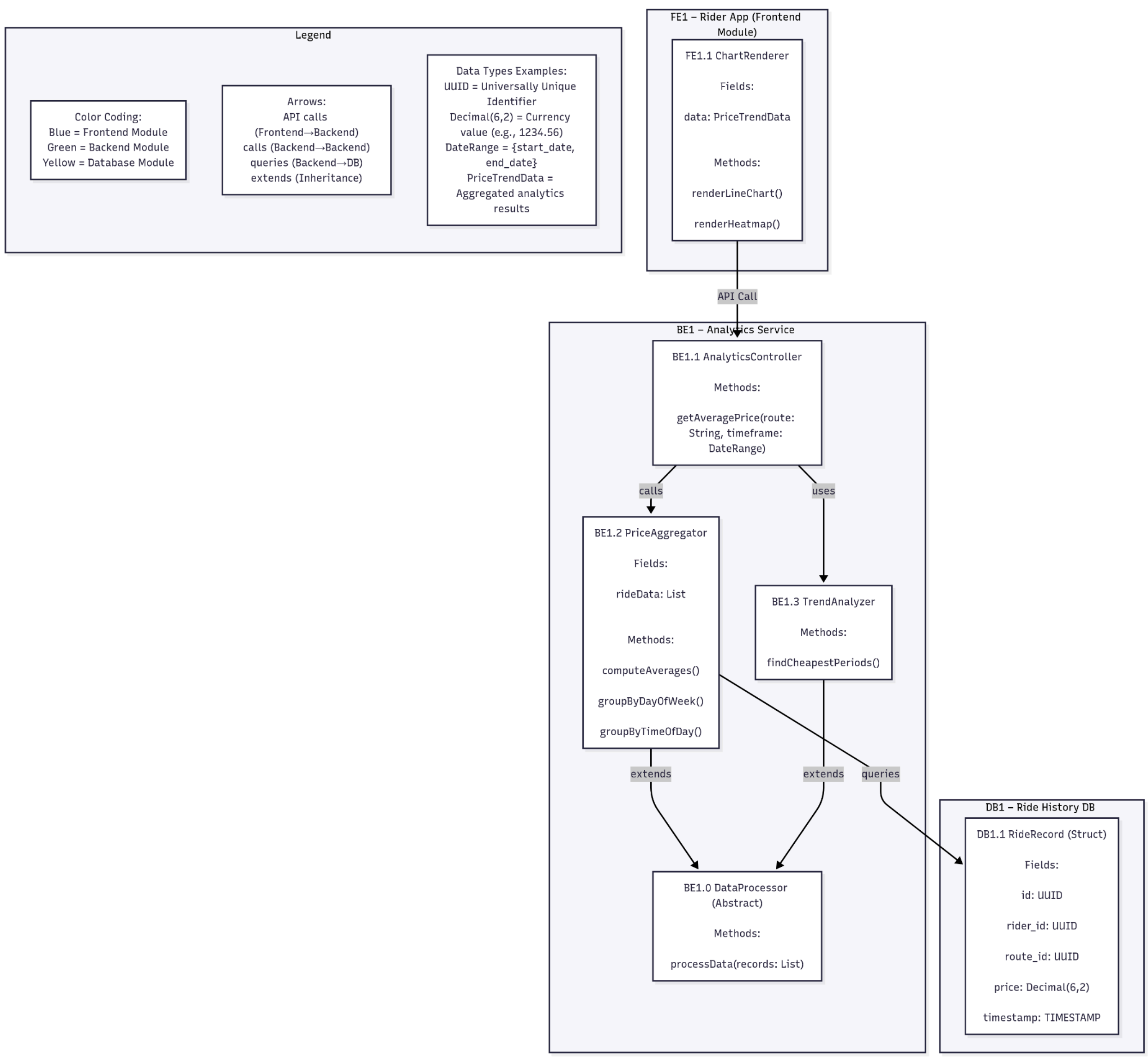
0. User story

As a rider planning my regular trips, I want to see the average price for my route broken down by day of week and time of day for the past month so that I can identify when rides are typically cheaper and plan flexible trips accordingly.

1. Header

Title: Development Specification: Ride Price Trends Feature
Version & Date: v1.0 – Sept 24, 2025
Authors & Roles:
Yena Wu – Product Manager (requirements)

