ENPM818T

‘FINAL PROJECT REPORT’

GROUP 4

John Battu
Rishabh Goel
Divya Kamila
Tanvi Kanchan

TABLE OF CONTENTS

1. INTRODUCTION AND BUSINESS SCENARIO

1.1 INTRODUCTION:

In the digital age, convenience is the currency of consumer satisfaction. Among the industries that have undergone a major technological transformation, food delivery services stand out as a clear example of how digital platforms have reshaped traditional businesses. What was once limited to local phone-in orders has evolved into a sophisticated ecosystem connecting customers, restaurants, and delivery partners through mobile apps and online services. ***'ForkLift'***, the database system proposed in this project, is designed to meet the demands of this rapidly growing market by offering a scalable and reliable food delivery platform. However, like any modern food delivery service, ForkLift faces a unique set of technical and operational challenges that demand more than just traditional database solutions.

Fig 1: ForkLift Logo

1.2 PROBLEM ADDRESSED AND BUSINESS REQUIREMENTS:

ForkLift is designed as a modern food delivery platform similar to industry leaders like DoorDash and Uber Eats. It connects customers to a network of local restaurants and delivery couriers, enabling users to browse menus, place orders, track deliveries in real-time, and complete payments—all through a single digital platform. One of the primary challenges in building a platform like ForkLift lies in the complexity of its data management requirements. It extends beyond simple order management. It must handle an array of interconnected components: customer profiles, restaurant partners, live menus with item variations, real-time order processing, courier assignments, location tracking, promotional campaigns, payment processing, and customer feedback. Each element brings its own data structure, update frequency, and performance expectations. Attempting to manage all of this using a single, traditional relational database would quickly lead to scalability bottlenecks, data management headaches, and performance degradation, especially during peak hours when hundreds or even thousands of transactions occur simultaneously.

The business requirements for ForkLift are, therefore, extensive and multi-faceted. The platform must ensure transactional accuracy for critical operations such as order placement, payment processing, and order management. These operations require strict adherence to ACID (Atomicity, Consistency, Isolation, Durability) principles to prevent data corruption or financial discrepancies. At the same time, the platform must also support highly flexible data models for features like dynamic menus, which vary significantly from one restaurant to another and can change frequently throughout the day. Customers expect to see accurate, real-time information about menu items, including modifiers, dietary tags, pricing, and availability. Additionally, the platform must capture and store time-sensitive data like delivery tracking updates, customer reviews, and personalized preferences, all of which contribute to the overall user experience but differ in structure and volume from transactional records.

1.3 NEED FOR MULTI-MODEL APPROACH:

Given these diverse requirements, a single-model database architecture is insufficient for ForkLift. Relational databases excel at managing structured, relationship-heavy data that requires strict consistency, such as customer records, orders, and payments. However, they are less suited for handling highly variable or nested data structures like restaurant menus or delivery tracking logs. Document databases, on the other hand, offer the flexibility to store complex, hierarchical data in a way that is both scalable and efficient for retrieval. They are ideal for use cases where the structure of the data can vary from one document to another, such as menus that include different categories, items, and modifiers depending on the restaurant. Document databases also perform well when dealing with event logs or time-series data, such as courier status updates, where the volume of writes can be high, but relational consistency is not strictly necessary.

This separation of concerns allows each database to specialize in its strengths, ensuring that ForkLift can scale effectively while maintaining data integrity and user experience. Structured, transactional data—such as customer profiles, restaurant details, orders, payments, and promos—is managed in a relational database. This ensures data integrity, supports complex queries and joins, and provides a stable foundation for the platform's core business operations. On the other hand, flexible and high-volume data—such as restaurant menus with nested categories and items, real-time delivery tracking events, customer preferences, and reviews, is stored in a document database like MongoDB. This allows for efficient storage and retrieval of data that does not conform to a rigid schema, while also supporting high write throughput and flexible querying.

The data classification for ForkLift is therefore driven by both business needs and technical realities.

