batch Calculation in ProjectX: The ProjectX `VendorPayoutHandler.calculateVendorPayout()` performs a similar flow but at scale [48] [70]. When triggered via the Dashboard API, it: - Retrieves all **available trips** for the vendor in the date range [42]. - Gets the vendor's payout charge metadata (essentially the rate card and the definition of different payout "dimensions") [39]. - For each dimension, it filters the trips (e.g. separate trips by vehicle type or route if the contract says so) [43], aggregates them if needed (e.g. sum distances for monthly calculations) [71], and calls a pricing function to get `tripDataWithPricing` and `totalPayout` for that set [71]. This is analogous to calling the pricing service, but it might be using an internal pricing library in ProjectX for the older system. - It then generates Excel data rows for each trip with detailed breakdown [72]. Each dimension yields an "excelData" array and a total. - Combines all results (multiple sheets of Excel) and sums a grand total [73]. - Generates a summary sheet and uploads the Excel to S3 (getting a URL) [74] [75]. - Optionally, generates a **Vendor Payout Invoice PDF** if configured, by preparing invoice data (including tax calculation) and calling a PDF service [49] [76]. The tax is computed using the vendor's tax_rate; e.g. if total payout is ₹5000 and tax_rate is 18%, it computes GST ₹900 and shows it on the invoice [77] [78]. The PDF and Excel links are then available for download. - Updates each trip and consignment with the charges applied (so that those records carry the information that payout for them is done) [54]. - The result returned by this method includes the `totalPayout` sum, the S3 URL for the Excel, and the PDF URL (if any) [79] [49].

This batch flow is more about producing outputs for review (Excel/PDF) rather than inserting records (note: it doesn't create individual Payout entries for each trip in the finance DB in the current implementation). It's possible that as the new Finance Service is adopted, this batch process could be adjusted to record each component payout, but currently the integration point is at invoice level (the ProjectX flow directly can generate an invoice PDF and likely the ops team uses that for processing outside or in parallel to the structured invoice in Finance Service).

Charge Calculations & Business Rules: Across both flows, key business logic includes: - **Rate lookup:** For each charge, the system looks up the rate from the configured rate card. For example, to calculate distance-based charge, it multiplies distance by per-km rate. To apply a fixed daily allowance, it might count days. All such logic is encapsulated in the pricing configuration. The release notes emphasize that the system enables *reliable trip rate calculation based on vendor rate cards* [38]. - **Conditional charges:** If a vendor has a **minimum guarantee**, the pricing might include a charge like "Minimum Guarantee Adjustment" if actual earnings < minimum. Conversely, if there's a cap, it might adjust down. The finance code doesn't explicitly handle this – it comes through from pricing. - **Penalties and Deductions:** These appear as negative charges. The code snippet we saw ensures any "additional_trip_penalties" charge is stored as negative [31], meaning if the pricing returned +100 for a penalty (just as a value), it flips it to -100 to subtract from final. - **No double-counting:** Once a payout is calculated for a trip, ProjectX updates that trip's record (via `asyncUpdateChargesOnTripAndConsignments`) with the charges and flags it. This likely prevents including that trip in another payout run (the query to fetch availableTrips likely excludes trips already having transporterCharges set, though that detail is

abstracted). This ensures one trip is paid only once. - **Grouping vs. Single:** A single payoutCalculate call usually corresponds to one trip or one vehicle's period. But for periodic vendor payouts, the practice could be either to use batch mode or call payoutCalculate for each grouping. The `payout_type` field allows the finance service to handle **weekly, monthly, daily** as values. In such cases, the expectation is that the input covers the whole period (e.g. set start_date and end_date to a month range, and perhaps entity_type = "vehicle" or "vendor" with aggregated metrics). The system would then calculate the payout for that entire period as one entry. For example, a monthly vehicle rental payout could be computed by passing total_km driven and days present for that vehicle in that month, using payout_type "monthly" – resulting in one Payout record for the month.

In summary, once a trigger initiates the calculation, the system systematically computes the payout using configured rules and records it, handling errors (like missing config or data) gracefully by returning error statuses. This automated computation replaces manual effort and ensures consistency. As noted in the product release: *clients can simply input the vendor code, date range, and status to get highly detailed outputs* including trip numbers, distances, final cost, and the breakdown of how the cost was calculated [80] – which speaks to the transparency of this automated payout process.

Validation Rules and Eligibility Checks

Throughout the carrier/vendor payout flow, various **validation rules** ensure that payouts are only generated for eligible scenarios and that data integrity is maintained:

- **Valid Vendor/Carrier:** The payout calculation should be for a legitimate, active vendor. The Finance Service expects a `courier_partner` (vendor code) and `object_id` (vendor ID). While the service does not itself verify the ID against a database table (it assumes the IDs come from the UI which provides only valid choices), the ProjectX layer does check this. In the Dashboard API, for each vendor ID provided, it queries the `VehicleVendor` model; if a given ID is not found, it errors out: *"Invalid vendor id: X"* [81]. This ensures you can't calculate a payout for a nonexistent vendor. Similarly, `vendorCode` must be present [82] – if the code (which usually is the unique code for the vendor) is missing, it throws an input error.

- **Trip/Shipment Eligibility:** Only certain shipment statuses qualify for payout. The system typically requires trips to be **Completed**. If an attempt is made to calculate payout on an inappropriate status, the system will reject it. For example, if `tripStatus` parameter is not one of the allowed values or is missing, the ProjectX handler throws *"Invalid Trip Status"* [57]. So a user must specify e.g. `completed_at` (which indicates we are calculating based on each trip's completion timestamp falling in the date range). This effectively filters out trips that aren't finished. Additionally, if no trips fall in the range, the handler will eventually throw *"No data found for excel"* [83], preventing a blank payout.

