BigQuery Schema Design Mastery: From Fundamentals to Enterprise Optimization

Data Engineering Interview Preparation

August 17, 2025

Contents

1 The Evolution of Data Warehouse Schema Design **3**
1.1 Traditional Relational Database Limitations 3
1.2 The BigQuery Revolution 3

2 Real-World Case Study: GO-JEK's Data Architecture **3**
2.1 Business Context and Scale 3
2.2 Data Structure Complexity 4

3 Schema Design Approaches: Comprehensive Analysis **4**
3.1 Approach 1: Normalized (Relational) Schema 4
3.2 Approach 2: Fully Denormalized Schema 5
3.3 Approach 3: Nested and Repeated Fields (BigQuery Optimal) 6

4 BigQuery Data Types Deep Dive **7**
4.1 STRUCT Data Type: Nested Fields 7
4.2 ARRAY Data Type: Repeated Fields 8

5 Schema Design Decision Framework **9**
5.1 When to Use Nested and Repeated Fields 9
5.2 When to Keep Normalized Schemas 10

6 Performance Optimization Strategies **10**
6.1 Partitioning: Foundation of Performance 10
6.2 Clustering: Advanced Performance Tuning 11
6.3 Query Optimization Techniques 13

7 Real-World Implementation Examples **13**
7.1 E-commerce Platform Schema 13
7.2 Analytics Platform Schema 15

8 Interview Questions and Discussion Points **16**
8.1 Core Concept Questions 16
8.2 Performance Optimization Questions 17
8.3 Real-World Scenario Questions 18

9 Advanced Topics and Best Practices **18**
9.1 Schema Design Patterns 18
9.2 Performance Monitoring and Optimization 19

10 Key Takeaways and Best Practices **19**
10.1 Schema Design Principles 19
10.2 Performance Optimization 19
10.3 Cost Management 20
10.4 Implementation Guidelines 20

Executive Summary

This comprehensive guide covers BigQuery schema design from fundamental concepts to enterprise-level optimization strategies. We explore the paradigm shift from traditional relational databases to BigQuery's nested and repeated fields, using real-world examples like GO-JEK's 13+ petabyte monthly processing to illustrate practical applications.

1 The Evolution of Data Warehouse Schema Design

1.1 Traditional Relational Database Limitations

Historical Context: Traditional relational databases were designed for transactional processing with normalized schemas that prioritize data integrity and storage efficiency over analytical query performance.

Key Limitations in Analytics Context:

- **JOIN Performance**: Most intensive computational workloads in analytical queries
- **Record-based Processing**: Must open entire records to extract JOIN keys
- **Scalability Issues**: 10+ table JOINs become prohibitively expensive
- **Query Complexity**: Need to know all related tables in advance
- **Storage Model Mismatch**: Row-based storage inefficient for column-based analytics

1.2 The BigQuery Revolution

Paradigm Shift: BigQuery introduced columnar storage with support for nested and repeated fields, fundamentally changing how we approach data warehouse schema design.

Key Innovations:

- **Columnar Storage**: Only reads required columns, ignoring unused data
- **Nested Fields (STRUCT)**: Pre-joined data within single tables
- **Repeated Fields (ARRAY)**: Multiple values within single rows
- **Serverless Architecture**: Automatic scaling and optimization
- **Pay-per-query Model**: Cost optimization through efficient processing

2 Real-World Case Study: GO-JEK's Data Architecture

2.1 Business Context and Scale

Company Profile:

- **Service**: Indonesia's leading ride-booking and multi-service platform

- **Data Volume**: 13+ petabytes processed monthly
- **Use Case**: Real-time business decision support
- **Challenge**: Efficient storage and querying of complex ride-booking data

2.2 Data Structure Complexity

Core Data Entities:

- **Orders**: Single pickup and drop-off locations per ride
- **Events**: Multiple events per order (ordered, confirmed, en route, completed)
- **Locations**: Geographic coordinates and addresses
- **User Data**: Customer and driver information
- **Business Metrics**: Revenue, ratings, performance indicators

Schema Design Challenge: How to efficiently store ride-booking data where each order has multiple events, locations, and associated metadata while supporting high-volume analytical queries?

