Carriage Ride System: Design Document

Version: 7.2.3

Date: September 08, 2025

Author: Desmond Atikpui

1. Ride System Overview

This document provides a complete technical design for the ride scheduling system. The system manages the lifecycle of a "Ride"- an event where a Rider (student) is transported between two locations by a Driver, with scheduling and oversight handled by an Admin (Dispatcher).

The design supports both single, one-off rides and complex recurring schedules, drawing from RFC 5545 for its recurrence logic. A key feature of this design is its event-driven architecture, ensuring all parties are notified of important changes in real-time.

1.0. Basic Ride lifecycle and workflow

Core Workflow

1. **Request Phase**
    - **Rider** submits a ride request with pickup/dropoff locations, preferred time and recurrence.
    - **Admin** receives the request for review and approval
    - The ride enters PENDING approval status and UNSCHEDULED state

2. Approval Phase

- **Admin** reviews the request against scheduling constraints and availability
- **Admin** can:
    - APPROVE the request as submitted
    - APPROVE_WITH_MODIFICATION (e.g., adjusted time due to conflicts)
    - REJECT the request

3. Scheduling Phase

- For approved rides, **Admin** assigns an available driver
- Ride transitions from UNSCHEDULED to SCHEDULED
- System validates no overlapping assignments for rider or driver
- All parties receive notifications of the assignment

4. Operational Phase (Active rides only)

- **Driver** updates operational status as the ride progresses:
  - NOT_STARTED → ON_THE_WAY → ARRIVED → PICKED_UP → COMPLETED
- **Rider** and **Admin** receive real-time status updates
- Alternative terminal states: CANCELLED, NO_SHOW, or ABORTED

5. Completion Phase

- Ride reaches COMPLETED status or a terminal state

1.1. Roles

The system is built around three user roles with distinct capabilities:

- **Rider (Student):**
  - They request rides.
  - Limited permissions and constrained by scheduling rules in **1.3**
- **Admin:**
  - They are responsible for assigning drivers, scheduling rides and managing the overall ride system.
- **Driver:**
  - Fulfills ride requests.
  - Responsible for updating the operational status of a ride during its operational phase.

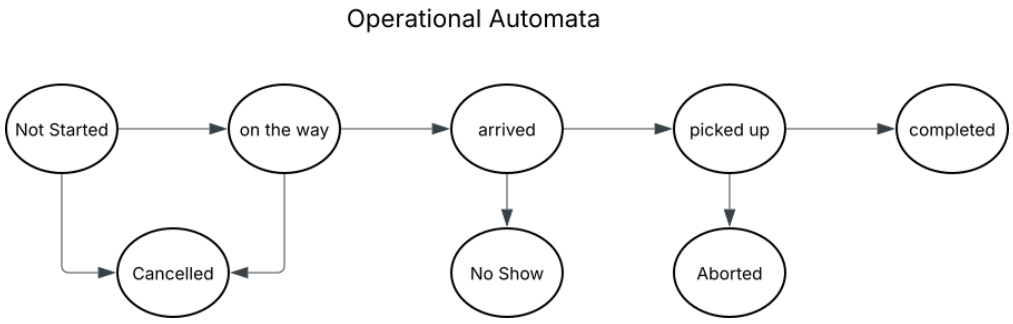
1.2. Ride States & Transitions

Every ride is defined by a set of states that evolves over its lifecycle.

- **Time State :** A derived property based on the current time relative to the ride's `startTime` and `endTime`.
  - `UPCOMING`: The ride's `startTime` is in the future.
  - `ACTIVE`: The current time is between the ride's `startTime` and `endTime`.
  - `PAST`: The ride's `endTime` is in the past.
- **Scheduling State:** Describes whether a driver has been assigned.
  - `UNSCHEDULED`: The initial state of a ride created by a Rider or an Admin without a driver.
  - `SCHEDULED`: An Admin has assigned a driver to the ride.
- **Approval Status :** The approval status of the ride, controlled ONLY by the Admin.
  - `PENDING`: A default ride request
  - `APPROVED`: An Admin approved the ride request as is.
  - `APPROVED_WITH_MODIFICATION`: An Admin approved the ride request but made modifications.
  - `REJECTED`: An Admin rejects the ride request by the Rider. Terminal State and never enters the Start_Ride Automata.
- **Operational Status :** The real-time operational state of the ride, primarily controlled by the Driver.