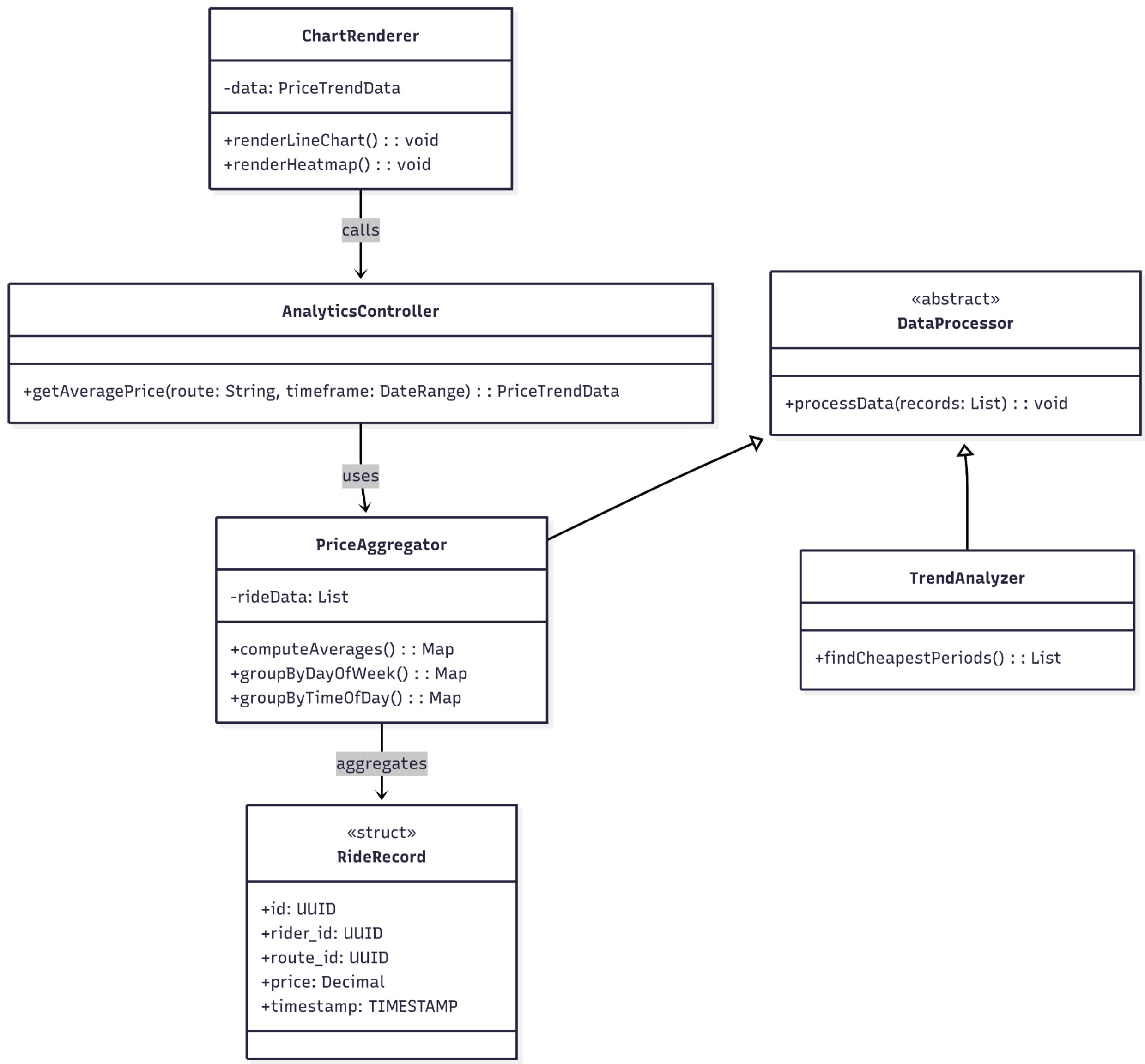
2. Architecture Diagram



The diagram shows the part that lets riders see **average ride prices by day and time**.

- **Frontend (FE1 – Rider App):**
  - The `ChartRenderer` displays charts (line/heatmap) showing when rides are cheaper.
  - It calls the backend to fetch processed data.
- **Backend (BE1 – Analytics Service):**
  - `AnalyticsController` is the entry point for frontend requests.
  - It delegates work to `PriceAggregator`, which fetches ride history and computes averages.
  - `TrendAnalyzer` looks at the averages to highlight the cheapest periods.
  - `DataProcessor` is an abstract parent class that defines the common interface for analytics components.
- **Database (DB1 – Ride History DB):**
  - Stores raw ride records (`RideRecord`) with fields like `rider_id`, `route_id`, `price`, and `timestamp`.
  - Queried by the backend for historical ride data.
- **Connections:**
  - Frontend → Backend via API call.
  - Backend → Database via queries.
  - Backend classes also interact internally (`Controller → Aggregator → Analyzer`).
  - Inheritance is shown where `PriceAggregator` and `TrendAnalyzer` extend `DataProcessor`.
- **Legend:**
  - Explains color coding, arrow types, and uncommon data types (like `Decimal(6,2)` or `DateRange`).

3. Class Diagram



This diagram models the **analytics feature** that computes and displays average ride prices by day and time.

- **ChartRenderer**
  - A frontend class that renders visualizations (`renderLineChart()`, `renderHeatmap()`) based on `PriceTrendData`.
  - Holds a `data` field containing the processed trend results.
  - Calls the backend via `AnalyticsController`.
- **AnalyticsController**
  - The main backend entry point.
  - Provides `getAveragePrice(route, timeframe): PriceTrendData`.
  - Uses `PriceAggregator` to compute averages.

