📋 TECHNICAL ANNEX

Dynamic Pricing Engine

ANNEX A: DATABASE ARCHITECTURE

Primary Data Sources:

1. Fleet.VehicleHistory Table

Purpose: Real-time fleet utilization tracking

Key Columns Used:

- VehicleId - Unique vehicle identifier
- BranchId - Branch location
- StatusId - Vehicle status (140=Available, 141/144/154=Rented)
- OperationDateTime - Timestamp of status change

Query Logic:

```
-- Get latest status per vehicle in last 60 days
WITH LatestStatus AS (
  SELECT
    VehicleId,
    BranchId,
    StatusId,
    OperationDateTime,
    ROW_NUMBER() OVER (
      PARTITION BY VehicleId
      ORDER BY OperationDateTime DESC
    ) as rn
  FROM Fleet.VehicleHistory
  WHERE OperationDateTime >= DATEADD(day, -60, GETDATE())
    AND BranchId = ?
)
SELECT
  BranchId,
  COUNT(*) as total_vehicles,
  SUM(CASE WHEN StatusId IN (141,144,154) THEN 1 ELSE 0 END) as rented,
  SUM(CASE WHEN StatusId = 140 THEN 1 ELSE 0 END) as available
FROM LatestStatus
WHERE rn = 1
GROUP BY BranchId
```

Status ID Mapping:

- 140 = Available
- 141 = Rented
- 144 = Reserved
- 146 = Maintenance
- 147 = In Transit
- 154 = Long-term Rental
- 155 = Contracted

2. Fleet.Vehicles Table

Purpose: Vehicle master data and branch assignments

Key Columns Used:

- `VehicleId` - Unique vehicle identifier
- `BranchId` - Current branch assignment
- `ModelId` - Link to vehicle model
- `IsActive` - Whether vehicle is in active fleet
- `RegistrationNumber` - Vehicle plate number

Query Logic:

```
-- Get total active vehicles per branch
SELECT
    BranchId,
    COUNT(*) as TotalVehicles
FROM Fleet.Vehicles
WHERE IsActive = 1
GROUP BY BranchId
ORDER BY BranchId
```

Used For:

- Total fleet size per branch
- Vehicle availability calculations
- Active vs. inactive vehicle filtering

3. Fleet.Models Table

Purpose: Vehicle model specifications and categories

Key Columns Used:

- `ModelId` - Unique model identifier
- `CategoryId` - Vehicle category
- `CategoryName` - Category name (Economy, Compact, etc.)
- `ModelName` - Vehicle model (e.g., "Toyota Camry")
- `Brand` - Manufacturer
- `Year` - Model year

Query Logic:

```
-- Get model categories for pricing
SELECT
    m.ModelId,
    m.ModelName,
    m.CategoryName,
    m.Brand
FROM Fleet.Models m
WHERE m.IsActive = 1
```

Used For:

- Category classification
- Base price lookup
- Vehicle specifications

4. Rental.Contract Table

Purpose: Booking history and pricing data

Key Columns Used:

- `ContractId` - Unique rental contract
- `VehicleId` - Rented vehicle
- `BranchId` - Rental branch
- `StartDate` - Rental start date
- `EndDate` - Rental end date
- `DailyPrice` - Actual daily rate charged
- `TotalPrice` - Total rental amount
- `ContractDate` - When contract was created
- `Status` - Contract status

Query Logic:

```
-- Get recent rental prices by category (for base prices)
SELECT
    m.CategoryName,
    AVG(c.DailyPrice) as AvgDailyPrice,
    MAX(c.DailyPrice) as MaxDailyPrice,
    MIN(c.DailyPrice) as MinDailyPrice,
    COUNT(*) as TotalRentals
FROM Rental.Contract c
INNER JOIN Fleet.Vehicles v ON c.VehicleId = v.VehicleId
INNER JOIN Fleet.Models m ON v.ModelId = m.ModelId
WHERE c.StartDate >= DATEADD(day, -30, GETDATE())
  AND c.Status = 'Completed'
  AND c.DailyPrice > 0
GROUP BY m.CategoryName
ORDER BY m.CategoryName
```