- **Linear States :** This is the state the ride is in when Active, exc. default
  - `NOT_STARTED`: The default status for any new ride.
  - `ON_THE_WAY`: The Driver has started heading towards the pickup location.
  - `ARRIVED`: The Driver has arrived at the pickup location.
  - `PICKED_UP`: The Rider is in the vehicle, and the journey to the destination has begun.
  - `COMPLETED`: The ride has successfully concluded at the destination.
- **Terminal States:** Once in a terminal state, a ride cannot change status, except if **Admin overrides** manually. Terminal States are only reachable when a Ride is in a **Scheduled** state.
  - `CANCELLED`: The ride was cancelled with Admin approval before the ARRIVED state. This state can be transitioned to from **NOT_STARTED** and **ON_THE_WAY**
  - `NO_SHOW`: The Driver arrived, but the Rider was not present for pickup after a grace period. This state can only be transitioned to from **ARRIVED**
  - `ABORTED`: The ride was started but could not be completed due to an unforeseen issue (e.g., vehicle problem, rider emergency). This state can be transitioned to from On the way up to and not including Completed.

Rule : Terminal states cannot transition into each other



1.3. Scheduling Rules and Constraints

1.3.1. General Rules

- **No Overlaps:** A Rider or a Driver cannot be assigned to two rides that have overlapping times. The system must enforce this constraint via the `CollisionDetectionService`.
- **Blackout Periods:** Rides cannot be scheduled on certain days or during specific times. This will be managed by the `HolidayAndBlackoutService`.
- **Service window**: rides must start and end between **06:00 and 22:00**.

1.3.2. Role based Rules

- **Admin**
  - **Update Approval Status** : Can **APPROVE**, **APPROVE_WITH_MODIFICATION**, or **REJECT** any pending request.
  - **Update Operational Status** : Can change operational status for **ACTIVE** & **PAST** rides
- **Rider**
  - **Advance Notice**: A ride for day **D** must be requested before 10:00AM on the day **D-1**. e.g requesting a Tuesday ride must be done before 10:00AM on Monday
  - **Editing a Ride :** A rider can only edit a ride if it is **UNSCHEDULED.** Any edits that need to be made to a **SCHEDULED** ride including cancellations will be done by the Admin
- **Driver**
  - **Update Operational Status** : A driver can only update the operational status of a ride in **ACTIVE** state.

1.4. Ride Recurrence

1.4.1. Overview

- Recurrence in the Carriage Ride System is based on the iCalendar specification (RFC 5545), providing a robust and flexible way to define repeating ride schedules.
- A `RecurrenceRule` interface (as defined in Section 3.1.1) is used to specify parameters such as frequency (e.g., `DAILY`, `WEEKLY`), interval, end conditions (e.g., `until` date, `count`), and specific days of the week (`byday`). This allows for the creation of complex recurring patterns.
- **Mathematical Notation :** Let R be the recurrence rule, d_s the series start date, d_e the series end date, *E* the set of exceptionDates. The set of eligible materialization dates is defined as

$$\mathcal{D} = \{t \in R \mid d_s \leq t \leq d_e,\ t \notin E\}$$

1.4.2. Rides : Physical Rides and Virtual Rides

1.4.2.1 RideSeries

- **Definition** : A `RideSeries` object (Section 3.1.1) serves as an object for recurring rides. It defines the common attributes (locations, rider, start time) for a series of future rides. It also defines a recurrence pattern using RFC 5545 fields. These fields represent the “virtual” fields of a ride.
- **Future focus** : after a ride is materialized with startdate t, the system shifts the generation date of the series to the next recurrence date - 2. This ensures the series only considers future unmaterialized rides and prevents the accumulation of past rides into the exceptionDates list.
- **Expired** : a `RideSeries` is considered expired when its start date > end date. i.e all intended rides have been materialized.