- **PriceAggregator**
  - Responsible for aggregating ride history (`rideData: List<RideRecord>`).
  - Provides methods to compute averages, group by day of week, and group by time of day.
  - Aggregates data from the `RideRecord` struct.
  - Inherits the abstract behavior of `DataProcessor`.
- **TrendAnalyzer**
  - A specialized analytics class that identifies the cheapest time periods (`findCheapestPeriods()`).
  - Extends `DataProcessor`.
- **DataProcessor (Abstract)**
  - Defines the common contract for data processing (`processData(records: List)`).
  - Ensures both `PriceAggregator` and `TrendAnalyzer` follow the same pattern.
- **RideRecord (Struct)**
  - Represents a single completed ride with fields: `id`, `rider_id`, `route_id`, `price`, and `timestamp`.
  - Serves as the raw input data for `PriceAggregator`.

4. List of Classes

Frontend Module – Rider App

1. ChartRenderer (Label: FE1.1)

- **Purpose:** Responsible for rendering visualizations (line charts and heatmaps) that display aggregated ride price trends to the rider.
- **Notes:** Directly interacts with backend API responses and maps them into chart-friendly data structures.
- **Struct / Class:** Class (UI logic).
- **Methods:** Listed in API section, not here.

Backend Module – Analytics Service

2. AnalyticsController (Label: BE1.1)

- **Purpose:** Entry point for handling rider requests related to ride price analytics. Orchestrates calls to `PriceAggregator` and formats the response.

- **Struct / Class:** Controller class.
  Methods: Listed in API section.

3. PriceAggregator (Label: BE1.2)

- **Purpose:** Aggregates ride history data and computes averages by day of week and time of day. Provides data to `AnalyticsController`.
- **Struct / Class:** Data processing class.
- **Notes:** Subclass of abstract `DataProcessor`.

4. TrendAnalyzer (Label: BE1.3)

- **Purpose:** Analyzes aggregated data to highlight the cheapest time periods for a given route. Enhances insights beyond raw averages.
- **Struct / Class:** Data analysis class.
- **Notes:** Subclass of abstract `DataProcessor`.

5. DataProcessor (Label: BE1.0)

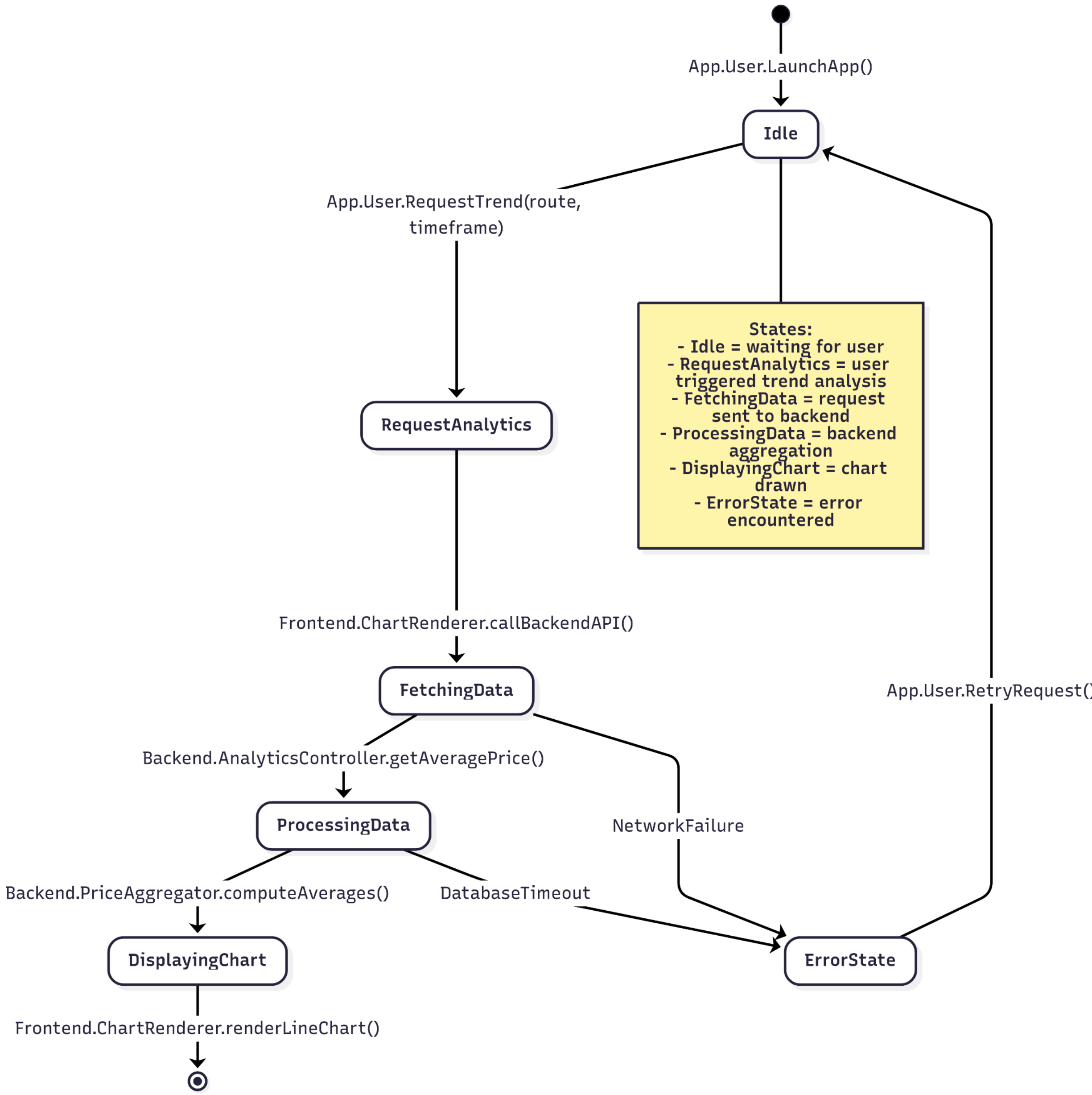
- **Purpose:** Abstract superclass that defines the base interface for data processing classes (`PriceAggregator`, `TrendAnalyzer`).
- **Struct / Class:** Abstract class.
- **Notes:** Contains placeholder `processData()` method to enforce common structure.