1. **Relational Database:** Entities that are highly structured and require strict consistency, such as customer accounts, orders, and payment records, are managed relationally. These include entities:
    - customer
    - customer_address_detail
    - restaurant
    - restaurant_address_detail
    - address
    - order
    - order_item
    - order_payment
    - promo_usage

- delivery_partner

These entities form the transactional core of the system and are tightly integrated through foreign keys and indexed relationships to ensure fast, reliable operations.

2. **Document Database:** Entities that are more dynamic or require flexible, nested structures are stored as collections. These include:
    - menu
    - delivery_tracking
    - restaurant_hours
    - customer_preferences
    - review
    - notification_events
    - ops_event_log

This separation ensures that each data domain is optimized for its specific access patterns, storage requirements, and performance expectations.

1.4 BUSINESS DRIVERS AND TECHNICAL CONSIDERATIONS:

Several business drivers influenced these design decisions. First and foremost is the need to provide a fast and reliable experience for customers and restaurant partners. Delays or inaccuracies in displaying menu items or processing orders can lead to lost sales and damaged relationships. The platform must also support operational transparency, enabling couriers to be tracked in real-time and customers to receive timely updates on their orders. Furthermore, the system must be scalable, capable of handling sudden spikes in traffic without compromising performance. This includes not only read and write operations but also analytical queries that help the business understand customer behavior, menu popularity, and delivery efficiency.

Technical considerations also played a significant role in shaping the architecture. For the relational database, maintaining referential integrity and supporting ACID-compliant transactions were non-negotiable requirements. This ensures that customer payments, order statuses, and promotional discounts are always accurate and reliable. For the document database, the ability to handle deeply nested and highly variable data structures without constant schema migrations was essential. This allows restaurants to manage their menus independently and flexibly, without requiring platform-wide database changes for every update. Additionally, the document database supports efficient storage and retrieval of time-series data, such as delivery tracking events, which are critical for providing a seamless customer experience.

In conclusion, the multi-model approach adopted for ForkLift is a deliberate and strategic choice, designed to balance the competing demands of data integrity, flexibility, and performance. By leveraging both relational and document-oriented technologies, ForkLift is well-positioned to meet the needs of its customers, restaurant partners, and delivery couriers, while also supporting the operational and analytical capabilities required to grow and scale the business effectively.

2. RELATIONAL DATABASE DESIGN AND IMPLEMENTATION

At the core of ForkLift's architecture lies the relational database, which serves as the system's transactional backbone. It manages the platform's most critical data—customer information, restaurant partnerships, courier profiles, orders, payments, and promotions. To ensure reliability and accuracy in these high-stakes operations, a relational model built on strong data integrity principles was chosen. This section explains the design rationale, normalization process, and technical considerations that guided the implementation of ForkLift's relational database.