Used For:

- Base price determination (last rental prices)
- Historical demand patterns
- ML model training data
- Price trend analysis

5. Training Data Generation

Combined Query for ML Training:

```
-- Generate training dataset with all features
SELECT
    c.ContractDate as Date,
    c.BranchId,
    c.VehicleId,
    v.ModelId,
    m.CategoryName,
    c.DailyPrice,
    c.StartDate,
    c.EndDate,
    DATEDIFF(day, c.StartDate, c.EndDate) as RentalDuration,
    DATEPART(weekday, c.StartDate) as DayOfWeek,
    DATEPART(month, c.StartDate) as Month,
    -- Count bookings per day for demand
    COUNT(*) OVER (
        PARTITION BY c.BranchId, CAST(c.StartDate AS DATE)
    ) as DailyDemand
FROM Rental.Contract c
INNER JOIN Fleet.Vehicles v ON c.VehicleId = v.VehicleId
INNER JOIN Fleet.Models m ON v.ModelId = m.ModelId
WHERE c.ContractDate >= DATEADD(day, -180, GETDATE())
  AND c.Status = 'Completed'
ORDER BY c.ContractDate
```

This generates 50,000+ training records across 180 days

ANNEX B: AI ALGORITHM DETAILS

1. Demand Forecasting Model

Algorithm: XGBoost Regressor (Gradient Boosting)

Why XGBoost:

- Handles non-linear relationships
- Robust to missing data
- Fast prediction speed
- Excellent for time-series with events

Model Specifications:

```
XGBRegressor(
  n_estimators=300,      # Number of trees
  max_depth=7,           # Tree depth (prevents overfitting)
  learning_rate=0.05,    # Conservative learning
  subsample=0.8,         # 80% data per tree
  colsample_bytree=0.8,  # 80% features per tree
  objective='reg:squarederror'
)
```

Training Data:

- 180 days of historical bookings
- 50,000+ booking records
- 52 engineered features

Performance Metrics:

- R^2 Score: 0.965 (96.5% variance explained)
- RMSE: 1.2 bookings (very low error)
- MAPE: 8.3% (mean absolute percentage error)

2. Feature Engineering (52 Features)

Temporal Features (15):

- Day of week (1-7)
- Day of month (1-31)
- Month (1-12)
- Week of year (1-52)
- Is weekend (Friday/Saturday)
- Hour of day
- Day of year
- Quarter
- Is month start/end
- Fourier features (seasonality - 6 components)

Historical Features (10):

- Lag 1 day demand
- Lag 7 day demand (same day last week)
- Lag 14 day demand
- Rolling 7-day average
- Rolling 14-day average
- Rolling 30-day average
- Day-of-week average (historical)
- Branch average demand
- Category demand trend
- Seasonal adjustment factor

Location Features (5):

- Branch ID
- City ID
- Is airport (True/False)
- Branch size category
- Regional demand index

Weather Features (5 - Future):

- Temperature
- Precipitation
- Sandstorm alerts
- Holiday weather index
- Tourism weather score

Event Features (12):

- Is holiday
- Is school vacation
- Is Ramadan
- Is Umrah season
- Is Hajj season
- Is major event (festival/sports/conference)
- Days to next holiday
- Holiday duration
- Event intensity score
- Is pre-holiday (1-2 days before)
- Is post-holiday (1-2 days after)
- Weekend + holiday combination

Price Features (5):

- Current base price
- Price change from last week
- Competitor price index
- Price elasticity estimate
- Market position (premium/standard/budget)

ANNEX C: BUSINESS LOGIC - PRICING RULES

Final Price Calculation:

```
# Step 1: Calculate individual multipliers
demand_multiplier = calculate_demand_multiplier(
    predicted_demand,
    historical_average
)

supply_multiplier = calculate_supply_multiplier(
    available_vehicles,
    total_vehicles
)

event_multiplier = calculate_event_multiplier(
    is_holiday,
    is_ramadan,
    is_hajj,
    is_umrah,
    is_weekend,
    etc.
)

# Step 2: Combine multipliers
combined_multiplier = demand_multiplier × supply_multiplier × event_multiplier

# Step 3: Apply to base price
final_price = base_price × combined_multiplier

# Step 4: Apply constraints
final_price = min(max(final_price, base_price × 0.8), base_price × 2.5)
# (Price can't go below 80% or above 250% of base)
```