Database Module – Ride History DB

6. RideRecord (Label: DB1.1)

- **Purpose:** Data storage struct that represents an individual completed ride transaction.
- **Fields:**
  - `id: UUID`
  - `rider_id: UUID`
  - `route_id: UUID`
  - `price: Decimal(6,2)`
  - `timestamp: TIMESTAMP`
- **Struct / Class:** Struct (data storage only).
- **Notes:** Used at runtime by `PriceAggregator` for historical data queries.

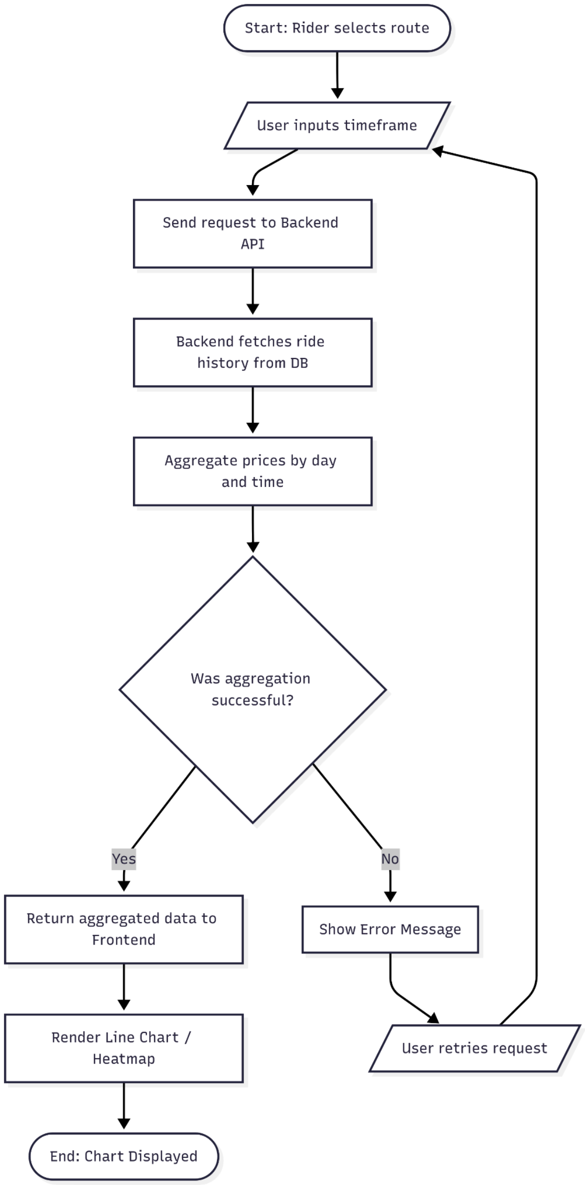
4. State Diagram



5. Flow Chart

The **flow starts** when the rider selects a route and timeframe.
The app sends a request to the backend, which fetches ride history data from the database.
The backend aggregates prices by day of week and time of day.
Decision point:

- If aggregation succeeds → data is returned to the frontend → chart is rendered → **end**.
- If aggregation fails → an error is shown, and the rider can retry → loop back to input.



6. Development Risks and Failures

Backend Module – Analytics Service

Failure BE1: Backend Process Crash

- *User-visible effect:* Rider receives an error message instead of analytics results.
- *Internal effect:* API request terminated mid-execution, no response returned.
- *Recovery Procedure:* Restart backend service automatically via process manager (e.g., Kubernetes pod restart). Retry failed requests.

