F1 Race Prediction System - Architecture Documentation

Table of Contents

---

1. High-Level System Architecture

Diagram 1: Complete System Overview

See Diagram: "F1 System Architecture Diagrams" (Mermaid diagram above)

Architecture Layers

The system follows a **5-layer architecture pattern**:

Layer 1: User Interface (Presentation Layer)

- **Technology**: HTML5, CSS3, Vanilla JavaScript
- **Components**:
  - Glassmorphism header with gradient title
  - Race conditions input panel (4 controls)
  - Dynamic driver cards (unlimited, add/remove/edit)
  - Results visualization (podium + insights)
  - Export functionality (5 formats)

Responsibilities:

- Capture user input
- Display predictions visually
- Handle user interactions
- Trigger calculations
- Export results

Design Pattern: Single-Page Application (SPA)

Layer 2: Prediction Logic (Business Logic Layer)

- **Components**:
  - API availability checker
  - Prediction router (ML vs Fallback)
  - Result processor
  - Export generators

Responsibilities:

- Determine which prediction model to use
- Route requests appropriately
- Process and format results
- Handle errors gracefully

Design Pattern: Strategy Pattern (ML or Rule-Based)

Layer 3: API Server (Service Layer)

- **Technology**: Flask 3.0.0 (Python)
- **Components**:
  - Health check endpoint (`/health`)
  - Model info endpoint (`/model-info`)
  - Prediction endpoint (`/predict`)
  - Feature vector creator
  - Error handler

Responsibilities:

- Serve ML model predictions
- Create 27-feature vectors from 8 inputs
- Apply preprocessing (impute + scale)
- Return JSON responses
- Handle CORS for browser requests

Design Pattern: REST API, Singleton (model loading)

Port: 5000
Host: localhost (127.0.0.1)
Protocol: HTTP
Response Format: JSON

Layer 4: ML Model (Data Science Layer)

- **Components**:
  - Neural Network (3 hidden layers)
  - Preprocessing pipeline (imputer + scaler)
  - Feature transformer
  - Probability calculator

Responsibilities:

- Load trained model from disk
- Preprocess input features
- Perform inference
- Return probability [0-1]

Design Pattern: Pipeline Pattern

Inference Time: <50ms
Memory Usage: ~200 MB (model loaded)

Layer 5: Data Storage (Persistence Layer)

- **Components**:
  - Model files (.pkl format)
  - Preprocessor objects (scaler, imputer)
  - Metadata (JSON)
  - Reference data (drivers, constructors, circuits)

Responsibilities:

- Store trained models
- Persist preprocessing objects
- Maintain reference databases

- Provide metadata

Storage Locations:

- `models/` - Model artifacts (15 MB)
- `data/` - JSON databases (500 KB)
- `data_raw/` - Original CSV files (30 MB)

System Characteristics

Deployment Model: Local-first architecture

- **Advantages**:
  - No cloud dependency
  - Data privacy (runs locally)
  - Fast response (<100ms)
  - No API costs
  - Offline capable

Scalability:

- **Vertical**: Can handle 100+ drivers on single machine
- **Horizontal**: Can deploy multiple instances (different ports)
- **Bottleneck**: ML inference (parallelizable)

Reliability:

- **Uptime**: 99.9%+ (local server)
- **Failover**: Automatic fallback (91.87%)
- **Error Recovery**: Graceful degradation
- **No single point of failure**

---

2. Data Flow Architecture

Diagram 2: Prediction Request Flow

See Diagram: "F1 System - Data Flow Diagram" (Sequence diagram above)

Request Lifecycle

Phase 1: User Input (0-10 seconds)

```
User Action → Browser Event
  ↓
Input Validation (client-side)
  ↓
JavaScript captures values
  ↓
Stores in drivers[] array
```

Phase 2: API Check (<50ms)

```
JavaScript → HTTP GET /health
  ↓
Flask responds OR timeout
  ↓
Set apiAvailable flag
  ↓
Update UI status badge
```

Phase 3A: Online Prediction Path (30-50ms per driver)

```
JavaScript → HTTP POST /predict
  ↓
Flask receives JSON
  ↓
Validate required fields
  ↓
Create 27-feature vector
  ↓
Apply imputer (handle NaN)
  ↓
Apply scaler (standardize)
  ↓
Neural Network forward pass
  ↓
Get probability [0-1]
  ↓
Calculate podium probabilities
  ↓
Format JSON response
  ↓
Return to JavaScript
```