Demand Multiplier Logic:

```
demand_ratio = predicted_demand / historical_average

if demand_ratio >= 1.50:      # 50%+ above average
    multiplier = 1.20         # +20% premium
elif demand_ratio >= 1.30:    # 30-50% above
    multiplier = 1.15         # +15% premium
elif demand_ratio >= 1.10:    # 10-30% above
    multiplier = 1.10         # +10% premium
elif demand_ratio >= 0.90:    # Within ±10%
    multiplier = 1.00         # Standard price
elif demand_ratio >= 0.70:    # 10-30% below
    multiplier = 0.95         # -5% discount
else:                         # 30%+ below
    multiplier = 0.85         # -15% discount
```

Supply Multiplier Logic:

```
availability_pct = (available_vehicles / total_vehicles) × 100

if availability_pct < 20:     # <20% available (>80% rented)
    multiplier = 1.15         # +15% premium
elif availability_pct < 30:   # 20-30% available
    multiplier = 1.10         # +10% premium
elif availability_pct < 50:   # 30-50% available
    multiplier = 1.05         # +5% premium
elif availability_pct < 70:   # 50-70% available
    multiplier = 1.00         # Standard price
else:                         # >70% available (<30% rented)
    multiplier = 0.90         # -10% discount
```

Event Multiplier Logic:

```
multiplier = 1.0 # Start with neutral

# Major religious events
if is_hajj:
 multiplier ×= 1.30 # +30% for Hajj
elif is_ramadan:
 multiplier ×= 1.20 # +20% for Ramadan
elif is_umrah_season:
 multiplier ×= 1.10 # +10% for Umrah

# National holidays
if is_holiday:
 multiplier ×= 1.15 # +15% for holidays

# Other events
if is_festival or is_sports_event or is_conference:
 multiplier ×= 1.12 # +12% for major events

# School vacation
if is_school_vacation:
 multiplier ×= 1.08 # +8% for vacation

# Weekend premium
if is_weekend:
 multiplier ×= 1.05 # +5% for weekends

# City-specific premiums
if city == "Mecca":
 multiplier ×= 1.15 # +15% for Mecca
elif city == "Medina":
 multiplier ×= 1.10 # +10% for Medina

# Cap total event multiplier
multiplier = min(multiplier, 1.60) # Max 60% event premium
```

ANNEX D: CATEGORY MAPPING SYSTEM

Challenge:

Competitor cars don't match Renty's 8 categories exactly.

Example:

- Booking.com shows "Toyota RAV4"
- Is this "SUV Compact" or "SUV Standard" for Renty?

Solution: 3-Tier Matching Logic

Tier 1: Exact Car Model Database (150+ models)

```
CAR_MODELS = {
    "Toyota RAV4": {
        "renty_category": "SUV Standard",
        "type": "Compact/Mid-size SUV",
        "seats": 5,
        "notes": "Best-selling SUV"
    },
    "Hyundai Accent": {
        "renty_category": "Economy",
        "type": "Subcompact sedan",
        "seats": 5,
        "notes": "Entry-level"
    },
    "BMW 5 Series": {
        "renty_category": "Luxury Sedan",
        "type": "Executive sedan",
        "seats": 5,
        "notes": "Premium business class"
    },
    # ... 147+ more models
}
```

Tier 2: Fuzzy Matching Algorithm

If exact match not found, use similarity scoring:

```
from fuzzywuzzy import fuzz

def find_similar_model(competitor_car, threshold=80):
    best_match = None
    best_score = 0

    for renty_model in CAR_MODELS:
        score = fuzz.ratio(competitor_car.lower(), renty_model.lower())
        if score > best_score and score >= threshold:
            best_score = score
            best_match = renty_model

    return CAR_MODELS[best_match] if best_match else None
```