- **Date Range Requirements:** Both Finance and ProjectX enforce that a date range is provided. If `from_date` or `to_date` is missing, it errors: *"Date range should be present"* [58]. The Finance Service's fetch/list APIs explicitly require start_date and end_date [84], returning an error if not provided. They also guard against overly large ranges: *"Maximum date range allowed is 45 days"* is returned if you exceed that [69]. This not only prevents unwieldy computations but also implicitly ensures cycles are of reasonable length.

- **Duplicate Prevention:** At the payout calculation stage, the system does not check for duplicate calculations in code (one could theoretically create two payouts for the same vendor & period if calling the API twice). However, at **invoice creation**, a validation exists to prevent invoicing the

same period twice. When creating an invoice, it checks if any invoice (status not cancelled) already exists for that vendor and date range, and if so returns *"Invoice already exists for the given date range"* [85] . This implies that the intention is one invoice per vendor per cycle. If a duplicate payout existed, presumably only one invoice would be made, but the presence of duplicate payouts could confuse things. In practice, users likely avoid triggering duplicates; the system logs and UI help ensure each period is processed once.

- **Status Transitions:** Once a payout record is created, it has to follow allowed status transitions. **Manual updates or approvals are only allowed in the correct status:**

- The code checks that a payout is `pending` before allowing updates. If you call update on a payout that's already approved (or invoiced), it returns an error *"Payout update allowed only for pending status"* [86] . Similarly, trying to approve a payout not in pending will error *"Payout is not in pending status"* [87] . These validations protect against altering a payout after it's been finalized or paid.
- Although a `rejected` status exists, there isn't a dedicated API in Finance Service to reject a payout. Possibly, rejecting is done by simply not approving and leaving a comment, or marking it in an external way. If needed, an update could mark status = rejected, but there's no explicit method exposed. The typical flow is either approve or cancel via invoice.

- Once payouts move to `invoice_generated` (when tied to an invoice), they can't be individually changed; any changes would require canceling the invoice first.

- **Monetary Validations:** The system ensures that numeric fields are handled correctly. For instance, when updating payouts, it converts inputs to Number and sums them up [88] . If the final total turns out to be NaN or nothing changed, it errors accordingly (*"No data to update"* if you didn't provide any actual modifications) [89] . This prevents accidentally writing a null or undefined payout value.

- **Tax and Compliance:** At invoice time, basic validation ensures tax info is present. The Finance Service fetches the vendor's tax rate; if not found, it defaults to 0 (no tax) [45] . It also assigns a default tax type 'GST' if none provided. This ensures an invoice always has some tax detail (even if zero). The invoice number is auto-generated in a sanitized format ("INV<random><timestamp>") [90] , guaranteeing uniqueness and a standard format.

- **Integration Auth:** Between services, calls are secured via API keys and org IDs. The FinanceServiceIntegration in ProjectX requires a valid internal API key in requests from Finance Service [91] [92] . This ensures only authorized services can fetch data like reference numbers or authenticate vendors. For example, when Finance Service calls `/authenticateVendorForFinanceService` to verify a vendor's token (for vendor portal requests), ProjectX validates the API key and the organisation context [93] .

In summary, these rules maintain that: only real vendors with completed shipments get payouts; date windows are correct; payouts aren't tampered with out-of-process; and financial data (like taxes, totals) remain consistent. The system will surface understandable errors for validation issues (e.g., if a user tries to update an already-approved payout, the API returns a specific error object with status `INVALID_PAYOUT_STATUS` and message explaining the rule [94]).

Updating and Approving Payouts (Review Process)

After calculation, payouts enter a **review stage** where operations/finance teams can adjust and then approve the amounts:

- **Viewing Details:** Using the `fetchPayoutDetail` API, the UI can display all information about a pending payout. This includes the breakdown of charges, the computed vs. final amounts, and references [33]. If the payout is pending, the UI will typically allow editing of certain fields (usually the payout amounts for each charge and adding remarks).

- **Adjustments via Update:** If the reviewer needs to correct any value (for example, waive a penalty, add an extra charge, or fix a distance), they can use the `POST /objectPayout/<source>/updatePayout` endpoint. This accepts a **PayoutUpdateDto** with a `payout_id` and a list of `charges` to update [95]. Each charge entry includes a `chargeId`, a new `payout` amount, and an optional `remark` [96]. The backend `handlePayoutUpdate` will:

- Verify the payout exists and belongs to the org (else return *Invalid payout id*) [97].
- Ensure it's still in `pending` status (not yet approved) [94]. If not, it returns an error as discussed.
- Compute the updated totals: it goes through the existing breakdown (the JSON array in DB) and for each charge that matches one in the update list, it updates the `payout` value to the new number and saves the old value in a `calculated_payout` field on that line (only if not already present) [98]. This means the first time a charge is overridden, it retains the original in a new field for traceability [99]. It also updates/overwrites any `remark` on that line with the new remark if provided.
- It then sums up all the `payout` values again to get a new total (`totalPayout`) [100].
- It similarly ensures the Payout's own `calculated_payout` (top-level field) is set if not already (i.e., if this is the first adjustment on this payout, it copies the current final_payout into calculated_payout to preserve the original total) [101].
- Sets the Payout's `final_payout` to the new total and the `payout` breakdown to the adjusted list [101].
- Saves these changes to the database (within a transaction) and returns success [102] [103].