Phase 3B: Offline Prediction Path (2-5ms per driver)

```
JavaScript local execution
  ↓
Layer 1: Grid calculation (exp decay)
  ↓
Layer 2: Form calculation (wins, podiums)
  ↓
Layer 3: Team calculation (constructor avg)
  ↓
Layer 4: Circuit calculation (history)
  ↓
Apply weather multiplier
  ↓
Return probability [0-100]
```

Phase 4: Results Display (<10ms)

```
JavaScript receives predictions
  ↓
Sort drivers by win probability
  ↓
Render podium (top 3)
  ↓
Generate insight cards
  ↓
Update DOM with animations
  ↓
Show export buttons
  ↓
User sees results
```

Total Time:

- **Single driver (online)**: 50-100ms
- **Single driver (offline)**: 5-10ms
- **5 drivers (online)**: 200-400ms
- **5 drivers (offline)**: 20-50ms

Data Format Transformations

User Input → API Request:

```javascript
```

```
// User enters (8 simple values)
{
  gridPosition: 1,
  driverAvgPos: 2.5,
  recentWins: 3,
  recentPodiums: 4,
  constructorAvgPos: 1.5,
  circuitAvgPos: 2.0,
  fastestLapRate: 40,
  finishRate: 95
}

// Transforms to (27 engineered features)
[
  1.0,    // grid_position
  1.0,    // grid_position_squared
  2.5,    // driver_avg_position_5
  1.5,    // driver_best_position_5 (estimated)
  4.5,    // driver_worst_position_5 (estimated)
  2.0,    // driver_position_std_5 (default)
  2.5,    // driver_avg_position_10
  6,      // driver_wins_10 (scaled)
  8,      // driver_podiums_10 (scaled)
  // ... (27 total)
]
```

API Response → User Display:

```javascript
```

```
// API returns
{
  "success": true,
  "predictions": {
    "win_probability_percent": 84.23,
    "podium": {
      "p1": 84.23,
      "p2": 12.5,
      "p3": 8.2
    }
  }
}

// Displays as
Win Probability: 84%
🥇 84% | 🥈 13% | 🥉 8%
```

---

3. Training Pipeline Architecture

Diagram 3: ML Training Workflow

See Diagram: "F1 ML Training Pipeline" (Flowchart above)

Training Phases

Phase 1: Data Acquisition (Manual, 5 minutes)

- Download Kaggle dataset (ZIP file, 30 MB)
- Extract 14 CSV files
- Place in `data_raw/` directory

Phase 2: Data Processing (~30 seconds)

```python
python process_data.py

Input:  races.csv, results.csv, qualifying.csv, etc.
Output: drivers.json, constructors.json, circuits.json, historical.json
```

Operations:

- Load 6 CSV files into Pandas DataFrames
- Calculate driver statistics (wins, podiums, rates)

- Calculate constructor statistics
- Analyze grid position win rates
- Export to JSON format

Output Statistics:

- 861 drivers processed
- 212 constructors processed
- 77 circuits processed
- Historical win rates calculated

Phase 3: Feature Engineering (~2 minutes)

```python
python train_ml_models.py
# Step 1: load_and_engineer_features()

Input:  Raw CSV data (26,759 results)
Output: Feature matrix (25,107 × 27)
```

Operations:

1. **Merge datasets**: races + results + qualifying + standings
2. **Sort chronologically**: Ensure temporal order
3. **Rolling window calculations**:
   - For each race (26,759 iterations):
     - Look back at driver history
     - Calculate L5, L10 statistics
     - Compute team performance
     - Aggregate circuit-specific stats
4. **Derive additional features**:
   - Win rates, podium rates
   - Momentum indicators
   - Consistency scores
5. **Filter**: Remove first 100 races (insufficient history)

Result: 25,107 complete feature vectors

Phase 4: Train/Test Split (Instant)

```
Time-based split (no shuffling!)

Training:  1950-2018 → 22,549 samples (89.8%)
Testing:   2019-2024 → 2,558 samples (10.2%)
```

Rationale:

- Simulates real-world: predict future from past
- No data leakage
- Realistic evaluation