2.1 ER DIAGRAM AND DESIGN EXPLANATION:

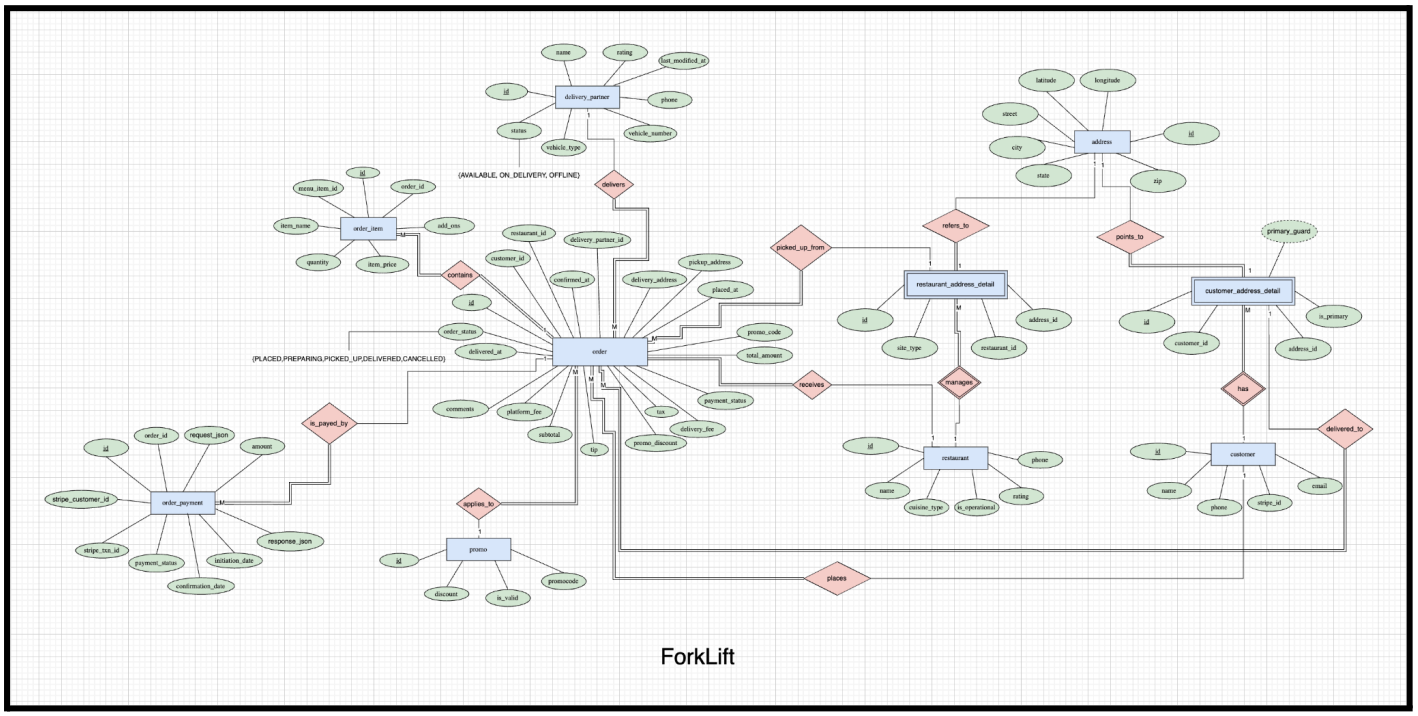

Fig 2: Chen Notation Diagram for ForkLift

The following entities and their attributes were designed to reflect real-world business operations, ensuring data consistency and supporting system scalability.

1. **customer**: The Customer entity stores personal and contact details required for order placement and payment processing.

   Attributes:

   - id (Primary Key): Auto-incremented unique identifier for the customer.
   - name: Full name of the customer.
   - email: Customer's email address, must be unique across the platform.
   - phone: Contact number of the customer.
   - stripe_id: Unique customer identifier used by the payment processor (Stripe).

2. **address:** The Address entity serves as a normalized store for delivery and pickup locations used by both customers and restaurants.

Attributes:

- id (Primary Key): Auto-incremented unique identifier for the address.
- street: Street address.
- city: City name.
- state: Two-character state abbreviation.
- zip: ZIP code or postal code.
- latitude: Optional latitude coordinate for geolocation.
- longitude: Optional longitude coordinate for geolocation.

3. **customer_address_detail**: This linking entity connects customers to their saved addresses while enforcing the rule of one primary address per customer.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier.
   - address_id: Foreign key referencing address.id.
   - customer_id: Foreign key referencing customer.id.
   - is_primary: Boolean flag indicating if this is the customer's primary delivery address.
   - primary_guard: Computed field enforcing that only one address can be marked as primary for each customer.

4. **restaurant**: The Restaurant entity stores restaurant details and operational status.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier for the restaurant.
   - name: Restaurant's name.
   - phone: Contact number for the restaurant.
   - cuisine_type: Type of cuisine served (e.g., Italian, Indian).
   - rating: Average customer rating, stored as a decimal value.
   - is_operational: Boolean flag indicating if the restaurant is actively taking orders.