After an update, the payout record's `final_payout` reflects the adjusted amount, while the original computed amount is still available in `calculated_payout`. Each adjusted line item in the breakdown now contains both the original (as `calculated_payout` in the JSON) and the new value (as `payout`), along with any remarks explaining the change [99]. This provides an audit trail for the adjustment. For example, if a fuel surcharge was originally ₹500 but the vendor provided receipts for fuel, an ops user might reduce it to ₹0 – the breakdown line for fuel_surcharge would then have `calculated_payout: 500, payout: 0, remark: "Fuel provided by vendor"`.

The update mechanism thus allows fine-tuning the payout while keeping track of what was system-calculated. It's important to note that only certain users have permission to do this (permission `UPDATE_VENDOR_PAYOUT` is required, otherwise the UI won't enable the option).

- **Approval:** Once satisfied, the team can **approve** the payout. This is done via `POST /objectPayout/<source>/approvePayout` with a simple payload of the payout_id. The service's `handlePayoutApprove` will:
- Check existence and org (like before) [104].
- Ensure status is `pending` [105] (if someone tries to approve twice or a wrong payout, it errors).
- If valid, update the payout's status to `approved` [106].

- Commit and return success [106].

After this, the payout is marked approved and is essentially locked from further editing (the update API would now refuse it since status is no longer pending). In the UI, an approved payout might move out of the "Pending Review" list into an "Approved" bucket. The permission needed for approval is `APPROVE_VENDOR_PAYOUT`, which might be restricted to a manager role.

- **Rejection:** How to handle a payout that is incorrect beyond adjustment? The system did not implement a direct `rejectPayout` endpoint. If a payout was calculated that should not be paid at all, one approach is to simply not approve it (it stays pending). If truly invalid, ops could mark it as rejected in the database or delete it, but those aren't exposed. It's likely that if a payout is wrong, the ops team will adjust all values to 0 and approve (effectively nullifying it), or just leave it pending and not include it in any invoice. In future, a reject could be implemented by setting status = rejected, but business-wise an unapproved payout already means "not going to pay".

- **Status Tracking:** The Finance service provides a "bucket" concept for UI to group payouts. For ops users, buckets include **pending (Review Pending)**, **approved**, and **invoice_generated (Invoiced)** [107] [108]. For vendors, they might only see **approved** and **invoice_generated** buckets (since they are not involved in the internal pending review) [109]. These bucket mappings are returned in the masterdata API [110] [111], so the UI can label tabs accordingly. As payouts move through statuses via approve or invoice linking, they automatically appear in the respective bucket on the frontend.

To illustrate, consider a workflow: A payout of ₹5000 is calculated for vendor X. On review, the ops analyst realizes ₹100 of penalty was mistakenly applied. They use *updatePayout* to remove that penalty (setting its amount to 0, adding a remark). The system sets final_payout to ₹4900 and retains calculated_payout = ₹5000 for reference [112]. The analyst then clicks "Approve". The payout status flips to **approved** [105]. Now this payout is ready to be invoiced. The original breakdown still shows the penalty line but as 0 (with a remark indicating why), providing transparency that a ₹100 was forgiven.

This review process adds a human check to the automated calculations, which is critical in financial workflows. The log of changes ensures accountability.

Integration with Invoicing and Payment

Once payouts are approved, the next step is to consolidate them into an **invoice** for the vendor, which can then be sent and eventually paid. Shipsy's finance module manages this through an **ObjectInvoice** entity and related APIs.

- **Invoice Creation (Ops Trigger):** An authorized user (with `CREATE_VENDOR_INVOICE` permission) can initiate invoice generation by calling `POST /objectInvoice/<source>/createInvoice` with a **CreateInvoiceDto**. This typically includes:
- object_type = "vendor" (for carrier payouts),
- object_id = the Vendor's ID,
- start_date and end_date for the billing cycle to invoice [113] [114]. The service `createObjectInvoice` will first ensure no invoice already exists for that vendor and period (not CANCELLED) [85]. Then it finds all **approved** payouts for that vendor in that date range (event_date between start and end, invoice_id is null, status = approved) [115] [116]. If none are

found, it errors out (*"Payout not found for the given date range"*) [117] – you can only invoice after approving payouts. Assuming there are some, it proceeds to compute invoice totals:

- It queries the vendor's info to get tax rate and type [45]. For example, tax_rate 18%, tax_type "GST".
- It sums up all the selected payouts' final_payout amounts as the **net amount** (since net = pre-tax in this context) [118] [119]. Say we have two payouts ₹3000 and ₹2000, net_amount = ₹5000.
- Calculates **total_tax** = net_amount * (tax_rate/100) [120]. With 18%, that'd be ₹900.
- Computes **final_amount** = net_amount + total_tax (₹5900 in this example) [120].
- Generates a unique **invoice_number** (e.g. "INVabcd12345") [90].
- Creates a new Invoice record with status `created`, storing these values (organisation_id, vendor as object_id/type, creator info, start_date, end_date, amounts, tax details) [121] [122]. The event_date for the invoice is likely set to current date (or possibly end_date; the code sets event_date in Invoice as a Date – often used as creation date or invoice date).
- Inserts the Invoice into DB and obtains the new invoice_id [123].
- Then updates all the selected Payouts: sets their status to `invoice_generated` and attaches the invoice_id to them [124]. This links those payouts to this invoice and moves them out of the "Approved" state (so they won't be accidentally invoiced again).
- Creates an "Invoice Created" event for logging/auditing purposes [125].
- Commits the transaction and returns success with the invoiceId and a message [126].

At this point, an **Invoice** record exists representing, say, "Vendor X's payout for Jan 2025" with an invoice number and total amount ₹5900. All underlying payouts now show status `invoice_generated` and have the invoice_number (and ID) linked [34]. The invoice is initially in `created` status.