3 Schema Design Approaches: Comprehensive Analysis

3.1 Approach 1: Normalized (Relational) Schema

Design Philosophy: Eliminate data redundancy through proper normalization
Implementation Strategy:

```sql
-- Normalized approach
CREATE TABLE orders (
    order_id INT64 PRIMARY KEY,
    customer_id INT64,
    driver_id INT64,
    pickup_lat FLOAT64,
    pickup_lng FLOAT64,
    dropoff_lat FLOAT64,
    dropoff_lng FLOAT64,
    order_time TIMESTAMP,
    total_amount FLOAT64
);

CREATE TABLE order_events (
    event_id INT64 PRIMARY KEY,
    order_id INT64,
    event_type STRING,
    event_time TIMESTAMP,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

```sql
CREATE TABLE customers (
    customer_id INT64 PRIMARY KEY,
    name STRING,
    phone STRING,
    email STRING
);
```

Performance Characteristics:

- **Storage Efficiency**: Minimal data redundancy
- **Data Integrity**: Strong referential constraints
- **Query Performance**: Expensive JOINs required
- **Scalability**: Poor performance with large datasets
- **Complexity**: Need to understand all table relationships

Analytical Query Example:

```sql
-- Complex query requiring multiple JOINs
SELECT
    o.order_id,
    c.name as customer_name,
    COUNT(e.event_id) as event_count,
    AVG(o.total_amount) as avg_amount
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_events e ON o.order_id = e.order_id
WHERE o.order_time >= '2024-01-01'
GROUP BY o.order_id, c.name;
```

3.2 Approach 2: Fully Denormalized Schema

Design Philosophy: Pre-join all data for maximum query performance

Implementation Strategy:

```sql
-- Denormalized approach
CREATE TABLE ride_data (
    order_id INT64,
    customer_id INT64,
    customer_name STRING,
    customer_phone STRING,
    driver_id INT64,
    driver_name STRING,
    pickup_lat FLOAT64,
    pickup_lng FLOAT64,
    dropoff_lat FLOAT64,
    dropoff_lng FLOAT64,
    order_time TIMESTAMP,
    event_type STRING,
    event_time TIMESTAMP,
```

```
    total_amount FLOAT64
);
```

Performance Characteristics:

- **Query Performance**: Fast single-table queries
- **Storage Overhead**: Significant data duplication
- **Aggregation Issues**: Risk of double-counting
- **Data Consistency**: Difficult to maintain
- **Update Complexity**: Changes require multiple row updates

Data Explosion Problem:

Order ID	Events	Rows Created
12345	4 events	4 rows (duplicate order data)
12346	3 events	3 rows (duplicate order data)
12347	5 events	5 rows (duplicate order data)

3.3 Approach 3: Nested and Repeated Fields (BigQuery Optimal)

Design Philosophy: Best of both worlds - logical organization with performance benefits

Implementation Strategy:

```
-- BigQuery nested and repeated approach
CREATE TABLE orders (
    order_id INT64,
    customer STRUCT<
        id INT64,
        name STRING,
        phone STRING,
        email STRING
    >,
    driver STRUCT<
        id INT64,
        name STRING,
        rating FLOAT64
    >,
    pickup_location STRUCT<
        latitude FLOAT64,
        longitude FLOAT64,
        address STRING
    >,
    destination STRUCT<
        latitude FLOAT64,
        longitude FLOAT64,
        address STRING
    >,
```

```
    events ARRAY<STRUCT<
        event_type STRING,
        timestamp TIMESTAMP,
        metadata STRUCT<
            location_lat FLOAT64,
            location_lng FLOAT64,
            status_code STRING
        >
    >>,
    order_time TIMESTAMP,
    total_amount FLOAT64,
    payment_method STRING
)
PARTITION BY DATE(order_time)
CLUSTER BY customer.id, driver.id;
```

Performance Characteristics:

- **Logical Organization**: One row per order
- **Query Performance**: No JOINs required
- **Storage Efficiency**: No data explosion
- **Data Integrity**: Maintains relationships
- **Analytical Power**: Supports complex aggregations

4 BigQuery Data Types Deep Dive

4.1 STRUCT Data Type: Nested Fields

Definition: Structured data type that allows grouping related fields within a single column

Key Characteristics:

- **Schema Type**: RECORD in BigQuery schema
- **Field Access**: Dot notation (e.g., customer.name)
- **Benefits**: Pre-joined data, logical organization
- **Use Cases**: Related data that should be co-located

Implementation Examples:

```
-- Simple STRUCT
customer STRUCT<
    id INT64,
    name STRING,
    email STRING
>

-- Nested STRUCT
```

```
address STRUCT<
    street STRUCT<
        number INT64,
        name STRING
    >,
    city STRING,
    state STRING,
    zip_code STRING
>

-- STRUCT with arrays
contact_info STRUCT<
    phones ARRAY<STRING>,
    emails ARRAY<STRING>,
    primary_contact STRUCT<
        name STRING,
        phone STRING
    >
>
```

Query Examples:

```
-- Accessing STRUCT fields
SELECT
    order_id,
    customer.name,
    customer.email,
    pickup_location.latitude,
    pickup_location.longitude
FROM orders
WHERE customer.name LIKE '%John%';

-- Filtering on STRUCT fields
SELECT order_id, total_amount
FROM orders
WHERE pickup_location.latitude BETWEEN 40.0 AND 41.0
  AND pickup_location.longitude BETWEEN -74.0 AND -73.0;
```

4.2 ARRAY Data Type: Repeated Fields

Definition: Repeated values within a single field, allowing multiple values per row

Key Characteristics:

- **Mode**: REPEATED in BigQuery schema
- **Benefits**: Handles one-to-many relationships
- **Use Cases**: Events, timestamps, related items
- **Performance**: No JOIN operations needed

Implementation Examples:

```sql
-- Simple ARRAY
tags ARRAY<STRING>

-- ARRAY of STRUCTs
events ARRAY<STRUCT<
    event_type STRING,
    timestamp TIMESTAMP,
    user_id INT64
>>

-- Nested ARRAYs
order_items ARRAY<STRUCT<
    product_id INT64,
    quantity INT64,
    price FLOAT64,
    options ARRAY<STRING>
>>
```

Query Examples:

```sql
-- Working with ARRAYs
SELECT
    order_id,
    ARRAY_LENGTH(events) as event_count,
    events[OFFSET(0)].event_type as first_event
FROM orders;

-- Unnesting ARRAYs
SELECT
    order_id,
    event.event_type,
    event.timestamp
FROM orders,
UNNEST(events) as event
WHERE event.timestamp >= '2024-01-01';

-- Aggregating ARRAY data
SELECT
    order_id,
    COUNT(*) as event_count,
    ARRAY_AGG(event.event_type) as all_events
FROM orders,
UNNEST(events) as event
GROUP BY order_id;
```

5 Schema Design Decision Framework

5.1 When to Use Nested and Repeated Fields

Primary Indicators:

1. **One-to-Many Relationships**: Data naturally has parent-child structure

2. **Query Patterns**: Frequently query related data together
3. **Data Volume**: Large datasets where JOINs become expensive
4. **Analytical Workloads**: Complex aggregations and reporting
5. **Real-time Requirements**: Need for fast query response times

Specific Use Cases:

- **E-commerce**: Orders with multiple items and events
- **Analytics**: User sessions with multiple events
- **IoT Data**: Device readings with multiple sensors
- **Financial Data**: Transactions with multiple legs
- **Healthcare**: Patient records with multiple visits

5.2 When to Keep Normalized Schemas

Primary Indicators:

1. **Small Dimension Tables**: ¡ 10GB in size
2. **Frequent Updates**: High UPDATE/DELETE operations
3. **Complex Business Logic**: Multiple validation rules
4. **Legacy Integration**: Need to maintain compatibility
5. **Data Consistency**: Critical for business operations

Decision Matrix:

Factor	**Use Nested/Repeated**	**Keep Normalized**
Table Size	¿ 10GB	¡ 10GB
Update Frequency	Low	High
Query Complexity	Complex aggregations	Simple lookups
Data Relationships	One-to-many	Many-to-many
Performance Requirements	Critical	Moderate
Storage Cost Sensitivity	Low	High

6 Performance Optimization Strategies

6.1 Partitioning: Foundation of Performance

Concept: Dividing tables into smaller, manageable pieces based on column values
Benefits:

- **Cost Reduction**: Read only relevant partitions
- **Performance Improvement**: Faster query execution

- **Accurate Cost Estimation**: Better query planning
- **Automatic Management**: BigQuery handles partition creation

Partitioning Types:

1. **Date/Time Partitioning**: Most common and effective
2. **Integer Range Partitioning**: For ID-based partitioning
3. **Ingestion Time Partitioning**: Based on load time

Implementation Examples:

```sql
-- Date partitioning
CREATE TABLE events (
    event_date DATE,
    user_id INT64,
    event_type STRING
)
PARTITION BY DATE(event_date);

-- Integer range partitioning
CREATE TABLE user_data (
    user_id INT64,
    data STRING
)
PARTITION BY RANGE_BUCKET(user_id, GENERATE_ARRAY(0, 1000000, 10000));

-- Partition with expiration
CREATE TABLE logs (
    log_date DATE,
    log_data STRING
)
PARTITION BY DATE(log_date)
OPTIONS(partition_expiration_days=90);
```

Best Practices:

- **Always filter on partition columns** in WHERE clauses
- **Use date partitioning** for time-series data
- **Set appropriate expiration** for old partitions
- **Monitor partition count** (metadata overhead)
- **Isolate partition field** on left side of comparisons

6.2 Clustering: Advanced Performance Tuning

Concept: Organizing data within partitions based on column values for optimal access patterns

Benefits:

- **Filter Performance**: Eliminates unnecessary data scans
- **Aggregation Performance**: Co-locates similar values
- **Cost Reduction**: Reduces bytes processed
- **Automatic Re-clustering**: BigQuery maintains optimal sorting

Implementation Examples:

```sql
-- Single column clustering
CREATE TABLE user_events (
    event_date DATE,
    user_id INT64,
    event_type STRING
)
PARTITION BY DATE(event_date)
CLUSTER BY user_id;

-- Multi-column clustering
CREATE TABLE orders (
    order_date DATE,
    customer_id INT64,
    product_category STRING,
    order_amount FLOAT64
)
PARTITION BY DATE(order_date)
CLUSTER BY customer_id, product_category;

-- Clustering with nested fields
CREATE TABLE ride_data (
    ride_date DATE,
    customer STRUCT<id INT64, tier STRING>,
    driver STRUCT<id INT64, rating FLOAT64>,
    events ARRAY<STRUCT<type STRING, time TIMESTAMP>>
)
PARTITION BY DATE(ride_date)
CLUSTER BY customer.id, driver.id;
```

Clustering Best Practices:

- **Column Order Matters**: Most selective columns first
- **Combine with Partitioning**: Maximum performance benefit
- **Choose High-Cardinality Columns**: Better clustering effectiveness
- **Limit to 4 Columns**: Diminishing returns beyond this
- **Monitor Effectiveness**: Clustering weakens over time

6.3 Query Optimization Techniques

Schema-Level Optimization:

1. **Use nested/repeated fields** instead of JOINs when possible
2. **Keep dimension tables ¡ 10GB** normalized
3. **Denormalize large tables** (¿ 10GB) for better performance
4. **Consider query patterns** when designing schema

Query-Level Optimization:

1. **Always filter on partition columns**
2. **Use clustering columns** in WHERE clauses
3. **Limit columns selected** to only what's needed
4. **Use appropriate data types** for better compression
5. **Avoid SELECT *** in large tables

Cost Optimization:

1. **Partition tables** for accurate cost estimation
2. **Use clustering** to reduce bytes processed
3. **Set partition expiration** for old data
4. **Monitor query costs** and optimize accordingly
5. **Use materialized views** for frequently accessed data

7 Real-World Implementation Examples

7.1 E-commerce Platform Schema

Business Requirements:

- **Scale**: Millions of orders daily
- **Data Types**: Orders, items, customers, payments
- **Queries**: Revenue analysis, customer behavior, inventory
- **Performance**: Sub-second response for dashboards

Optimized Schema Design:

```sql
CREATE TABLE orders (
    order_id INT64,
    customer STRUCT<
        id INT64,
        name STRING,
        email STRING,
        tier STRING,
        registration_date DATE
    >,
    order_info STRUCT<
        order_date DATE,
        status STRING,
        total_amount FLOAT64,
        tax_amount FLOAT64,
        shipping_amount FLOAT64
    >,
    items ARRAY<STRUCT<
        product_id INT64,
        product_name STRING,
        category STRING,
        quantity INT64,
        unit_price FLOAT64,
        total_price FLOAT64,
        options ARRAY<STRING>
    >>,
    payment STRUCT<
        method STRING,
        transaction_id STRING,
        status STRING,
        processed_at TIMESTAMP
    >,
    shipping STRUCT<
        address STRUCT<
            street STRING,
            city STRING,
            state STRING,
            zip_code STRING,
            country STRING
        >,
        method STRING,
        tracking_number STRING,
        estimated_delivery DATE
    >,
    events ARRAY<STRUCT<
        event_type STRING,
        timestamp TIMESTAMP,
        user_id INT64,
        metadata STRUCT<
            ip_address STRING,
            user_agent STRING,
            referrer STRING
```

```sql
        >
    >>
)
PARTITION BY DATE(order_info.order_date)
CLUSTER BY customer.id, customer.tier;
```

Query Examples:

```sql
-- Revenue analysis by customer tier
SELECT
    customer.tier,
    COUNT(DISTINCT order_id) as order_count,
    SUM(order_info.total_amount) as total_revenue,
    AVG(order_info.total_amount) as avg_order_value
FROM orders
WHERE order_info.order_date >= '2024-01-01'
GROUP BY customer.tier;

-- Product category analysis
SELECT
    item.category,
    COUNT(*) as item_count,
    SUM(item.total_price) as category_revenue
FROM orders,
UNNEST(items) as item
WHERE order_info.order_date >= '2024-01-01'
GROUP BY item.category
ORDER BY category_revenue DESC;

-- Customer journey analysis
SELECT
    customer.id,
    customer.name,
    ARRAY_LENGTH(events) as interaction_count,
    ARRAY_AGG(event.event_type) as event_sequence
FROM orders,
UNNEST(events) as event
WHERE order_info.order_date >= '2024-01-01'
GROUP BY customer.id, customer.name;
```

7.2 Analytics Platform Schema

Business Requirements:

- **Scale**: Billions of events daily
- **Data Types**: User events, sessions, page views
- **Queries**: User behavior analysis, funnel analysis
- **Performance**: Real-time dashboards

Optimized Schema Design:

```sql
CREATE TABLE user_sessions (
    session_id STRING,
    user STRUCT<
        id INT64,
        email STRING,
        signup_date DATE,
        country STRING,
        device_type STRING
    >,
    session_info STRUCT<
        start_time TIMESTAMP,
        end_time TIMESTAMP,
        duration_seconds INT64,
        page_count INT64
    >,
    events ARRAY<STRUCT<
        event_type STRING,
        timestamp TIMESTAMP,
        page_url STRING,
        referrer STRING,
        properties STRUCT<
            button_clicked STRING,
            form_submitted BOOL,
            scroll_depth INT64,
            time_on_page INT64
        >
    >>,
    conversion STRUCT<
        converted BOOL,
        conversion_type STRING,
        conversion_value FLOAT64,
        conversion_time TIMESTAMP
    >
)
PARTITION BY DATE(session_info.start_time)
CLUSTER BY user.id, user.country;
```

8 Interview Questions and Discussion Points

8.1 Core Concept Questions

1. **Q: Explain the fundamental differences between normalized, denormalized, and nested schema designs in BigQuery. When would you choose each approach?**

   Expected Answer Points:

   - Normalized: Eliminates redundancy, good for small tables, expensive JOINs
   - Denormalized: Fast queries, data explosion, aggregation issues
   - Nested: Best of both worlds, logical organization, no JOINs needed

- Decision factors: table size, update frequency, query patterns

2. **Q: How does BigQuery's columnar storage architecture impact schema design decisions?**

   Expected Answer Points:

   - Only reads required columns, ignores unused nested fields
   - 99 unused columns don't impact query performance
   - Pay only for data processed
   - Enables wide schemas without performance penalty

3. **Q: What are the performance implications of one-to-many relationships in different schema designs?**

   Expected Answer Points:

   - Normalized: Requires JOINs, becomes expensive at scale
   - Denormalized: Data explosion, aggregation complexity
   - Nested: No JOINs, maintains data integrity, efficient storage

8.2 Performance Optimization Questions

1. **Q: How would you optimize a BigQuery table for both filtering and aggregation queries?**

   Expected Answer Points:

   - Partition by date for time-based filtering
   - Cluster by high-cardinality columns for equality filters
   - Use nested fields to avoid JOINs
   - Consider query patterns in schema design

2. **Q: When should you use partitioning vs clustering, and when should you use both?**

   Expected Answer Points:

   - Partitioning: For cost reduction and time-based queries
   - Clustering: For filter performance and co-location
   - Both: Maximum performance benefit, partition first then cluster
   - Partitioning provides cost estimation, clustering provides performance

3. **Q: How do you handle schema evolution in BigQuery while maintaining performance?**

   Expected Answer Points:

   - Use backward-compatible schema changes
   - Consider data migration strategies
   - Monitor query performance after changes
   - Use views for gradual transitions

8.3 Real-World Scenario Questions

1. **Q: Design a schema for a ride-sharing service processing millions of rides daily. Consider performance, cost, and analytical requirements.**

   Expected Answer Points:

   - Use nested fields for ride details, locations, events
   - Partition by ride date for time-based queries
   - Cluster by customer and driver IDs
   - Include events array for ride lifecycle tracking
   - Consider real-time vs batch processing requirements

2. **Q: How would you migrate a traditional relational schema to BigQuery? What are the key considerations?**

   Expected Answer Points:

   - Analyze query patterns and data relationships
   - Identify opportunities for nested/repeated fields
   - Plan partitioning and clustering strategy
   - Consider data loading and transformation requirements
   - Test performance with representative data volumes

3. **Q: Design a schema for a real-time analytics dashboard requiring sub-second response times.**

   Expected Answer Points:

   - Use materialized views for pre-computed aggregations
   - Implement effective partitioning and clustering
   - Consider data freshness vs performance trade-offs
   - Use nested fields to minimize JOINs
   - Optimize for specific dashboard queries

9 Advanced Topics and Best Practices

9.1 Schema Design Patterns

Star Schema Adaptation:

- **Fact Tables**: Use nested fields for dimension data
- **Dimension Tables**: Keep small dimensions normalized
- **Bridge Tables**: Use arrays for many-to-many relationships
- **Slowly Changing Dimensions**: Handle with nested structures

Data Vault Adaptation:

- **Hubs**: Use STRUCT for entity information
- **Links**: Use ARRAY for relationship data
- **Satellites**: Use nested fields for descriptive data
- **Point-in-Time Tables**: Use arrays for historical data

9.2 Performance Monitoring and Optimization

Key Metrics to Monitor:

- **Query Performance**: Execution time and slot utilization
- **Cost Efficiency**: Bytes processed and query costs
- **Partition Effectiveness**: Partition pruning statistics
- **Clustering Quality**: Data distribution and sorting

Optimization Strategies:

- **Regular Analysis**: Monitor query patterns and performance
- **Schema Refinement**: Adjust based on usage patterns
- **Partition Management**: Set appropriate expiration policies
- **Clustering Maintenance**: Monitor and adjust clustering columns

10 Key Takeaways and Best Practices

10.1 Schema Design Principles

- **Think in Arrays**: Use ARRAY data types for repeated values
- **Pre-join with STRUCTs**: Organize related data logically
- **Consider Query Patterns**: Design for your most common queries
- **Balance Flexibility and Performance**: Choose appropriate normalization level
- **Plan for Scale**: Design schemas that can handle growth

10.2 Performance Optimization

- **Partition by Date**: Most common and effective strategy
- **Cluster by High-Cardinality Columns**: Improve filter performance
- **Use Nested Fields**: Avoid expensive JOINs
- **Monitor and Optimize**: Continuously improve based on usage patterns
- **Consider Cost vs Performance**: Balance optimization with cost

10.3 Cost Management

- **Accurate Cost Estimation**: Use partitioning for better planning
- **Reduce Bytes Processed**: Use clustering and selective queries
- **Set Data Lifecycle**: Use partition expiration for cost control
- **Monitor Usage**: Track and optimize query patterns
- **Use Materialized Views**: Pre-compute expensive aggregations

10.4 Implementation Guidelines

- **Start with Partitioning**: Foundation for performance
- **Add Clustering Strategically**: Based on query patterns
- **Use Nested Fields Judiciously**: When relationships are clear
- **Test with Real Data**: Validate performance assumptions
- **Document Design Decisions**: For future maintenance