5. **restaurant_address_detail**: This entity allows restaurants to associate multiple physical locations with their operations.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier.
   - address_id: Foreign key referencing address.id.
   - restaurant_id: Foreign key referencing restaurant.id.
   - site_type: Indicates whether the address is a kitchen, pickup-only location, etc.

6. **delivery_partner**: Delivery Partner captures profiles and availability of couriers who fulfill orders.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier for the courier.
   - name: Delivery partner's full name.
   - phone: Contact number of the delivery partner.
   - vehicle_type: Type of vehicle used (e.g., Car, Bike, Scooter).
   - vehicle_number: Registered vehicle identification.
   - status: Enum indicating courier availability: AVAILABLE, ON_DELIVERY, or OFFLINE.

- last_modified_at: Timestamp for the last status update.
- rating: Average customer rating, stored as a decimal value.

7. **promo**: The Promo entity tracks discount codes available to customers.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier for the promotion.
   - promo_code: Unique promotional code.
   - discount: Decimal representing the discount rate (e.g., 0.10 for 10%).
   - is_valid: Boolean flag indicating if the promotion is currently active.

8. **order**: Order captures the full transaction record for every purchase.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier for the order.
   - customer_id: Foreign key referencing customer.id.
   - restaurant_id: Foreign key referencing restaurant.id.
   - delivery_partner_id: Nullable foreign key referencing delivery_partner.id.
   - order_status: Enum indicating the status of the order (PLACED, PREPARING, PICKED_UP, DELIVERED, CANCELLED).
   - confirmed_at: Timestamp when the order was confirmed.
   - delivery_address: Foreign key referencing customer_address_detail.id.
   - pickup_address: Foreign key referencing restaurant_address_detail.id.
   - placed_at: Timestamp when the order was placed.
   - delivered_at: Nullable timestamp when the order was delivered.
   - subtotal: Subtotal amount before fees and discounts.
   - delivery_fee: Delivery fee charged to the customer.
   - promo_code: Nullable foreign key referencing promo.id.
   - promo_discount: Amount discounted via promotion.
   - tip: Optional tip amount.
   - tax: Tax applied to the order.
   - platform_fee: Platform service fee (default is 1.00).
   - total_amount: Total amount charged.
   - payment_status: Status of payment (e.g., Pending, Captured).
   - comments: Optional special instructions provided by the customer.

9. **order_item**: It captures the details of each individual product within an order.
   Attributes:
   - id (Primary Key): Auto-incremented unique identifier for the line item.
   - order_id: Foreign key referencing order.id.
   - menu_item_id: ID of the menu item, referencing data from the document database.
   - item_name: Captured name of the item at the time of order.
   - quantity: Number of units ordered.
   - item_price: Price per unit at the time of order.
   - add_ons: JSON field storing selected add-ons or special requests.

10. **order_payment**: It ensures accurate tracking of all payment attempts, responses, and final statuses.

    Attributes:

    - id (Primary Key): Auto-incremented unique identifier for the payment record.
    - order_id: Foreign key referencing order.id.
    - stripe_customer_id: Customer reference used by the payment processor (Stripe).
    - stripe_txn_id: Unique transaction ID from the payment processor.
    - request_json: JSON snapshot of the payment request.
    - response_json: JSON snapshot of the payment processor's response.
    - payment_status: Payment status (e.g., Pending, Captured, Failed).
    - amount: Amount processed in the transaction.
    - initiation_date: Timestamp when the payment was initiated.
    - confirmation_date: Nullable timestamp when the payment was confirmed.

Collectively, these entities form a tightly integrated, fully relational model that mirrors the operational reality of a food delivery platform, ensuring that every action taken on the system is traceable, consistent, and reliable.

2.2 NORMALIZATION PROCESS AND 3NF COMPLIANCE:

A critical part of the design process involved ensuring that all tables were normalized to ***Third Normal Form (3NF)***. This level of normalization eliminates data redundancy and ensures data integrity across the database.