- **Vendor Action – Raising Invoice:** Now, the vendor typically needs to confirm or "raise" this invoice on their end (especially if the platform allows the vendor to formally submit the invoice). In Shipsy's flow, the vendor portal user can log in, see the invoice in a "To be Raised" state (in vendor's terminology). The vendor can review the invoice details (which would list all the payouts part of it, possibly via an Excel if attached or via the breakdown visible). If everything is acceptable, the vendor clicks to **Submit/Raise Invoice**. This calls the `submitInvoice` endpoint (`/objectInvoice/vendor/submitInvoice`) with an action_type of `raise_invoice` (or `resubmit` if it was rejected before) [127]. Under the hood, this triggers `updateInvoiceData(... action_type: RAISE ...)`. The InvoiceActionType mapping specifies that a RAISE action will move status from `created` to `raised` [128] [129]. The system requires that the current status is `created` (for raise) or `dispute_raised` (for resubmit) [128]. Upon success, the invoice status becomes **raised**.

"Raised" essentially means the vendor has submitted the invoice for approval. If the vendor were to dispute some entries at this stage, they might not raise it and instead possibly contact ops, or use a dispute mechanism (see below).

- **Internal Approval of Invoice:** Once an invoice is raised by the vendor, the internal team can **approve the invoice for payment**. This is done with `POST /objectInvoice/ops/approveInvoice` (permission `APPROVE_VENDOR_INVOICE`). That sets action_type = APPROVE, which will transition the status from `raised` to **approved** [130] [129], provided the current status is `raised` (which it is after vendor raise) [131]. Approving an invoice likely indicates that the finance team has vetted the invoice and it's ready for payout disbursal.

- **Disputes:** If the vendor finds an issue in the invoice (e.g., an incorrect charge that was not resolved earlier), they have options to raise disputes:

- They can raise a **Invoice Dispute** before it's approved – calling `/raiseDispute` (permission `RAISE_VENDOR_INVOICE_DISPUTE`). This sets action_type RAISE_DISPUTE, moving status from `raised` to `dispute_raised` [132] [133]. Essentially the vendor rejects the invoice figures. The ops team would then need to investigate, adjust something (possibly via updating underlying payouts or adding a credit note), and then allow the vendor to resubmit.

- If an invoice was already approved and even paid, there is also a concept of **Settlement Dispute** (`raiseSettlementDispute`, permission `VENDOR_INVOICE_SETTLEMENT_DISPUTE`) [134] which marks it as `settlement_dispute` (from paid status) [135]. This is a more advanced scenario: e.g. vendor claims the payment received doesn't match, etc. The system allows recording that dispute. In either case, there are corresponding resolution steps not fully detailed here (likely off-platform resolution and then updating the invoice or canceling it).

- **Payment and Settlement:** After an invoice is approved, the actual payment can be made through the company's finance system (could be offline or integrated). The Shipsy finance module allows recording the payment:

- The `invoicePayment` API (permission `VENDOR_INVOICE_PAYMENT`) can be called with transaction details to mark the invoice as paid [136]. This sets status to **paid** and logs the payment (e.g., transaction ID, amount) in a transaction log table.
- Once paid, the final step is marking it **settled** (if applicable). "Settled" might mean the vendor has confirmed receipt or the payment cycle is closed. The `settleInvoice` endpoint (permission `SETTLE_VENDOR_INVOICE`) moves status from paid to **settled** [137] [135].

At this point, the lifecycle is complete: payouts were calculated, approved, invoiced, and the invoice paid.

- **Invoice Documents:** The system, as we saw, can generate supporting documents. In Finance Service, after creating the invoice record, it immediately calls `generateSupportingCalculationExcel` in the background [126]. This function composes an SQL query to fetch all payouts under that invoice (status = invoice_generated for that vendor & period) and selects fields like reference_number, event_date, payout breakdown, etc. [138]. It then submits a download job to produce an Excel (with possibly all those details) and store it, attaching the resulting file URL to the invoice via a callback. The invoice entity has fields for `invoice_computation_excel_url` and `invoice_pdf_url` [139]. So the fully integrated solution would attach the Excel of payout calculations and perhaps a PDF of the invoice. In ProjectX, we saw code to generate a PDF invoice for the vendor in `generateVendorPayoutPdfInvoice` [140] [141] (using a label generation service with a template), which includes the vendor's GSTIN, address, a table of charge heads with net, tax, total columns for each, and totals [76] [142]. The finance microservice might eventually use its own templating or simply store the ProjectX-generated PDF link in `invoice_pdf_url`. Indeed, if `allow_vendor_invoice_in_pdf_format` is true, ProjectX returns a `vendorPayoutInvoicePdfUrl` [143] [49] which could be passed to Finance. However, since Finance also calculates tax and totals, it might generate its own PDF in future.

- **Cancel/Redo:** If an invoice is wrong or needs change after creation but before payment, the ops team can **cancel** it. Cancellation isn't directly shown in the snippet above, but the InvoiceStatus has `cancelled`. Likely there is a flow where if an invoice is in created/raised and issues are found (or vendor refuses to raise it), ops can cancel it (setting status cancelled). When an invoice is cancelled, the system should disassociate the payouts so they can be recalculated or re-invoiced. From the createInvoice logic, we infer cancelled invoices are ignored when checking

duplicates [85], so one can create a new invoice for that period if the previous was cancelled. The payouts probably remain in approved status but with invoice_id cleared. (The code for cancel wasn't shown, but presumably `updateInvoiceData` with a cancel action would set status cancelled and perhaps set associated payouts' invoice_id back to null and status back to approved to reopen them – that would be logical to allow corrections.)