1.4.2.2 Ride

- **Physical Ride**

  A `PhysicalRide` represents a single, concrete ride instance that exists on the **database**. It defines the physical features + virtual features of a ride.

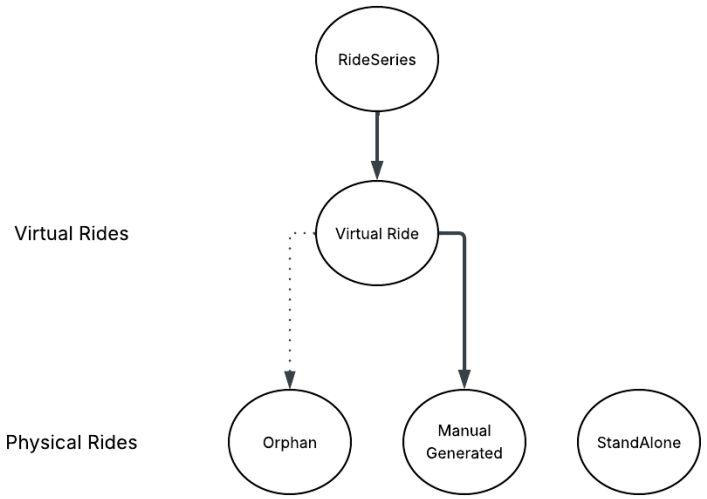
- **Types of Physical Rides**
  - **Standalone Rides:** One-off rides not associated with any series.
  - **Generated Rides:**
    - **Auto-Generated** : Individual instances that are generated via **auto-materialization** from a `RideSeries`.
    - **Manual-Generated** : Individual instances that are manually generated from a `RideSeries.`This ride maintains a link to the series.
    - **Orphaned Rides** : Individual instances generated from a `RideSeries` when a virtual ride **DIVERGES** from the series. This ride no longer has any link to the series.

  All physical rides follow the same schema and are distinguishable based on rules and not fields.

- **Virtual Ride**
  - **Definition :** A `VirtualRide` represents a single ride that does not exist on the database and only exist as part of a `RideSeries`
  - These are only for UI purposes and have the same schema as physical rides. Any interactions to the physical features will cause the ride to be made into a `PhysicalRide`
  - It can only be materialized into a generated ride.