Failure BE2: Aggregation Timeout

- *User-visible effect:* Slow or no response when requesting price trends.
- *Internal effect:* PriceAggregator stuck on long-running query.
- *Recovery Procedure:* Use query timeouts, return cached pre-aggregated data. Notify user to retry.

Database Module – Ride History DB

Failure DB1: Database Connection Lost

- *User-visible effect:* Rider sees “Unable to fetch data” error.
- *Internal effect:* Backend cannot access RideRecord table.
- *Recovery Procedure:* Backend retries with exponential backoff. Fallback to cached results if available.

Failure DB2: Data Corruption

- *User-visible effect:* Incorrect averages or missing chart data.
- *Internal effect:* Records fail invariant checks (e.g., negative prices).
- *Recovery Procedure:* Roll back to last known good backup. Flag corrupted records for inspection.

Failure DB3: Database Out of Space

- *User-visible effect:* Rider sees error when requesting analytics.
- *Internal effect:* New ride records cannot be stored.
- *Recovery Procedure:* Expand storage allocation. Archive old data to cold storage.

Frontend Module – Rider App

Failure FE1: API Call Failure

- *User-visible effect:* Rider sees error banner in app.
- *Internal effect:* ChartRenderer never receives data payload.
- *Recovery Procedure:* Show error to user with option to retry. Cache last successful chart to display fallback.

Connectivity

Failure CN1: Network Traffic Spike

- *User-visible effect:* Requests are slow or time out.
- *Internal effect:* Overloaded load balancer / API gateway.

- *Recovery Procedure:* Rate limit requests, auto-scale backend instances.

Failure CN2: Lost Internet Connectivity (User-Side)

- *User-visible effect:* Rider cannot load chart.
- *Internal effect:* Request never reaches backend.
- *Recovery Procedure:* App shows offline mode message. Allow user to retry when connection resumes.

Security and Intrusion

Failure SC1: Denial of Service Attack

- *User-visible effect:* Service unavailable for all riders.
- *Internal effect:* Backend resources exhausted by malicious traffic.
- *Recovery Procedure:* Activate WAF (Web Application Firewall) and throttling. Block offending IPs.

Failure SC2: Unauthorized Data Access Attempt

- *User-visible effect:* None (attack is invisible to normal users).
- *Internal effect:* Suspicious queries targeting sensitive RideRecord data.
- *Recovery Procedure:* Detect and block query patterns. Alert security team. Rotate credentials if needed.

Failure Ranking

1. **High Likelihood, High Impact:** Backend process crash, network spikes.
2. **Medium Likelihood, Medium Impact:** Aggregation timeout, DB connection loss.
3. **Low Likelihood, High Impact:** Data corruption, denial of service attack.
4. **Low Likelihood, Low Impact:** User-side connectivity loss.

7. Technologies

Backend

Tech BE1: Node.js (v18.x)

- **Purpose:** Used as the runtime environment for the Analytics Service backend. Handles API requests and business logic.
- **Reason for Choice:** Mature ecosystem, strong async capabilities (good for handling concurrent requests), and widely supported cloud deployment options.
- **Alternatives Considered:** Python (slower for concurrency-heavy workloads), Java (heavier runtime).

- **Source & Docs:** https://nodejs.org

Tech BE2: Express.js (v4.x)

- **Purpose:** Web framework for defining API endpoints (`/analytics/price-trends`).
- **Reason for Choice:** Lightweight, fast to implement, widely adopted with strong middleware ecosystem.
- **Alternatives Considered:** Koa, Fastify (smaller ecosystem).
- **Source & Docs:** https://expressjs.com

Database

Tech DB1: PostgreSQL (v15.x)

- **Purpose:** Primary database to store ride history records (`RideRecord` struct).
- **Reason for Choice:** Reliable relational DB with strong ACID guarantees and advanced indexing.
- **Alternatives Considered:** MySQL (less advanced indexing), MongoDB (not optimized for relational queries).
- **Source & Docs:** https://www.postgresql.org

Tech DB2: TimescaleDB (extension for PostgreSQL, v2.x)

- **Purpose:** Time-series optimization for ride history data, enabling efficient queries by day of week and time of day.
- **Reason for Choice:** Built directly on PostgreSQL, easy integration, designed for time-series analytics.
- **Alternatives Considered:** InfluxDB (separate stack, harder to integrate).
- **Source & Docs:** https://www.timescale.com

Frontend

Tech FE1: React Native (v0.73.x)

- **Purpose:** Cross-platform mobile application framework for rider UI.
- **Reason for Choice:** Shared codebase for iOS and Android, strong ecosystem, fast iteration.
- **Alternatives Considered:** Native Swift/Kotlin (higher development cost), Flutter (smaller developer pool).
- **Source & Docs:** https://reactnative.dev

Tech FE2: Recharts (v2.x)

- **Purpose:** Data visualization library used for rendering line charts and heatmaps.
- **Reason for Choice:** Easy integration with React, good performance, responsive charts.