In summary, the **invoicing stage ties together multiple payouts** and applies **taxation and document generation** to produce an official invoice for the vendor. It also provides a structured workflow for vendor acknowledgment (raise) and final approval/payment. All of this is integrated so that once an invoice is raised, the vendor can no longer individually dispute a payout – they'd dispute the invoice as a whole if needed. The payouts move into an invoiced state, effectively finalizing their values in that context.

The combination of payout records and invoices ensures both line-level detail and consolidated financial records: - Payout = micro-level (per trip or group) financial computation (which can be numerous). - Invoice = macro-level (aggregate) financial record used for settlement (usually one per billing cycle per vendor).

The system keeps these linked (payouts carry invoice_id). This also means one invoice could cover multiple payout records (e.g., if you calculated trip-level payouts for each trip, you might invoice many together; or if you did one monthly payout, the invoice might cover just that one).

Frontend Flows (CRM Ops Portal & Vendor Portal)

Operations (CRM) Frontend: The internal users interact with the payout system via a dedicated UI module, which leverages the aforementioned APIs and enforces the business logic on the front-end side as well: - In the CRM web app, a **Vendor Payout** section allows users to initiate calculations and manage payouts. When an ops user navigates to this section, the frontend first loads master data by calling `objectPayout/<source>/masterdata` [144] [145]. This returns: - Permission-based toggles (e.g. `enable_vendor_payout_update`, `enable_vendor_payout_approve`) indicating what actions the logged-in user can perform [110]. - The bucket mappings and **table column definitions** for the payout list view [146]. The columns define what fields are shown. For example, for ops users, columns include *Carrier Type*, *Carrier*, *Calculated For* (the payout cycle or period), *Entity Type* (Trip/Order), *Reference Number* (trip or order ref), *Group Reference* (if payouts are grouped), *Created At*, *Calculated Amount*, *Approved Amount*, etc. [147] [148] [149]. Some columns apply only in certain buckets (e.g. *Approved Amount* appears in the Approved bucket) [150]. - This metadata drives a dynamic UI table where each payout is a row and the appropriate columns are shown.

- **Initiating Calculations:** On the UI, a user can select a vendor (or multiple) and specify a date range (and possibly a hub filter and trip status, typically defaulted to Completed) and click "Calculate Payouts". The frontend likely calls the ProjectX Dashboard API (`/vendor-payout/v1/calculate`) as we saw [50], since that supports multiple selections and asynchronous processing. It passes `vendor_id_list`, `from_date_string`, `to_date_string`, and `planning_hub_id` if filtered, as well as `trip_status`. The UI might present a modal form for these inputs. If the API returns success, the UI gets a response listing which vendors' payout requests were queued (and any failures) [151] [152]. For each success, it might show a notification like "Payout calculation for Vendor X initiated". The actual calculations happen in the background, so the UI may not immediately show the results. However, the system inserted a log (VendorPayoutLog with a UUID and status PROCESSING) [52]. The UI could periodically poll a "status" or simply wait for the user to refresh the list.

- **Viewing Payouts List:** The payout list view (table) can filter by date range, status bucket, vendor, etc. When the user inputs a filter (e.g. choose bucket = pending and a date window), the frontend calls `fetchPayouts` with those parameters [153]. The backend responds with a list of payout summary items. The code assembles each item possibly with vendor info:

- If the request is from ops, after querying payouts it fetches the vendors' code and name for each unique object_id in the results (via the same FinanceIntegration fetch) [154] [155]. It then attaches `object_type_code` (probably vendor code) and `object_type_name` (vendor name) to each payout item in the list [156] [157]. So the table can display "Carrier: ABC Logistics" for object_id etc.
- The list items include id, entity_reference_number, calculated_payout, final_payout, status, etc., already formatted for display [158].
- Pagination is handled (with limit and page_number), and the API indicates if more pages are present [159] [160].

The UI will show, for example, all pending payouts in the selected date range with columns like Vendor, Reference (trip ID), Calculated Amount, etc. Pending payouts will have *Calculated Amount* = *Approved Amount* (since not yet changed or same as final) and status "Review Pending".

- **Reviewing and Editing:** The user can click on a payout entry to open a detail view (or drawer). The frontend calls `fetchPayoutDetail` with the payout ID [161]. The backend returns the full Payout data including the breakdown array and any remarks, along with vendor name/code and invoice number (if any) [162] [34]. The UI likely presents this as a list of charges with amounts. If `enable_vendor_payout_update` is true for the user, the UI would allow editing these amounts. This could be implemented as an inline editable table of charges or an "Adjust Payout" form. The user might see each charge (e.g. "Base Fare: 4500, Fuel Surcharge: 500, Penalty: -100") and could edit the values or add a remark why. Once they make changes, they hit save, and the frontend sends the `updatePayout` request with the payout_id and modified charges. If the response comes back success, the UI updates the values in the detail view and perhaps marks that payout as modified.

The UI might also highlight changes (since the backend now provides original vs new in the payout breakdown JSON, the UI could potentially show original values struck-through or in tooltip). The remark field from the update could be displayed as well (so everyone knows why an adjustment was made).

- **Approving Payout (Ops):** In the payout detail view or list, an "Approve" action (if `enable_vendor_payout_approve` is true) lets the user approve the payout. This triggers the `approvePayout` API [163]. On success, the UI likely removes that payout from the "Pending" list and it will appear in the "Approved" bucket. The UI may show a success message. The payout detail might become read-only after approval (no edit allowed).