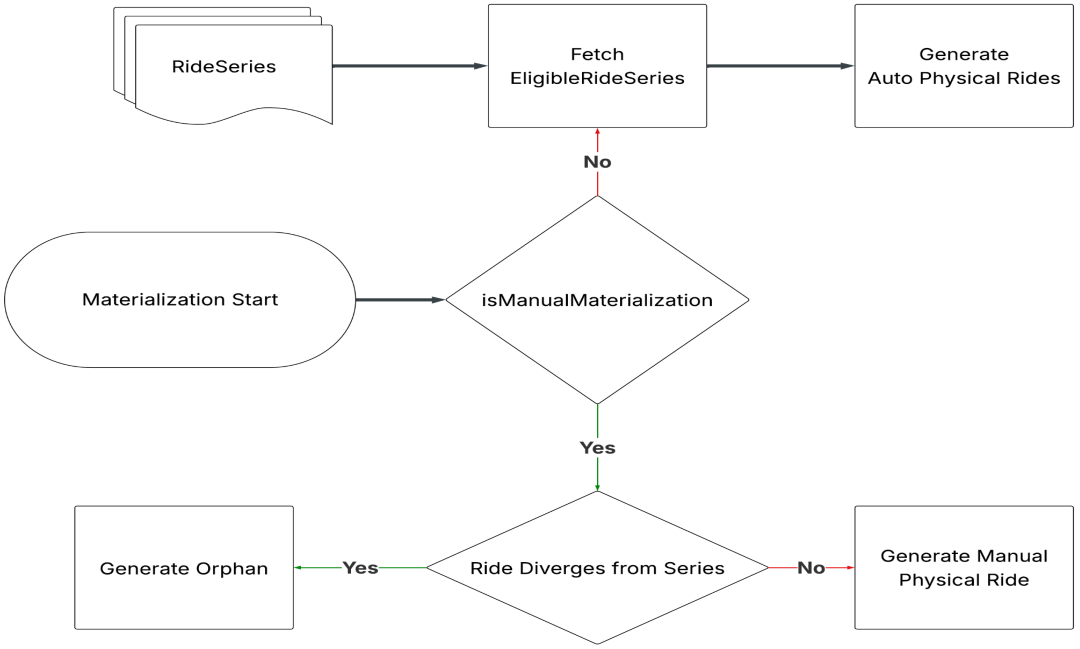
1.4.2.3 Features and Divergence

- A ride has two categories of fields
  - **Virtual Features** : These are template-aligned fields such as startTime, startLocation, endLocation, recurrence pattern.
  - **Physical Features** : These are ride-execution or logistical fields like assigned driver, scheduling state, operational status.
- A virtual ride diverges when any of its virtual features is edited before materialization. This leads to an **Orphaned Ride**. A virtual ride becomes physical (**Manual-Generated**) when only physical features are modified.



1.4.3. Ride Materialization

- **Definition :** The process of transforming a `VirtualRide` into a `PhysicalRide` based on a `RideSeries`. This can be done through manual materialization or auto-materialization.
- **Flowchart**



1.5.3.1 Manual Materialization

- **Definition** : Manual materialization happens when a user interacts with a `VirtualRide` (e.g via the UI), triggering a conversion to a `PhysicalRide`.
- **Types of Interactions**
  - **Changing physical features :** results in a Manual-Generated Ride

        - **Changing virtual features :** results in an Orphaned Ride, diverging from the series
    - **Rules**
        - For diverging rides, add ride date, t to the exceptionDates, E. This is to prevent materialization of the ride on the day by the auto-materializer.
        - For non-diverging rides, add ride date, *t* to the suppressedDates, *S*. This is to prevent duplicate generation by the auto-materializer.
    - **Pseudocode**

```
onVirtualRideInteraction(rideSeries, changes):
  if changes affect virtual features:
     materialize as OrphanedRide
     Add OrphanedRide.startTime to rideSeries.exceptionDates.
  else:
     materialize as ManualGeneratedRide
     add ManualGeneratedRide.startTime to rideSeries.suppressedDates
```

1.5.3.1 Auto Materialization

- **Definition** : Auto materialization is the system-driven process of converting `VirtualRides` into `PhysicalRides` based on a `RideSeries` recurrence rule. Auto-materialization happens on a defined schedule and is **forward-focused**.
- **Generation schedule** : Auto-materialization runs **once daily** across all eligible RideSeries to generate rides that are occurring the next day.
- **Rules**
    - Eligible `RideSeries`
        - RideSeries is **ACTIVE**
        - RideSeries.recurrence_generation_date = today
    - Eligible Ride
        - Date is a valid recurrence date
        - Date is not in the exceptionDates
        - Date is not in the suppressedDates
- **Generation Overview**
    - A RideSeries that passes the rules will have an eligible ride generated.
    - A ride from a ride series will be generated if it is not in exceptionDates or suppressedDates.
    - **Pruning and Shifting**
        - Shift recurrence_generation_date to the next recurrence_generation_date to keep the RideSeries forward-focused.
        - Prune S - remove entries strictly before the next Ride date since those dates are no longer relevant.
- **Pseudocode**

```
Function GenerateRide(RideSeries):
      for each RideSeries in system:
          if ( RideSeries is ACTIVE and RideSeries.recurrence_generation_date == today ):
               targetDate = getNextRideDateFromPattern(today,RideSeries.pattern)
```

```
          if (targetDate NOT in exceptionDates AND targetDate NOT in suppressedDates ) :
               generate PhysicalRide for targetDate
          setRecurrenceGenerationDate(RideSeries,targetDate)

Function setRecurrenceGenerationDate(RideSeries,date)
    nextRideDate = getNextRideDateFromPattern(date,RideSeries.pattern)
    if (nextRideDate):
         // shift
         RideSeries.recurrence_generation_date = nextRideDate - ADVANCE_NOTICE_RULE
         // in this case advance notice rule is 10:00AM the day before
         // prune
         RideSeries.suppressedDates = RideSeries.suppressedDates.filter(date >= nextRideDate)
    else:
         mark RideSeries as EXPIRED
```

1.5. Ride Creating, Editing and Deleting

1.5.1. Overview

The system supports CRUD operations with role-based permissions and special handling for recurring rides. Operations must respect the state-based rules and scheduling constraints defined in the system. The three update/delete operation types (Single, This and Following, All) are designed to handle recurring ride series appropriately.