The first step in the normalization process was to ensure that each entity represented a single concept with attributes fully dependent on its primary key. For example, the '***customer*** ' table holds only customer-related data, such as name and contact information, without mixing in address or payment details. Address information is stored separately in the '***address***' table and linked to customers through '**customer_address_detail**', following normalization principles by avoiding duplicate address fields within the '***customer***' table.

The second step addressed partial dependencies. In '***order_item***', the combination of order ID and line number uniquely identifies each item in the order. This composite key ensures that all attributes in '***order_item***' depend entirely on this combination, preventing any partial dependency on just the order ID or just the line number.

The final step was to eliminate transitive dependencies. For example, in the '***order***' table, financial calculations such as subtotal, delivery fee, and total amount are stored alongside references to customer, restaurant, and delivery addresses. These attributes are fully dependent on the order's primary key, not on any related foreign keys, ensuring that changes to customer or restaurant records do not unintentionally affect the integrity of past orders.

With these steps carefully applied, the ForkLift relational model achieves full 3NF compliance, positioning the system to handle real-world data complexity without introducing unnecessary duplication or integrity risks.

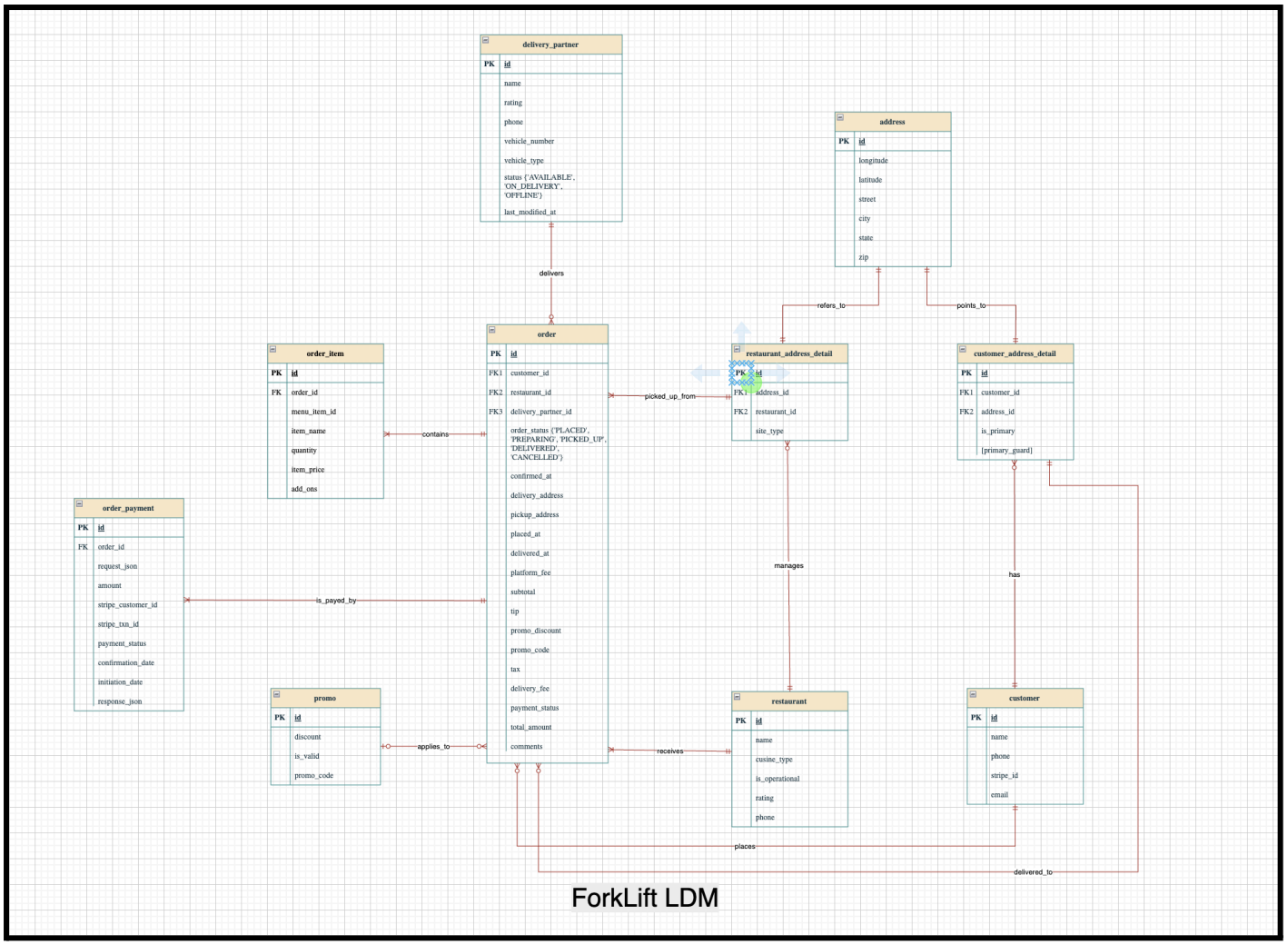

Fig 3: Crow Foot Notation Diagram for ForkLift

2.3 IMPLEMENTATION APPROACH AND CONSTRAINTS:

The database implementation was carried out using MySQL, chosen for its robust support of relational integrity, transaction management, and broad compatibility with operational environments. The implementation script defines all necessary tables with clear primary keys, foreign keys, and constraints to ensure data quality. Several constraints were defined to protect data integrity. Primary keys were applied to ensure the uniqueness of records across all entities. Foreign keys were used extensively to enforce relationships between customers, addresses, orders, restaurants, and delivery partners. This prevents orphaned records and ensures that, for example, an order cannot reference a non-existent customer or restaurant.

Unique constraints were added to critical fields such as customer email addresses and promotional codes to avoid duplication and maintain system integrity. Enumerated values were defined for status fields, such as courier availability and order status, ensuring that only valid states could be stored in the database. Check constraints and default values were also introduced where appropriate, such as setting a default platform fee or marking couriers as OFFLINE by default until they become available. To support system performance, several composite indexes were created, particularly on frequently queried columns such as customer ID and order placement date. These indices enable efficient retrieval of customer order history, restaurant performance reports, and operational dashboards.