- **Generating Invoice (Ops):** The CRM frontend also has an **Invoice module or section** for vendor payments. After approving payouts, the user switches to an Invoice screen to raise an invoice. They will typically choose the vendor and date range for invoicing (usually matching the cycle, e.g. the same month or week used for payouts). The UI calls `createInvoice` (on the finance service) with those parameters [113]. If the backend responds with success and an invoice ID, the UI shows the new invoice (perhaps in an invoice list). It might also immediately call `fetchInvoiceDetail` or refresh the invoice list to include this invoice.

On invoice creation, since Finance Service kicks off generating the supporting Excel, the UI might not have it immediately. The UI can call `getS3SignedUrl` for the invoice PDF/Excel when the user requests to download, or it might poll the Invoice detail until `invoice_computation_excel_url` is

populated. The presence of an asynchronous callback (Constants.INVOICE_EXCEL_UPDATE_CALLBACK) suggests the file URL will be populated after generation.

- **Invoice List & Detail (Ops):** Similar to payouts, there's `fetchInvoices` and `fetchInvoiceDetail`. The UI can filter invoices by status (all/raised/approved/etc), date type (invoice event_date vs billing_cycle range) [164], and vendor. It gets a list of invoices with key details: invoice number, status, amount, dates, etc. The invoice detail would include line items or references to payouts (perhaps via transaction logs or a separate call to list included payouts). We see `transaction_details` being fetched per invoice [165], likely listing payments or events on that invoice. The UI for invoice detail might present the summary of net/tax/total and links to download invoice PDF or calculation Excel. It also shows status and allows actions like Approve, Mark Paid, etc., depending on permissions.

- **Approving Invoice (Ops):** On an invoice detail page, an ops user with permission can click "Approve Invoice" (after vendor has raised it). This triggers `approveInvoice` API [130], moving status to approved. The UI updates the status and might notify that the invoice is now ready for payment.

- **Vendor Portal Frontend:** On the vendor side, there is a separate login (or portal UI). The vendor sees only their own data and with limited actions. Based on the connected permissions in the vendor middleware [166], vendor users have `READ_VENDOR_PAYOUT`, `READ_VENDOR_INVOICE`, `RAISE_VENDOR_INVOICE`, etc. but not the update or approve permissions (those are internal). This means:

- The vendor's **Payout view** likely shows *Approved payouts* (and possibly Invoiced). When the vendor logs in and navigates to payouts, the frontend calls `fetchPayouts` with source=vendor. The backend in that case automatically filters payouts to that vendor's userId (object_id = user's id) [167] [168]. Also, masterdata for vendor will only include buckets "approved" and "invoiced" [169]. The vendor can see which payouts have been approved by ops, along with their reference numbers, amounts, dates, etc. They likely cannot see pending ones (since those are not yet confirmed).
- The vendor can click an approved payout to view detail (`fetchPayoutDetail` with source=vendor will enforce that the payout's object_id matches the user, ensuring they only fetch their own [170]). In detail, they can see the breakdown but the UI for them would be read-only (no edit, no approve button). They might see remarks that ops added. This gives transparency (the vendor can understand how the payout was calculated).
- The vendor's main interaction is with **Invoices**. In the vendor portal invoice section, they call `fetchInvoices` with source=vendor; the backend then filters invoices to object_id = user (so only their invoices) [171]. The buckets for vendor invoices are mapped differently – for example, `created` status might be shown as "To be Raised", `dispute_raised` as "Dispute Raised", `paid` as "Settlement Pending", etc. [172]. So the vendor might see an invoice in "To be Raised" status when ops have created it.
- The vendor opens the invoice detail (`fetchInvoiceDetail` source=vendor, which ensures the invoice's object_id = userId) [173]. They can see the invoice number, period, list of payouts (maybe as attachments or listed charges), the net amount, tax, total, and any documents (Excel/PDF).
- If everything is acceptable, the vendor clicks "Raise Invoice" (or "Submit"). This triggers the `submitInvoice` with action_type "raise_invoice" [127]. If successful, the UI will mark that invoice as Raised (and maybe move it to a different bucket/state).

- If the vendor finds an issue, they might click "Dispute" instead. This would call `raiseDispute` [132], moving it to dispute_raised. The UI would indicate the invoice is now under dispute (and ops would be alerted to it likely through a dashboard or the status change).
- Once an invoice is in raised status (i.e., vendor has sent it), the vendor waits for payment. They can monitor the invoice status. When ops approve it, vendor might see it move to "Approved/Pending Payment". When payment is recorded (paid), vendor sees maybe "Settlement Pending" (since the money is on its way). When marked settled, it might show as "Settled/Paid". These correspond to the status mapping provided to vendor UI [174] [175].

- The vendor portal likely also shows some aggregate info like total due, etc., but that's beyond core logic.

- **Error Handling UX:** The frontends handle error messages returned by the backend. For instance, if a user tries to update a payout that was already approved (maybe the page was stale), the backend returns `{ success: false, status: 'INVALID_PAYOUT_STATUS', message: 'Payout update allowed only for pending status' }` [94]. The UI would display that message to the user (probably as a popup "Error: Payout update allowed only for pending status"). Similarly for attempts to approve an already approved payout or other invalid actions. The UI likely prevents most such actions (buttons get disabled once state changes), but these server-side checks ensure consistency.

All these interactions make the system user-friendly while enforcing the business rules. The operations team can easily calculate and adjust payouts without SQL or spreadsheets – the system generates the breakdown automatically. The vendor gets full visibility: they see exactly how their payout was calculated (each trip and charge), building trust [80]. The workflow of raise/approve for invoices aligns with common financial approval processes.