Key Principles

- **Role-based permissions:** Each role has specific capabilities based on system rules
- **State-aware editing:** Editing permissions depend on current ride and scheduling state
- **Recurrence handling:** Operations can target individual rides or entire series
- **Constraint enforcement:** All operations must respect scheduling rules and constraints

1.5.2. Role-based CRUD Operations

1.5.2.1 Create

Admin

- Can create rides for any rider
- Can create RideSeries with RFC 5545 recurrence patterns
- Can assign drivers during ride creation
- Not subject to advance notice or other rider constraints

Rider

- Can create rides for themselves only
- Must comply with advance notice rule (before 10:00AM on day D-1)

- Must respect service window (06:00-22:00)
- Can create RideSeries with recurrence patterns
- Cannot assign drivers (rides created as UNSCHEDULED)

1.5.2.2 Read

Admin

- Can view all rides in the system
- Access to all ride states and operational details

Rider

- Can view their own rides
- Limited to rides where they are the assigned rider

Driver

- Can view rides assigned to them
- Access to operational details for their assigned rides

1.5.2.3 Update

General Constraints for Updates

Admin:

- Can update approval status (APPROVE, APPROVE_WITH_MODIFICATION, REJECT)
- Can change operational status for ACTIVE and PAST rides
- Can modify any ride attributes
- Can reassign drivers

Rider

- A rider can only update rides under these constraints
    - ApprovalState : PENDING
    - TimeStatus : Upcoming
    - SchedulingState : Unscheduled
    - OperationalStatus : Not_started
- Any changes they make are still subject to the Advance notice and service window constraints.

Update Single

- An Update Single interaction is when a specific Ride has its features changed. This can occur on any type of Ride as defined in 1.4.2.
- Always results in the creation of a PhysicalRide if it is done on a VirtualRide. Depending on the features updated (physical / virtual) it materializes the appropriate type of ride.
- Can affect all Rides in any state. User permissions and constraints still matter.

Update This and Following

- An Update This and Following interaction is when a Rides generated by a RideSeries have its features changed. This occurs on all rides except **Standalone Rides** and **Orphaned Rides**.
- Updates can only affect virtual features. Physical features cannot be updated for This and Following.
- Any updates will split the RideSeries at the update point: the original series ends at the last ride before the updated ride, and a new RideSeries begins with the updated ride and continues with the modified parameters.

Admin:

- Can modify recurring series.
- Can change virtual features.

Rider:

- Can modify their recurring series based on their role constraints.
- Can only modify virtual rides and not physical rides for This and Following.

Update All

- An Update All interaction is when Rides generated by a RideSeries have its features changed. This occurs on all rides except **Standalone Rides** and **Orphaned Rides**.
- Updates can only affect virtual features. Physical features have to be updated as and when they are encountered per ride.

Admin:

- Can modify entire recurring series
    - If any physical ride exists the virtual features are changed as well to match the RideSeries.
    - The RideSeries is then updated as brand new except with the PrimaryKey.
- Can change only virtual features

Rider:

- Can only Edit All if the entire series consists of virtual rides.
- Can only Edit All based on their constraints

1.5.2.4 Delete

- Delete has the same exact constraints and flows as Update.
- A deletion is the removal of a record from the Database.

Delete Single

Admin:

- Can delete any ride in any state

- For recurring rides, it will add the date to the exceptionDates of the RideSeries.

Rider:

- Can only delete rides based on their constraints.
- For recurring rides, deletion adds date to exceptionDates

Delete This and Following

- This and Following will terminate the RideSeries at the date occurring before the selected date. E.g If i have a daily ride and choose to delete from Wednesday forward, the RideSeries will terminate on Tuesday.