- **Alternatives Considered:** D3.js (more powerful but higher complexity).
- **Source & Docs:** https://recharts.org

Infrastructure & Tools

Tech INF1: Docker (v24.x)

- **Purpose:** Containerization of backend and database services for consistent deployment.
- **Reason for Choice:** Standard tool for microservices deployment, easy portability.
- **Alternatives Considered:** Podman (less widely adopted).
- **Source & Docs:** https://www.docker.com

Tech INF2: Kubernetes (v1.29.x)

- **Purpose:** Orchestration of backend services and scaling under traffic spikes.
- **Reason for Choice:** Industry standard for scaling microservices.
- **Alternatives Considered:** AWS ECS, Nomad (less flexible).
- **Source & Docs:** https://kubernetes.io

Tech INF3: GitHub Actions (CI/CD)

- **Purpose:** Continuous integration and automated deployment pipeline.
- **Reason for Choice:** Seamless integration with GitHub repos, flexible workflows.
- **Alternatives Considered:** Jenkins (more setup overhead).
- **Source & Docs:** https://github.com/features/actions

8. APIs

Frontend Module – Rider App

Class: ChartRenderer (Label: FE1.1)

- **Public Methods:**
  - `renderLineChart(data: PriceTrendData): void`
  - `renderHeatmap(data: PriceTrendData): void`
- **Private/Internal Methods:**
  - `formatData(rawData: Map): ChartData`

Backend Module – Analytics Service

Class: AnalyticsController (Label: BE1.1)

- **Public Methods:**

- ○ getAveragePrice(route: String, timeframe: DateRange): PriceTrendData
  - ■ *Description:* API entry point for frontend requests. Calls PriceAggregator and TrendAnalyzer.
- **Private/Internal Methods:**
  - ○ validateRequest(route: String, timeframe: DateRange): boolean

Class: PriceAggregator (Label: BE1.2)

- **Public Methods:**
  - ○ computeAverages(records: List<RideRecord>): Map<String, Decimal>
    - ■ *Description:* Computes average prices grouped by keys (day of week, time of day).
  - ○ groupByDayOfWeek(records: List<RideRecord>): Map<String, Decimal>
  - ○ groupByTimeOfDay(records: List<RideRecord>): Map<String, Decimal>
- **Overrides/Overloads:**
  - ○ Inherits processData(records: List<RideRecord>): void from DataProcessor.

Class: TrendAnalyzer (Label: BE1.3)

- **Public Methods:**
  - ○ findCheapestPeriods(data: Map<String, Decimal>): List<String>
    - ■ *Description:* Identifies cheapest time slots for the given route.
- **Overrides/Overloads:**
  - ○ Inherits processData(records: List<RideRecord>): void from DataProcessor.

Class: DataProcessor (Abstract) (Label: BE1.0)

- **Abstract Methods:**
  - ○ processData(records: List<RideRecord>): void

Database Module – Ride History DB

Struct: RideRecord (Label: DB1.1)

- **Fields (Data Only, No Methods):**
  - `id: UUID`
  - `rider_id: UUID`
  - `route_id: UUID`
  - `price: Decimal(6,2)`
  - `timestamp: TIMESTAMP`

9. Public Interfaces

Frontend Module – Rider App

Class: ChartRenderer (Label: FE1.1)

- **Interfaces used within same component:**
  - N/A (UI-only, no internal subcomponents).
- **Interfaces exposed to other components (Backend):**
  - `renderLineChart(data: PriceTrendData): void`
  - `renderHeatmap(data: PriceTrendData): void`
- **Notes:** These methods are **public to the Rider App component**, but **not directly used by Backend**. They depend on Backend APIs returning PriceTrendData.

Backend Module – Analytics Service

Class: AnalyticsController (Label: BE1.1)

- **Interfaces exposed to Frontend (cross-module):**
  - `getAveragePrice(route: String, timeframe: DateRange): PriceTrendData`
    - *Called by Frontend.ChartRenderer to fetch price analytics.*
- **Interfaces used within Backend component:**
  - Calls: `PriceAggregator.computeAverages()`, `TrendAnalyzer.findCheapestPeriods()`.

Class: PriceAggregator (Label: BE1.2)

- **Interfaces exposed within Backend (intra-component use):**

- computeAverages(records: List<RideRecord>): Map<String, Decimal>
- groupByDayOfWeek(records: List<RideRecord>): Map<String, Decimal>
- groupByTimeOfDay(records: List<RideRecord>): Map<String, Decimal>

- **Interfaces exposed cross-component (Database):**
  - Reads data from RideRecord (DB1.1).