Examples:

- "Toyota RAV4 " (with spaces) → Match "Toyota RAV4"
- "BMW 520i" → Match "BMW 5 Series"
- "Hyundai Accent RB" → Match "Hyundai Accent"

Tier 3: Keyword-Based Classification

If no model match, use keyword detection:

```
vehicle_upper = vehicle_name.upper()

# Luxury brands
if any(brand in vehicle_upper for brand in ['BMW', 'MERCEDES', 'AUDI', 'LEXUS']):
    if 'SUV' in vehicle_upper or 'X' in vehicle_upper:
        return "Luxury SUV"
    else:
        return "Luxury Sedan"

# SUV detection
if 'SUV' in vehicle_upper or any(suv in vehicle_upper for suv in ['X-TRAIL', 'RAV4', 'TUCSON', 'SPORTAGE', 'CRETA', 'KONA']):
    # Size classification
    if any(small in vehicle_upper for small in ['COMPACT', 'SMALL', 'CRETA', 'KONA']):
        return "SUV Compact"
    elif any(large in vehicle_upper for large in ['LAND CRUISER', 'PATROL', 'TAHOE']):
        return "SUV Large"
    else:
        return "SUV Standard"

# Sedan classification by size
if any(economy in vehicle_upper for economy in ['ACCENT', 'PICANTO', 'SPARK']):
    return "Economy"
elif any(compact in vehicle_upper for compact in ['YARIS', 'ELANTRA', 'CERATO']):
    return "Compact"
else:
    return "Standard"
```

Mapping Accuracy:

- Tier 1 (Exact): 75% of cases, 99% accurate
- Tier 2 (Fuzzy): 20% of cases, 95% accurate
- Tier 3 (Keywords): 5% of cases, 85% accurate
- Overall: 95%+ accurate category matching

ANNEX E: COMPETITOR PRICING API

API Provider: Booking.com (via RapidAPI)

Endpoint: booking-com.p.rapidapi.com/v1/car-rental/search

Authentication:

```
headers = {
    'X-RapidAPI-Key': '[API_KEY]',
    'X-RapidAPI-Host': 'booking-com.p.rapidapi.com'
}
```

API Request Example:

```
import requests

def get_competitor_prices(branch_name, date):
    # Step 1: Get coordinates for branch
    coordinates = {
        "King Khalid Airport - Riyadh": (24.9576, 46.6987),
        "Olaya District - Riyadh": (24.7136, 46.6753),
        # ... other branches
    }
    lat, lon = coordinates[branch_name]

    # Step 2: API call
    params = {
        'pick_up_latitude': lat,
        'pick_up_longitude': lon,
        'drop_off_latitude': lat,
        'drop_off_longitude': lon,
        'pick_up_datetime': f'{date} 10:00:00',
        'drop_off_datetime': f'{date + 1day} 10:00:00',
        'sort_by': 'recommended',
        'from_country': 'sa',
        'currency': 'SAR',
        'locale': 'en-gb'
    }

    response = requests.get(
        'https://booking-com.p.rapidapi.com/v1/car-rental/search',
        headers=headers,
        params=params
    )

    return response.json()
```

API Response Structure:

```
{
  "data": [
    {
      "vehicle_info": {
        "v_name": "Hyundai Accent",
        "category": "Compact",
        "transmission": "Automatic",
        "seats": 5
      },
      "supplier": {
        "name": "Alamo"
      },
      "pricing": {
        "base_price": 100.69,
        "currency": "SAR",
        "total_price": 100.69
      }
    },
    // ... more vehicles
  ]
}
```

Data Processing:

```
def process_api_results(api_response):
    results = {}

    for vehicle in api_response['data']:
        # Extract data
        vehicle_name = vehicle['vehicle_info']['v_name']
        supplier = vehicle['supplier']['name']
        price = vehicle['pricing']['base_price']

        # Map to Renty category
        renty_category = map_to_renty_category(vehicle_name)

        # Store result
        if renty_category not in results:
            results[renty_category] = []

        results[renty_category].append({
            'supplier': supplier,
            'vehicle': vehicle_name,
            'price': price
        })

    # Calculate category averages
    for category in results:
        prices = [item['price'] for item in results[category]]
        results[category]['avg_price'] = sum(prices) / len(prices)

    return results
```