End-to-End Workflow Example

Finally, to tie it all together, here's an **example scenario** that demonstrates the journey from shipment completion to payout to invoice payment, highlighting the cross-service interactions:

1. **Shipment Completion:** Vendor ABC completes 10 trips in January. As each trip is completed in ProjectX, it's marked with status "completed" and all relevant data (distance, etc.) is stored. These trips are now eligible for payout. (No payout record is created yet, but data is ready. If the contract had any per-trip immediate components, those could be logged, but here we wait to end of month.)
2. **Initiating Payout Calculation:** On Feb 1, an Ops user logs into Shipsy CRM and navigates to **Vendor Payouts**. They select *Vendor ABC*, choose date range 01-Jan to 31-Jan, and confirm the status filter as *Completed*. They click "Calculate Payout". The system (CRM backend) invokes the ProjectX `calculateVendorPayout` API, which in turn starts an async job for ABC [51]. A log entry is created with a unique UUID for this calculation run [52].
3. **Computing Payouts (ProjectX job):** The async job fetches all ABC's completed trips in Jan (10 trips) [42]. It retrieves ABC's payout configuration (for example, suppose ABC's contract says ₹50 per trip fixed, ₹10 per km, and ₹100 penalty for delays >1h, with a monthly minimum of ₹6000). The job groups trips by relevant dimension (maybe all in one group since same contract) and calls the pricing logic. Let's say total distance = 500 km for those trips, so distance charge = 500*₹10* = *₹5000, trip count charge* = *10*₹50 = ₹500. Two of the trips had delays incurring penalties of ₹100 each = ₹200 total penalties. The minimum guarantee is ₹6000, and the computed net (5000+500-200 = ₹5300) falls below that, so an additional adjustment of ₹700 is added to meet ₹6000. The pricing returns charges: Base (distance) ₹5000, TripCount ₹500, DelayPenalty ₹200

(as a positive number for total penalty which will be negated), MinGuaranteeAdjustment ₹700. Final total = ₹6000. ProjectX compiles an Excel with each trip's data and these charge components. It then uploads the Excel and also generates a PDF invoice for this payout (since allow_invoice_pdf is true) [79] [49]. It logs each trip's charges back to the trip records (marking those trips as accounted for) [54].

4. **Payout Record Creation:** Because this is the older batch method, ProjectX directly created a **Vendor Payout Invoice PDF** for Jan. However, in a Finance Service-centric approach, it would instead create one payout record for that monthly period. Let's assume the integration is such that Finance Service is informed. The Finance Service could have been called for each trip (if configured that way) or for the aggregate. In our scenario, perhaps the *monthly* payout was created via a single Finance API call (alternatively, the ops user could have used the Finance service directly with payout_type "monthly"). Finance `handlePayoutCalculation` would take the aggregated metrics (total distance etc.) and produce a payout record:
5. final_payout = ₹6000, calculated_payout = ₹6000, status = pending [29].
6. breakdown: baseFare ₹5000, trip_count ₹500, additional_trip_penalties -₹200, min_guarantee ₹700 [31].
7. computation JSON stores e.g. payout_type: monthly, start_date: Jan1, end_date: Jan31, etc., and vendor, etc.
8. entity_reference might be blank or a group reference (some implementations treat a monthly payout as object_id = vendor and a pseudo entity like group_id for that month).

9. The system returns success of calculation.

10. **Review & Adjust:** The ops user refreshes the Payouts screen. The new payout for Vendor ABC Jan-2025 appears in the **Review Pending** list (status pending). They open it and see:

11. Reference Number: (maybe blank or a generated group ID).
12. Calculated Amount: ₹6000.

13. Breakdown: Base ₹5000, TripCount ₹500, DelayPenalty -₹200, MinGuarantee +₹700. They verify the trips and charges using the Excel link (if provided). Suppose they realize one penalty was incorrectly applied (perhaps one delay was excused). They decide to remove ₹100 of the penalties. They click "Edit" and change the DelayPenalty from ₹200 to ₹100, adding a remark "Waived penalty for Trip #TRIP8". They save; the system updates the payout: original calculated_payout (6000) is preserved internally, final_payout becomes ₹6100 - ₹100 waived = ₹5900 (actually, in this case, removing a penalty *increases* the payout to ₹6100, since we waived a deduction; so final payout goes **up** by ₹100). The breakdown now has DelayPenalty as -₹100 (with remark) and likely MinGuaranteeAdjustment might have adjusted: actually, if we removed a penalty, the vendor was already at minimum guarantee ₹6000, now without that ₹100 penalty the computed would have been ₹5400, requiring ₹600 adjustment instead of ₹700. However, since we manually override, we might either adjust the MinGuarantee accordingly or leave it. This highlights a complexity: manual changes can upset the original guarantee logic. In practice, the ops might recalc the guarantee manually. Let's say they leave MinGuarantee at ₹700, which means effectively paying ₹6100 total, ₹100 above guarantee. It's a business decision. The system doesn't recalc other fields on manual edit; it's up to ops to adjust any related line if needed.) Anyway, the payout final_payout is now ₹6100 and status still pending.

14. **Approval of Payout:** The ops user is satisfied and clicks "Approve Payout". The Finance service sets that payout status to approved [105]. It now moves to the Approved bucket. No further changes are possible on it.

15. **Invoice Generation:** The ops user goes to the Invoice module and creates an invoice for Vendor ABC for Jan 1–Jan 31. The system finds the one approved payout for that vendor in that range (now with final_payout ₹6100) [115]. It calculates GST: vendor ABC has tax_rate 18%, so net = 6100, tax = ₹1098, total = ₹7198. It inserts an Invoice with number INV0001..., net_amount 6100, total_tax 1098, final_amount 7198, status created [120] [121]. It links the payout to this invoice (payout status -> invoice_generated) [124]. The ops user sees the invoice record created.