Admin:

- Can terminate recurring series from a specific date forward
- Effectively modifies the series end date

Rider:

- Can terminate their recurring series from a specific date forward based on their constraints.

Delete All

Admin:

- Can delete entire recurring series
- Can delete all associated rides

Rider:

- Can ONLY delete all if the entire series consists of virtual rides
- Can ONLY delete all based on their constraints

15.3. Constraints and Validation

- All operations must respect CollisionDetectionService constraints
- Changes must comply with HolidayAndBlackoutService restrictions
- Advance notice requirements apply to rider-initiated changes
- Feature changes in recurring rides may trigger divergence and orphaning depending on which features are changed (physical / virtual)

2. Frontend

3. Backend

3.1. Data Models and Types

This section defines the data models, enums and API view models required to implement the Carriage Ride System logic.

3.1.1 Enums

These represent the specific, controlled states defined in Section 1.2.

- **ApprovalStatus**: Defines the Admin-controlled approval state of a ride request.
  - `PENDING`: The default status for any new ride request.
  - `APPROVED`: Status for a ride approved by an Admin as submitted.
  - `APPROVED_WITH_MODIFICATION`: Status for a ride approved by an Admin, but with modifications.
  - `REJECTED`: A terminal state for a ride request rejected by an Admin.
- **SchedulingState**: Describes whether a driver has been assigned to the ride.
  - `UNSCHEDULED`: The initial state for a ride without a driver.
  - `SCHEDULED`: State indicating an Admin has assigned a driver to the ride.
- **OperationalStatus**: The real-time operational state, primarily controlled by the Driver.
  - `NOT_STARTED`: The default status for any new ride.
  - `ON_THE_WAY`: The Driver has started heading toward the pickup location.
  - `ARRIVED`: The Driver has arrived at the pickup location.
  - `PICKED_UP`: The Rider is in the vehicle, and the journey has begun.
  - `COMPLETED`: The ride has successfully concluded.
  - `CANCELLED`: Terminal state; the ride was cancelled with Admin approval before the `ARRIVED` state.
  - `NO_SHOW`: Terminal state; the Driver arrived, but the Rider was not present.
  - `ABORTED`: Terminal state; the ride started but could not be completed.
- **SeriesStatus** : The state of the series.
  - `ACTIVE` : The series still has future rides
  - `EXPIRED` : All rides for the series have been materialized. Formally, recurrenceGenerationDate > seriesEndDate.

3.1.2 Database Models (Schemas)

These represent the entities stored in the DynamoDB database.

3.1.2.1 RideSeries

Definition: This schema represents the `RideSeries` object, which serves as the template for recurring rides. It defines the recurrence logic and common attributes ("virtual fields") for the series .

- **Fields:**
  - `id`: Primary Key.
  - `riderId`: Foreign key relationship to the Rider (Student).
  - `seriesStatus`: The state of the series (e.g., `ACTIVE` or `EXPIRED`).

- `startLocationId`: Foreign key to the `Locations` table for the template pickup location.
- `endLocationId`: Foreign key to the `Locations` table for the template dropoff location.
- `templateStartTime`: The start time (e.g., "09:00") common to the series.
- `seriesStartDate`: The series start date (ds).
- `seriesEndDate`: The series end date (de).
- `recurrenceRule`:
  - An embedded object implementing the **RecurrenceRule interface**. This stores the atomic components of the RFC 5545 rule
  - `frequency` : The frequency of the recurrence (e.g DAILY, WEEKLY)
  - `interval`: The interval between recurrences.
  - `byday` : Specific days of the week for the recurrence.
  - `endConditions` : End conditions such as *counts* or an *until* date.
- `exceptionDates`: A set (E) of specific dates skipped by the materializer.
- `suppressedDates`: A set (S) of dates populated when a ride is manually materialized, used to prevent the auto-materializer from creating a duplicate.
- `recurrenceGenerationDate`: The date the auto-materializer uses to check for the next ride generation.

3.1.2.2 Ride (Physical)

Definition: This schema represents a `PhysicalRide`: a "single, concrete ride instance that exists on the database". This single schema supports all defined physical ride types (Standalone, Auto-Generated, Manual-Generated, and Orphaned).