Class: TrendAnalyzer (Label: BE1.3)

- **Interfaces exposed within Backend (intra-component use):**
  - findCheapestPeriods(data: Map<String, Decimal>): List<String>

Class: DataProcessor (Label: BE1.0, Abstract)

- **Exposed interface (to subclasses):**
  - processData(records: List<RideRecord>): void

Database Module – Ride History DB

Struct: RideRecord (Label: DB1.1)

- **Public Fields (accessible to Backend):**
  - id: UUID
  - rider_id: UUID
  - route_id: UUID
  - price: Decimal(6,2)
  - timestamp: TIMESTAMP
- **Notes:** Fields are read-only for Analytics Service queries. Mutations (writes) are handled elsewhere (Ride Logging Service, outside scope of this spec).

Cross-Module Usage Summary

- **Frontend → Backend:**
  - ChartRenderer → AnalyticsController.getAveragePrice()
- **Backend → Database:**

  - PriceAggregator → queries RideRecord
- **Intra-Backend (cross-component):**
  - AnalyticsController → PriceAggregator
  - AnalyticsController → TrendAnalyzer

10. Data Schemas

Schema: RideRecord (Label: DS1.1)

- **Runtime Mapping:** Corresponds to RideRecord struct (DB1.1) used in Backend PriceAggregator.
- **Table Name:** ride_records

Columns:

- id: UUID – primary key, unique ride identifier.
- rider_id: UUID – foreign key referencing rider entity.
- route_id: UUID – foreign key referencing route entity.
- price: DECIMAL(6,2) – price of completed ride in USD.
- timestamp: TIMESTAMP – completion time of ride (UTC).

Storage Estimate:

- ~100 bytes per record.
- For 10 million rides per month, ~1 GB/month storage.

Schema: AggregatedPriceTrend (Label: DS1.2)

- **Runtime Mapping:** Corresponds to PriceTrendData (used by ChartRenderer in Frontend).
- **Table Name:** aggregated_price_trends

Columns:

- id: UUID – primary key.
- route_id: UUID – foreign key referencing route entity.
- day_of_week: SMALLINT – numeric day (0=Sunday, 6=Saturday).
- time_of_day_bucket: VARCHAR(20) – e.g., “Morning”, “Afternoon”, “Evening”.
- average_price: DECIMAL(6,2) – computed average for bucket.

- aggregation_period_start: DATE — start date of aggregation window.
- aggregation_period_end: DATE — end date of aggregation window.

Storage Estimate:

- Each record ~120 bytes.
- For 1,000 active routes × 7 days × 4 time buckets = 28,000 rows per month (~3 MB/month).

Schema: Route (Label: DS1.3)

- **Runtime Mapping:** Referenced by both RideRecord and AggregatedPriceTrend.
- **Table Name:** routes

Columns:

- route_id: UUID — primary key.
- origin: VARCHAR(255) — pickup location.
- destination: VARCHAR(255) — drop-off location.

Storage Estimate:

- ~400 bytes per row.
- For 100k routes, ~40 MB total.

11. Risks to Completion

Frontend Module – Rider App

Class: ChartRenderer (FE1.1)

- **Risk Factors:**
  - **Learning/Implementation:** Low difficulty — chart libraries like Recharts are well-documented, but developers must learn best practices for performance on mobile devices.
  - **Verification:** Medium difficulty — ensuring charts render correctly on multiple devices and screen sizes.

    - **Maintenance:** Medium – updates to chart library may introduce breaking changes.

Backend Module – Analytics Service

Class: AnalyticsController (BE1.1)

- **Risk Factors:**
    - **Learning/Implementation:** Low – controller logic is straightforward request handling.
    - **Verification:** Medium – needs extensive API testing to cover all edge cases.
    - **Maintenance:** Low – minimal changes expected once stable.

Class: PriceAggregator (BE1.2)

- **Risk Factors:**
    - **Learning/Implementation:** High – requires complex time-series aggregation queries and handling large datasets efficiently.
    - **Verification:** High – correctness of averages must be validated across varied routes and times.
    - **Maintenance:** Medium – schema changes (e.g., new buckets) will affect query logic.

Class: TrendAnalyzer (BE1.3)

- **Risk Factors:**
    - **Learning/Implementation:** Medium – requires building logic for identifying cheapest periods, potentially expanding to predictive modeling.
    - **Verification:** Medium – correctness depends on input quality.
    - **Maintenance:** Medium – logic may need updates as business rules evolve.

Class: DataProcessor (Abstract, BE1.0)

- **Risk Factors:**
    - **Learning/Implementation:** Low – only defines structure.
    - **Verification:** Low – simple abstraction layer.
    - **Maintenance:** Low – stable by design.

Database Module – Ride History DB

Struct: RideRecord (DB1.1)

- **Risk Factors:**
    - **Learning/Implementation:** Low – schema design is straightforward.
    - **Verification:** Medium – must ensure data integrity across millions of records.
    - **Maintenance:** High – table will grow quickly and may require partitioning or archiving strategies.