Phase 5: Preprocessing (~5 seconds)

```python
# Missing value imputation
imputer = SimpleImputer(strategy='median')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Feature standardization
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Saved artifacts:

- `imputer.pkl` (for deployment)
- `scaler.pkl` (for deployment)

Phase 6: Model Training (~4 minutes total)

Training Order:

1. **XGBoost** (~25 seconds)
   - 200 boosting rounds
   - Tree depth 6
   - Histogram algorithm
2. **Random Forest** (~18 seconds)
   - 200 trees

- Max depth 10
- Parallel training (n_jobs=-1)

3. **Gradient Boosting** (~32 seconds)
   - 150 estimators
   - Learning rate 0.1
   - Sequential training
4. **Neural Network** (~45 seconds)
   - 200 epochs (with early stopping)
   - Batch size 32
   - Adam optimizer
   - Validation split 10%
5. **Logistic Regression** (~8 seconds)
   - 1000 iterations
   - L2 regularization
   - Fast convergence

Total Training Time: ~4 minutes for all 5 models

Phase 7: Model Evaluation (~10 seconds)

```python
for model in [XGBoost, RF, GB, NN, LR]:
    y_pred = model.predict(X_test)
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    metrics = calculate_metrics(y_test, y_pred, y_pred_proba)
    # accuracy, precision, recall, f1, roc_auc
```

Generates:

- `model_comparison.csv`
- `training_report.txt`
- Feature importance CSVs

Phase 8: Model Selection & Persistence (~2 seconds)

```python
```

```
best_model = compare_models()  # Neural Network: 95.54%

# Save all models
for model_name, model in models.items():
    pickle.dump(model, open(f'{model_name}.pkl', 'wb'))

# Save metadata
json.dump(model_info, open('model_info.json', 'w'))
```

Output Files (15 MB total):

- 5 model .pkl files
- 2 preprocessing .pkl files
- model_info.json
- 2 feature importance .csv files
- training_report.txt

---

4. Neural Network Architecture

Diagram 4: Neural Network Structure

See Diagram: "Neural Network Architecture Diagram" (Graph above)

Network Specifications

Architecture Type: Multi-Layer Perceptron (MLP)
Framework: Scikit-learn MLPClassifier
Total Parameters: ~25,000 trainable weights

Layer-by-Layer Breakdown:

Input Layer:

- **Neurons**: 27 (one per feature)
- **Activation**: None (pass-through)
- **Shape**: (batch_size, 27)

Hidden Layer 1:

- **Neurons**: 128
- **Activation**: ReLU (Rectified Linear Unit)
  - Formula: `f(x) = max(0, x)`

- Purpose: Introduce non-linearity

- **Weights**: 27 × 128 = 3,456 parameters
- **Biases**: 128 parameters
- **Total**: 3,584 parameters
- **Dropout**: None (using early stopping instead)

Hidden Layer 2:

- **Neurons**: 64
- **Activation**: ReLU
- **Weights**: 128 × 64 = 8,192 parameters
- **Biases**: 64 parameters
- **Total**: 8,256 parameters

Hidden Layer 3:

- **Neurons**: 32
- **Activation**: ReLU
- **Weights**: 64 × 32 = 2,048 parameters
- **Biases**: 32 parameters
- **Total**: 2,080 parameters

Output Layer:

- **Neurons**: 1 (binary classification)
- **Activation**: Sigmoid
  - Formula: `σ(x) = 1 / (1 + e^(-x))`
  - Output: Probability [0, 1]
- **Weights**: 32 × 1 = 32 parameters
- **Biases**: 1 parameter
- **Total**: 33 parameters

Total Network Parameters: 3,584 + 8,256 + 2,080 + 33 = **13,953 parameters**

Training Configuration

Optimizer: Adam (Adaptive Moment Estimation)

- **Learning rate**: Adaptive (starts high, decreases)

- **Beta1**: 0.9 (momentum)
- **Beta2**: 0.999 (RMSprop)
- **Epsilon**: 1e-8 (numerical stability)

Loss Function: Binary Cross-Entropy

```
Loss = -[y·log(ŷ) + (1-y)·log(1-ŷ)]
```

Regularization:

- **L2 penalty (alpha)**: 0.0001
- **Early stopping**: Yes (validation patience)
- **Validation split**: 10% of training data

Training Process:

```
Epoch 1:  Loss: 0.234 | Val Loss: 0.198
Epoch 2:  Loss: 0.187 | Val Loss: 0.165
Epoch 3:  Loss: 0.156 | Val Loss: 0.142
...
Epoch 47: Loss: 0.089 | Val Loss: 0.095
Epoch 48: Loss: 0.088 | Val Loss: 0.096 ← Validation loss increases
→ Early stopping triggered
→ Restore best weights (Epoch 47)
```

Final Performance:

- Training loss: 0.089
- Validation loss: 0.095
- Test accuracy: 95.54%
- No significant