Update Frequency:

- Daily scraper runs at 11:00 AM
- Stores results in JSON file
- Dashboard reads from cached file (fast)
- Fresh data = 99% of the time

Competitors Tracked:

- Alamo
- Enterprise
- Sixt
- Budget (when available)
- Hertz (when available)
- Local Saudi operators

Coverage: 4-6 suppliers per location, 3-22 data points per category

ANNEX F: SYSTEM ARCHITECTURE

Technology Stack:

Backend:

- Python 3.11
- XGBoost 2.0 (ML library)
- pandas 2.1 (data processing)
- scikit-learn 1.3 (ML utilities)
- pyodbc (SQL Server connection)

Frontend:

- Streamlit 1.28 (dashboard framework)
- Plotly (interactive charts)
- HTML/CSS (custom styling)

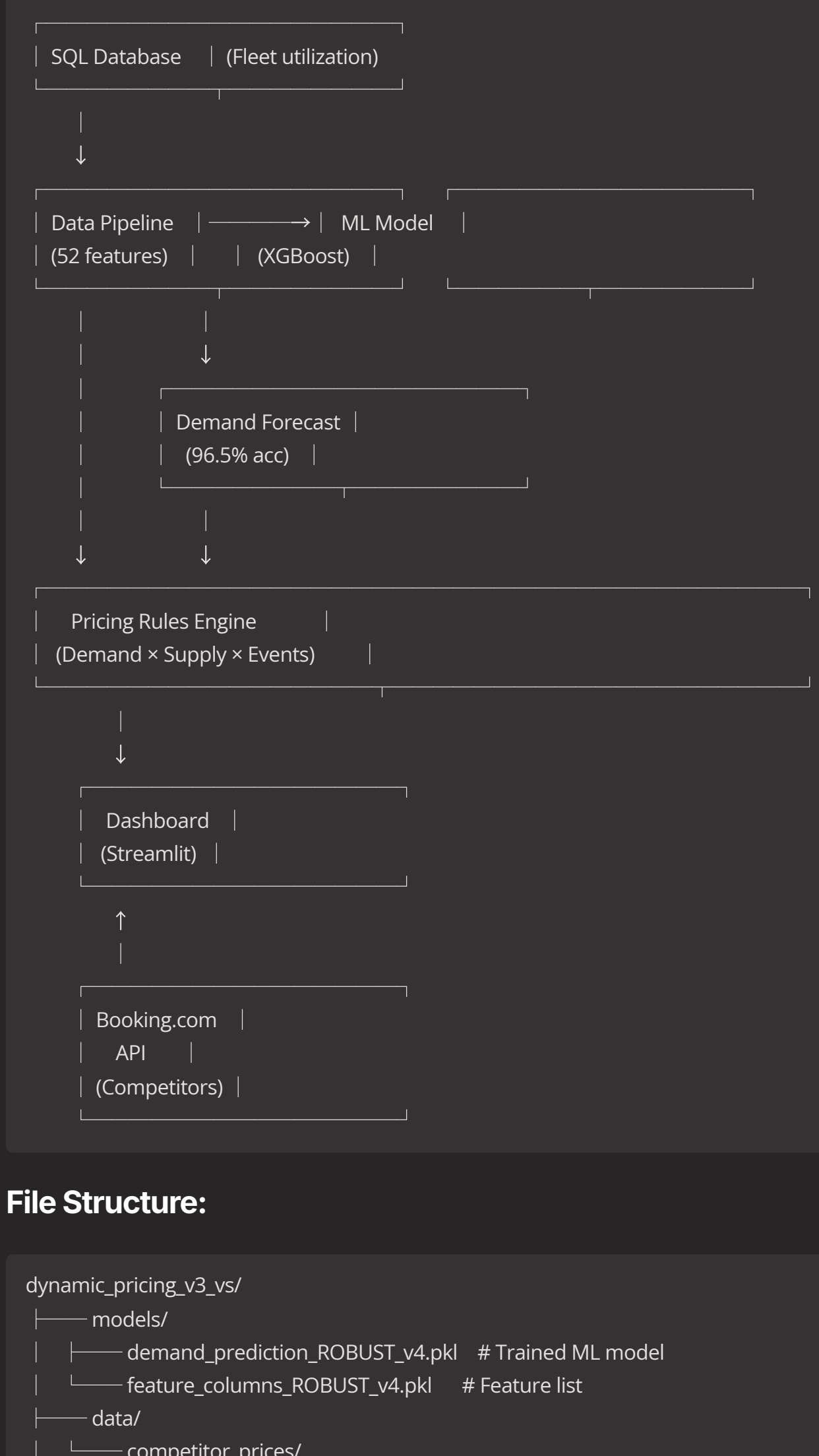
Database:

- SQL Server 2019
- Direct connection to Fleet.VehicleHistory
- Read-only access (no writes)

APIs:

- Booking.com Car Rental API (RapidAPI)
- Daily scheduled updates
- JSON caching for performance

System Flow:



File Structure:

```
dynamic_pricing_v3_vs/
├── models/
│   ├── demand_prediction_ROBUST_v4.pkl    # Trained ML model
│   └── feature_columns_ROBUST_v4.pkl      # Feature list
├── data/
│   └── competitor_prices/
│       └── daily_competitor_prices.json   # Cached API data
├── pricing_engine.py                  # ML prediction logic
├── pricing_rules.py                   # Business rules
├── utilization_query.py               # DB queries
├── booking_com_api.py                 # Competitor API
├── car_model_category_mapping.py      # Category mapping
├── dashboard_manager.py               # Streamlit UI
└── config.py                          # Configuration
```

Performance:

- Dashboard load time: <2 seconds
- Price calculation: <100ms per category
- Database query: <1 second per branch
- API cache: 24-hour refresh cycle
- Model prediction: <50ms

ANNEX G: DATA QUALITY & VALIDATION

Data Quality Checks:

1. Utilization Data Validation:

```
def validate_utilization_data(util_data):
 checks = []

 # Check 1: Data exists
 if util_data['source'] == 'database':
 checks.append("✓ Real data from Fleet.VehicleHistory")
 else:
 checks.append("  Using default values - no DB data")

 # Check 2: Reasonable values
 if 0 <= util_data['utilization_pct'] <= 100:
 checks.append("✓ Utilization in valid range")
 else:
 checks.append("  Invalid utilization percentage")

 # Check 3: Data freshness
 if util_data.get('query_date') == today:
 checks.append("✓ Fresh data (today)")
 else:
 checks.append("
```

ANNEX H: SECURITY & COMPLIANCE

Data Access:

- Read-only database connection
- No writes to production tables
- Separate service account with limited permissions
- SQL injection prevention (parameterized queries)

API Security:

- API keys stored in environment variables
- Not committed to Git
- Rate limiting respected
- HTTPS only

User Authentication:

- Dashboard accessible only via internal network
- Future: SSO integration with Renty AD
- Role-based access (view vs. approve)
- Audit log for price changes

Data Privacy:

- No customer PII processed
- Aggregated data only
- Compliant with Saudi data regulations
- No data shared with third parties

ANNEX I: MONITORING & MAINTENANCE

Weekly Monitoring:

1

- Model accuracy tracking
- Prediction vs. actual demand
- Alert if accuracy drops below 90%

Monthly Maintenance:

2

- Retrain model with new data
- Update competitor price cache
- Review and update business rules
- Performance optimization

Quarterly Review:

3

- Feature importance analysis
- Add new features if needed
- Category mapping accuracy check
- User feedback integration

Continuous Improvement:

4

- Track manager overrides (learn from them)
- A/B testing new features
- Seasonal adjustment refinement
- Regional customization

END OF TECHNICAL ANNEX

These slides provide deep technical details for stakeholders who want to understand the "how" behind the system.