The complete implementation details, including all DDL scripts and sample data insertions, have been provided separately as part of the project deliverables. These scripts create the database schema and populate it with realistic test data to demonstrate system functionality. Actual query demonstrations and data samples are also included in the supporting files.

3. DOCUMENT DATABASE DESIGN AND IMPLEMENTATION

While ForkLift's relational database forms the backbone of its transactional operations, it is the document-oriented database that enables the platform to remain flexible and responsive to the fast-changing needs of restaurants, delivery partners, and customers. ForkLift leverages MongoDB as the document database component, taking advantage of its schema-flexible structure and ability to handle nested, complex data. This section describes the design of the document collections, the indexing strategy that supports efficient querying, and the overall role of this component in ForkLift's architecture.

3.1 DOCUMENT SCHEMAS:

ForkLift's document database consists of seven purpose-built collections, each designed to capture a different aspect of the platform's dynamic content. These collections are defined with ***JSON schema validators*** to ensure data quality while preserving flexibility.

1. The ***menu collection*** stores complete menus for each restaurant in a single document. Each document includes a reference to the associated restaurant's relational ID, a timestamp indicating the last synchronization with the restaurant's system, and an array of categories. These categories contain nested arrays of menu items, each with details such as a unique menu item ID, name, base price, optional modifiers, dietary tags, and nutritional information. This structure allows each restaurant to customize its menu without impacting others, supporting real-time updates with no need for schema migrations. Sample menus for ten restaurants have been loaded into the database.

2. The ***delivery_tracking collection*** manages the lifecycle of each order's delivery. Each document includes the order's unique ID and an array of status events, such as "ACCEPTED," "PREPARING," "PICKED_UP," "AT_DOOR," "DELIVERED," and "CANCELLED." Each event is timestamped, and pickup or delivery events optionally capture geolocation coordinates using the GeoJSON format. This design allows ForkLift to provide customers with live courier tracking and order progress updates. The dataset includes tracking events for ten sample orders.