Schema: AggregatedPriceTrend (DS1.2)

- **Risk Factors:**
    - **Learning/Implementation:** Medium – requires ETL jobs or scheduled aggregation.
    - **Verification:** High – correctness of aggregated data critical for user trust.
    - **Maintenance:** Medium – schema changes can affect both backend queries and frontend rendering.

Technologies

Node.js / Express (BE1, BE2)

- **Risk Factors:**
    - **Learning/Implementation:** Low – widely used and documented.
    - **Verification:** Medium – async bugs (race conditions) can be subtle.
    - **Maintenance:** Medium – frequent version updates may require dependency upgrades.

PostgreSQL + TimescaleDB (DB1, DB2)

- **Risk Factors:**
    - **Learning/Implementation:** High – requires strong SQL and time-series optimization skills.
    - **Verification:** High – performance tuning and indexing strategies critical.

- **Maintenance:** High – scaling and partitioning large datasets is complex.

React Native + Recharts (FE1, FE2)

- **Risk Factors:**
  - **Learning/Implementation:** Medium – cross-platform compatibility testing required.
  - **Verification:** Medium – device-specific rendering bugs.
  - **Maintenance:** Medium – library updates can break older APIs.

Docker + Kubernetes (INF1, INF2)

- **Risk Factors:**
  - **Learning/Implementation:** Medium – requires DevOps expertise.
  - **Verification:** Medium – CI/CD pipelines must be tested for correctness.
  - **Maintenance:** High – cluster scaling, monitoring, and upgrades are non-trivial.

General Risks

1. **Scaling:** Querying millions of ride records could overload the database if not indexed/partitioned properly.
2. **Data Quality:** Incorrect or incomplete ride data leads to misleading analytics.
3. **Performance:** Aggregation jobs may time out under heavy load.
4. **Security:** API endpoints must be protected from abuse (e.g., bulk scraping).
5. **Maintenance:** Dependencies (React Native, Node.js, Recharts) evolve quickly, creating technical debt.

12. Security and Privacy

1. Personally Identifying Information (PII) Temporarily Stored

- **Types of PII:**
  - rider_id: UUID (internal identifier)
  - origin, destination (locations tied to the rider's trip)
- **Justification:**

    - Required to filter and compute average ride prices per rider's chosen route and timeframe.
- **Data Flow (Temporary Use):**
    - Enters through **Frontend (ChartRenderer)** when rider selects a route.
    - Passes to **Backend (AnalyticsController)** via API request.
    - Processed in **PriceAggregator**, where rider_id is used to fetch ride history records.
    - Returned as **aggregated, anonymized data** (no rider-specific IDs).
- **Disposal:**
    - PII is discarded immediately after aggregation. Only anonymized averages are returned.
- **Protections:**
    - Encrypted in transit (HTTPS/TLS).
    - Stored temporarily in backend memory only (no logs).

2. Personally Identifying Information (PII) in Long-Term Storage

- **Types of PII Stored:**
    1. rider_id: UUID (persistent for ride history records).
    2. origin, destination (stored for analytics).
- **Justification:**
    1. Needed to support analytics queries across time and routes.
- **Storage Details:**
    1. Stored in **PostgreSQL (ride_records table)**.
    2. Indexed by route_id and timestamp for analytics performance.
    3. Data encrypted at rest (AES-256).
- **Data Flow (Long-Term Use):**
    1. Captured at ride completion.
    2. Written into ride_records.
    3. Queried by PriceAggregator for analytics.
    4. Outputs aggregated values, not raw PII.

3. Responsibility for Securing Data

- **Ride History DB (PostgreSQL):**
    - Maintained by **Database Administrators (DBAs)**.
    - Access limited to authorized Backend Engineers.

- **Security Officer:**
    - Responsible for auditing access logs and verifying compliance with policies.

4. Privacy Policy

- **Customer-Visible Policy:**
    - Riders informed that **ride history may be used in anonymized form for analytics**.
    - Policy shown during signup and accessible via app settings.
- **Access Policy:**
    - Only authorized backend services can access rider data.
    - No third-party data sharing without explicit consent.
- **Auditing Procedures:**
    - All access to ride history tables logged with user ID, timestamp, and purpose.
    - Routine audits performed monthly by security officer.

5. Minors' Data

- **Policy on Riders Under 18:**
    - No explicit solicitation of PII from minors.
    - If a minor uses the system, guardian consent is required before storing ride history.
    - Compliance with **COPPA** and **GDPR-K (Children's Privacy)**.

6. Protections Against Attacks

- **Runtime Protections:**
    - No PII stored in logs.
    - In-memory PII cleared after aggregation completes.
- **Network Protections:**
    - All API traffic secured with HTTPS/TLS 1.3.
    - Rate limiting and WAF (Web Application Firewall) to prevent scraping.
- **Database Protections:**
    - Role-based access control (RBAC).
    - Periodic penetration testing and vulnerability scanning.

14. GPT Log History

https://chatgpt.com/share/68d45453-30e4-800d-a7b0-ce4bc7931794