- **Fields:**
  - `id`: Primary Key.
  - `rideSeriesId`: A nullable Foreign Key. This links to the parent `RideSeries` for Generated rides. This link is **null** for Standalone rides and Orphaned rides.
  - `riderId`: Foreign key to the Rider. (See justification below).
  - `driverId`: A nullable Foreign Key to the Driver. When null, this signifies the ride is `UNSCHEDULED`.
  - `startLocationId`: Foreign key to the `Locations` table for the instance-specific pickup location.
  - `endLocationId`: Foreign key to the `Locations` table for the instance-specific dropoff location.
  - `startTime`: The specific DATETIME the ride is scheduled to begin.
  - `endTime`: The estimated specific DATETIME the ride is scheduled to end.
  - `approvalStatus`: The Admin-controlled approval state (PENDING, APPROVED, etc.).

- `schedulingState`: The driver assignment status (UNSCHEDULED, SCHEDULED).
- `operationalStatus`: The real-time operational status (NOT_STARTED, ON_THE_WAY, etc.)

3.1.3. API and View Models

3.1.3.1. RideViewModel

Definition: This is the Data Transfer Object (DTO) used by the API to present ride data to the UI. This single structure represents both `PhysicalRide` objects (retrieved from the database) and `VirtualRide` objects (which "do not exist on the database" and are dynamically calculated from a `RideSeries`).

- **Schema:** This model has the "same schema as physical rides".
- **Differentiator:** The application logic must append an optional flag (e.g., `isVirtual: true`) to the objects representing `VirtualRides`. This signals the UI that the object is "only for UI purposes" and that any interaction with it must trigger the manual materialization process.

3.1.4. Physical and Virtual Feature Logic

This section defines the business logic required to implement the divergence and manual materialization rules specified in the design. These categories and constants are specifically used by the **`onVirtualRideInteraction`** pseudocode, which is triggered when a user edits a *single* `VirtualRide` instance.This logic determines if the materialized ride diverges from the series

Virtual Features:

- Defined by the document as "template-aligned fields". These are the *instance-level* fields on a single ride that define the template.
- **Specification (Pseudocode Constant):** The application logic must maintain a constant Set identifying these specific fields as "virtual":

```
const VIRTUAL_FEATURES_INSTANCE_LIST = new Set([
"startTime",
 "startLocationId",
  "endLocationId"
]);
```

- **Logic**: When an "Update Single" interaction modifies any field present in this `VIRTUAL_FEATURES_INSTANCE_LIST`, the interaction is considered a divergence. The system must "materialize as OrphanedRide" , and this new physical ride record "no longer has any link to the series".

Physical Features:

- Defined by the document as "ride-execution or logistical fields". These are fields that can be updated on a single instance without causing it to diverge from the series template logic.
- **Specification (Pseudocode Constant):** The application logic must maintain a constant Set identifying these specific, updatable fields as "physical":

```
const PHYSICAL_FEATURES_INSTANCE_LIST = new Set([
 "driverId",
 "operationalStatus",
 "approvalStatus"
]);
```

- **Logic**: When an "Update Single" interaction modifies *only* fields present in this `PHYSICAL_FEATURES_INSTANCE_LIST`, the system must "materialize as ManualGeneratedRide". This new physical ride maintains its link to the `rideSeriesId`, and its date is added to the parent `RideSeries.suppressedDates` list to prevent duplicate auto-generation.

Notes on Feature Definitions:

1. **Scheduling State:** Although the document lists "scheduling state" as a conceptual physical feature, it is a *derived state*, not a field that is directly updated. The state (UNSCHEDULED/SCHEDULED) is derived directly from the nullability of the `driverId` field Therefore, the list only needs to check for changes to `driverId`.
2. **Recurrence Pattern:** The document also classifies the "recurrence pattern" as a Virtual Feature. However, it is not included in the constant list above because that pattern exists only on the `RideSeries` object, not the individual `VirtualRide` instance. Changes to the recurrence pattern itself are handled by separate, higher-level operations: **"Update This and Following"** or **"Update All"**

3.1.5. Formal JSON Schemas

The following JSON schemas represent the formal data definitions for the database models described above.