3. The ***restaurant_hours collection*** captures weekly operating schedules and holiday exceptions for each restaurant. Each document defines standard hours for all seven days of the week and may include override entries for special dates. This enables ForkLift to accurately display restaurant availability to customers in real time. Operational schedules for ten restaurant entities have been populated as part of the initial data load.

4. The ***customer_preferences collection*** personalizes the user experience by storing customer-specific preferences. These include lists of saved addresses, favorite cuisines, dietary restrictions, and device tokens for push notifications.

5. Customer feedback is stored in the ***review collection***, where each document records a rating, textual review, optional photo URLs, and the timestamp of submission. Reviews are tied to both customer and restaurant IDs. Example reviews have been loaded to simulate customer feedback across various orders.

6. The ***notification_events collection*** logs outbound messages sent to customers, including the communication channel used (such as SMS, email, or push notifications), message templates, payload content, delivery status, number of delivery attempts, and timestamps. This helps ForkLift

track the effectiveness of its customer engagement efforts. Ten sample notification events have been inserted, covering different channels and templates.

7. The ***ops_event_log collection*** captures operational system events such as service errors or system health checks. These events include details like component name, severity level, message content, and optional diagnostic metadata. One example event simulating a payment gateway timeout has been added to demonstrate operational logging capabilities.

ForkLift has been loaded with realistic sample data across all collections, covering a wide range of scenarios including multi-cuisine menus, live delivery tracking, customer feedback, and operational notifications.

In addition to the data, a comprehensive set of demonstration queries has been prepared to showcase the capabilities of the document model. These include:

- Listing all menus or filtering them by dietary tags like "vegan".
- Retrieving customer-specific reviews, sorted by timestamp or filtered by keyword mentions such as "delivery".
- Aggregating average ratings per restaurant and identifying the most active customers based on review counts.
- Updating or flagging reviews for administrative purposes.
- Managing push notification tokens or counting delivery tracking records to monitor platform usage.
- Running pagination on large datasets like reviews to simulate customer-facing browsing experiences.

3.2 INDEXING STRATEGY:

Indexes are defined to optimize common query patterns. To ensure responsive performance across all collections, ForkLift has implemented a strategic set of indexes.

1. For the ***menu collection***, an index on ***restaurant_id*** allows for quick retrieval of restaurant-specific menus. An additional multikey index on ***categories.items.tags*** support efficient filtering of menu items based on dietary tags like "vegan" or "gluten-free."
2. In the ***delivery_tracking collection***, a unique index on ***order_id*** ensures that each order's tracking record can be quickly located. A geospatial index on ***events.location*** enables geospatial queries, such as identifying couriers near a customer's address in real time.
3. The ***restaurant_hours collection*** uses an index on ***restaurant_id*** for fast schedule lookups. Similarly, the ***customer_preferences collection*** is indexed by ***customer_id*** to facilitate quick retrieval of personalized settings.
4. Indexes on ***review***, ***notification_events***, and ***ops_event_log*** collections are designed to support their operational use cases. For example, reviews are queried by ***restaurant_id*** and ***customer_id***, notification events are accessed by ***order_id*** and ***customer_id***, and operational logs are sorted by ***timestamp*** for monitoring purposes.

These indexing strategies are critical for maintaining performance as the dataset grows, ensuring that queries remain fast and responsive under load.

4. ANALYSIS AND CONCLUSION

Looking back at the design process, one of the biggest takeaways for our team is how important it is to *match the right data model to the right problem*. This section reflects on the performance characteristics of the relational and document models, discusses trade-offs made during the design process, considers future scalability needs, and summarizes the lessons learned during development.