16. **Vendor Raises Invoice:** Vendor ABC logs into the portal and sees an invoice "To be Raised" for ₹7198 for Jan 2025. They open it and review the details: perhaps they download the supporting Excel which lists all 10 trips with the calculations (the Finance service's generateSupportingCalculationExcel would have prepared a dump of the payout data [138]). They find everything in order (the waived penalty was communicated). They click "Raise Invoice" to indicate acceptance [127]. The invoice status becomes raised.

17. **Ops Approve Invoice:** The ops manager sees the invoice now marked as raised (waiting for approval). They double-check that vendor's GSTIN, amounts, etc., are correct. They then click "Approve Invoice for Payment" [130]. The system sets status to approved. Now the invoice is ready for payment. The payouts under it remain invoice_generated and are essentially finalized.

18. **Payment Execution:** The finance team then issues payment of ₹7198 to vendor ABC externally (through bank or integrated payment). Once done, they record it in Shipsy: e.g., by clicking "Mark Paid" and entering transaction details (UTR number, date). The `invoicePayment` API is called, adding a transaction log and setting status to paid [136]. Optionally, once the vendor confirms receipt, the team marks it "Settled" [137]. Now the invoice is settled/closed.

19. **Vendor Confirmation:** Vendor ABC sees the invoice move to Settled in their portal (meaning the payment cycle concluded). They also see that the payout entries in their "Approved" payouts list likely now show as "Invoiced" or tied to that invoice number.

20. **Future cycles:** Next month, the process repeats for February trips.

Throughout this example, we saw multiple components interacting: - **ProjectX** handled trip completion data and heavy-duty calculations (Excel/PDF generation and optional grouping logic). - **Finance Service** recorded the payout and invoice data and enforced the review/approval workflow and integration with vendor portal. - **CRM Frontend** provided the interface for ops to trigger and manage these steps, using the appropriate service calls. - **Vendor Frontend** allowed the vendor to see and acknowledge the payouts and invoices. - **Validations** ensured things happened in the correct order (couldn't invoice before approving payouts, couldn't approve twice, etc.).

In conclusion, the Carrier/Vendor Payout logic in Shipsy is a comprehensive system that ensures vendors are paid accurately and transparently for their services. It integrates operational events (trip completion), contract-based rate calculations, human review with adjustments, and financial reconciliation (invoicing & payment) into one seamless flow. This automation minimizes manual billing effort and errors, as highlighted by the Shipsy release: clients can **"automatically generate vendor payouts"** and enjoy *accurate and transparent finance operations*, with granular visibility for both clients and vendors into how each payout is computed [38] [176]. The combination of backend services and frontend modules achieves a controlled yet efficient payout process, from trigger to treasury.

---

1 2 3 19 20 21 24 25 26 28 payout.entity.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/entities/payout.entity.ts

4 5 6 7 8 9 10 11 12 13 14 15 object-payout.dto.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectPayout/dto/object-payout.dto.ts

16 17 18 22 23 27 29 30 32 33 34 35 40 41 61 62 63 66 67 68 69 86 87 88 89 94 97 98 99 100 101 102 103 104 105 106 110 111 112 145 146 158 162 167 168 170 objectPayout.service.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectPayout/objectPayout.service.ts

31 39 42 43 44 46 47 48 49 52 53 54 55 56 57 58 70 71 72 73 74 75 76 77 78 79 82 83 140 141 142 143 vendor-payout-handler.js

https://github.com/shipsy/projectx/blob/3ac216c8ebc368c0b4cde2ff7d93e251b6c44dfd/common/domain-models/vendor-payout-handler/vendor-payout-handler.js

36 107 108 109 147 148 149 150 169 172 174 175 util.service.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/util/util.service.ts

37 url.service.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/url/url.service.ts

38 80 176 Say Goodbye to Manual Billings and Invoicing with Vendor Payout Module

https://product.shipsy.io/vendor-payout-module-for-api-based-automation-of-payout-trip-rates-calculation

45 84 85 90 115 116 117 118 119 120 121 122 123 124 125 126 138 154 155 156 157 159 160 164 165 171 173 objectInvoice.service.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectInvoice/objectInvoice.service.ts

50 51 81 151 152 vendor-payout-actions-api.js

https://github.com/shipsy/projectx/blob/3ac216c8ebc368c0b4cde2ff7d93e251b6c44dfd/common/models/dashboard-parts/vendor-payout-module-parts/vendor-payout-actions-api.js

59 60 144 153 161 163 objectPayout.controller.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectPayout/objectPayout.controller.ts

64 65 master-data-api.js

https://github.com/shipsy/projectx/blob/3ac216c8ebc368c0b4cde2ff7d93e251b6c44dfd/common/models/finance-service-integration-parts/master-data-api.js

91 92 93 finance-service-integration.js

https://github.com/shipsy/projectx/blob/3ac216c8ebc368c0b4cde2ff7d93e251b6c44dfd/common/models/finance-service-integration.js

95 96 payout-update.dto.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectPayout/dto/payout-update.dto.ts

113 114 127 130 132 134 136 137 objectInvoice.controller.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/objectInvoice/objectInvoice.controller.ts

128 129 131 133 135 139 invoice.entity.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/entities/invoice.entity.ts

166 vendor-middleware.ts

https://github.com/shipsy/shipsy-finance-service/blob/a6488a6a2f6edb94d11eeb9fb916e2cfc7f12f23/src/common/middlewares/authentication/vendor-middleware.ts