RideSeries Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "RideSeries Schema (Backend Model)",
  "description": "Schema for the RideSeries object, which defines a recurring ride template .",
  "type": "object",
  "properties": {
    "id": { "type": "string", "description": "Primary Key" },
```

```
    "riderId": { "type": "string", "description": "Foreign key to the Rider (Student) " },
    "seriesStatus": { "enum": ”seriesStatus”, "description": "Status of the series" },
    "startLocationId": { "type": “string”, "description": "Foreign Key to the Locations table" },
    "endLocationId": { "type": "string", "description": "Foreign Key to the Locations table" },
    "templateStartTime": { "type": "string", "format": "time", "description": "Series start time"},

    "seriesStartDate": { "type": "string", "format": "date", "description": "The series start date (d_s) " },
    "seriesEndDate": { "type": "string", "format": "date", "description": "The series end date (d_e) " },

    "recurrenceRule": {
      "type": "object",
      "description": "Implementation of the RecurrenceRule interface , based on RFC 5545[cite: 109].",
      "properties": {
        "frequency": { "enum": ["DAILY", "WEEKLY", "MONTHLY", "YEARLY"],"description": “recurrence frequency"},
        "interval": { "type": "integer", "minimum": 1,"default": 1,"description": "recurrence interval"},
        "byday": {"type": "array","items": { "enum": ["SU", "MO", "TU", "WE", "TH", "FR", "SA"] },"des": "Days"},
        "count": { "type": ["integer", "null"], "description": "End condition: number of occurrences "},
        "until": { "type": ["string", "null"], "format": "date-time","description": "End condition: until date "}
      },
      "required": ["frequency"]
    },
    "exceptionDates": {"type": "array","items": { "type": "string", "format": "date" },"des": "Set of dates to skip generation"},
    "suppressedDates": {"type": "array","items": { "type": "string", "format": "date" },"des": "Set of dates to skip auto-generation"},
    "recurrenceGenerationDate": { "type": "string", "format": "date", "description": "The next date by the auto-materializer"}
  },
  "required": ["id", "riderId", "status", "startLocation", "endLocation", "seriesStartDate", "recurrenceRule",”recurrenceGenerationDate”]
}
```

Ride (Physical) Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Ride Schema (PhysicalRide)",
  "description": "Schema for the Ride object, representing a concrete PhysicalRide instance in the database.",
  "type": "object",
  "properties": {
    "id": { "type": "string", "description": "Primary Key" },
    "rideSeriesId": { "type": ["string", "null"], "description": "Link to parent RideSeries. Null if Standalone or Orphaned"},
    "riderId": { "type": "string", "description": "Foreign key to the Rider (Required for Standalone/Orphaned rides)." },
    "driverId": { "type": ["string", "null"], "description": "Assigned driver. Null if UNSCHEDULED."},
    "startLocationId": { "type": "string", "description": "Foreign Key to the Locations table" },
    "endLocationId": { "type": "string", "description": "Foreign Key to the Locations table" },
    "startTime": { "type": "string", "format": "date-time", "description": "Required for TimeState derivation " },
    "endTime": { "type": "string", "format": "date-time", "description": "Estimated end time. Required for ACTIVE/PAST derivation
"},
    "approvalStatus": {
      "enum": ["PENDING", "APPROVED", "APPROVED_WITH_MODIFICATION", "REJECTED"],
      "default": "PENDING",
      "description": "The Admin-controlled approval status"
```

```
    },
    "schedulingState": {
      "enum": ["UNSCHEDULED", "SCHEDULED"],
      "default": "UNSCHEDULED",
      "description": "Driver assignment status [cite: 52-55]"
    },
    "operationalStatus": {
      "enum": ["NOT_STARTED", "ON_THE_WAY", "ARRIVED", "PICKED_UP", "COMPLETED","CANCELLED", "NO_SHOW", "ABORTED"],
      "default": "NOT_STARTED",
      "description": "Real-time ride progression status"
    }
  },
  "required": [
              "id", "riderId", "startLocationId", "endLocationId", "startTime", "endTime", "approvalStatus",
              "schedulingState","operationalStatus"
             ]
}
```