4.1 PERFORMANCE CONSIDERATIONS:

Each component of the system's architecture was selected with a clear understanding of how performance would be impacted by both the nature of the data and the expected access patterns. The relational database, built on MySQL, delivers strong ACID guarantees, which are essential for preserving consistency across orders, payments, and other critical transactions. Indexing on composite keys like (customer_id, placed_at) ensures that queries such as customer order histories or restaurant-level sales trends can execute quickly, even under high load. Its strengths include ACID compliance and support for complex joins and aggregations. However, it is less suited for managing highly variable or deeply nested data structures, and schema migrations can be costly.

The document database, powered by MongoDB, serves a different performance profile. It is optimized for handling deeply nested, flexible data like restaurant menus and delivery status logs. Documents are designed to be self-contained, allowing reads and writes to occur in a single round trip. For example, retrieving a full restaurant menu or appending a new delivery status update involves minimal overhead. To support high-performance querying, particularly in scenarios like filtering menu items by dietary tags or rendering delivery tracking on a map, strategic indexes were implemented, including multikey and geospatial indexes. In practice, these allow ForkLift to deliver a highly responsive user experience, even when data structures are complex and frequently updated.

Both database models present unique performance considerations. The decision to separate workloads based on their data model ensures that neither database is burdened with tasks it isn't optimized for. Transactional operations remain efficient in the relational tier, while flexible, high-throughput tasks are delegated to the document tier.

4.2 DESIGN TRADE-OFFS AND RATIONALE:

Designing ForkLift's architecture involved a careful balancing of competing priorities—namely, consistency, flexibility, and speed. The primary trade-off in this multi-model design is the separation of transactional and content data across two systems. This approach maximizes the strengths of each database model but introduces complexity in maintaining data synchronization and consistency across models.

On the relational side, one limitation is schema rigidity. While normalization and enforced constraints ensure data integrity, they also introduce friction when business rules change or when new data types must be introduced. A small structural change often requires a coordinated schema migration. To mitigate this, the system was intentionally designed to keep only highly structured and stable entities in the relational layer.

Conversely, the document layer sacrifices some enforcement of cross-collection integrity for the sake of speed and adaptability. For instance, while restaurant menus are stored in a nested, flexible format, there is

no built-in enforcement that ensures a menu item's structure remains consistent across all restaurants. This flexibility is necessary but introduces a dependency on application logic and validator schemas to maintain order. The absence of native joins also means that analytics queries across both data models must be executed in the application layer or with a dedicated analytics pipeline.

Nevertheless, the benefits of scalability, performance, and flexibility outweigh these challenges, making the multi-model architecture a suitable choice for ForkLift's operational needs.

4.3 FUTURE SCALABILITY CONSIDERATIONS:

Scalability was not an afterthought in ForkLift's design. On the relational side, vertical scaling can be achieved through upgraded hardware or optimized query plans, while horizontal scaling can be supported through read replicas and partitioning (e.g., sharding orders by date or restaurant). For example, given the volume of orders that can occur during meal rushes, the system's order table is indexed with time-aware composite keys to distribute the load efficiently.

The document database is inherently better suited to horizontal scaling. Collections like menu and delivery_tracking can be sharded by restaurant ID or order ID, enabling concurrent access and writes across nodes. MongoDB's sharding and replica set capabilities mean that as ForkLift's restaurant count or order volume grows, the system can scale elastically to meet demand without degrading performance.

Furthermore, the data model supports future extensions, such as customer loyalty programs, menu item A/B testing, or machine-learning-powered personalization, without requiring disruptive changes to the schema. By keeping the boundaries between models clean, the platform can evolve modularly over time.

4.4 SUMMARY:

Building the ForkLift database system reinforced one of the most important truths in system design: no single database model fits all use cases. Trying to force structured and unstructured data into one system, whether relational or document-based, would have compromised both performance and maintainability. Instead, embracing a multi-model approach allowed us to specialize each layer of the architecture. Our relational database delivered consistency and integrity for orders, payments, and customers, while MongoDB gave us the flexibility to manage evolving menus, preferences, and real-time tracking events.

This separation of concerns led to better modularity in our application logic and more efficient development overall. Working with two models introduced some operational overhead, particularly around data synchronization. But we believe the performance and scalability gains made this trade-off worthwhile.