# ML Feature DSL (FDSL)

September 7, 2025

**Abstract**

In machine learning pipelines, feature engineering is a critical yet often fragmented process. Teams frequently duplicate efforts in defining features across SQL queries, Python scripts, monitoring configurations, and API schemas, leading to inconsistencies, errors, and maintenance challenges. The ML Feature DSL (FDSL) addresses these issues by providing a declarative, human-readable language for defining features once and automatically generating compatible code for various environments. This proposal outlines the FDSL's design, implementation, benefits, examples, and potential enhancements, demonstrating how it streamlines ML workflows and promotes consistency across teams.

# Contents

# 1 Overview

The ML Feature DSL (FDSL) is a domain-specific language designed to enable clear, reusable, and consistent definitions of features used in machine learning models. By abstracting complex SQL or Python transformation logic into a simple declarative format, FDSL allows teams to define feature logic once and generate compatible code across training, serving, monitoring, and reporting environments.

In traditional ML pipelines, features are often redefined in multiple places: data scientists might write Pandas code for prototyping, engineers implement SQL for production serving, and monitoring teams configure separate drift checks. This redundancy leads to discrepancies, bugs, and wasted effort. FDSL solves this by serving as a single source of truth, ensuring that all generated artifacts remain synchronized.

Key benefits include:

- **Consistency**: Eliminates variations in feature logic across environments.
- **Reusability**: Features can be shared and audited easily via version control.
- **Efficiency**: Automates code generation, reducing manual coding time.
- **Scalability**: Supports integration with ML platforms like Feature Stores or monitoring tools.

FDSL is particularly helpful for organizations with collaborative ML teams, where maintaining alignment between data science, engineering, and operations is crucial for reliable model performance.

# 2 Goals

The primary objectives of FDSL are:

- Define machine learning features in a human-readable, declarative format.
- Generate code for multiple targets (Python, SQL, API schemas, monitoring).
- Eliminate code duplication and inconsistency across teams.
- Make feature logic auditable and shareable.

By achieving these, FDSL not only accelerates development but also enhances model reliability through standardized feature handling.

# 3 DSL Syntax (Input)

FDSL uses a straightforward syntax where each feature is defined on a new line in the format:

```
feature: feature_name = expression
```

The `expression` supports:

- Arithmetic operators: `+, -, *, /`

- Comparison operators: `<, >, <=, >=, ==, !=`
- Logical operators: `AND, OR`
- Function calls: e.g., `hour(login_time)`
- `Parentheses for grouping.`

Features can reference raw input variables or other features, enabling composability.

### 3.1 Example Input (Simple Features)

```
feature: is_active = days_since_last_login < 30
feature: low_spender = total_spent < 100
feature: no_recent_support = support_tickets_last_90d == 0
```

### 3.2 Example Input (Complex Features)

```
feature: is_recent_login = days_since_last_login < 14
feature: active_last_month = monthly_sessions_last_30d > 5
feature: multi_device_user = device_count > 1

feature: avg_order_value = total_spent / total_orders
feature: high_spender = total_spent > 1000
feature: frequent_buyer = total_orders > 10
feature: last_purchase_recent = days_since_last_purchase < 30

feature: subscription_active = subscription_status == "active"
feature: churn_risk = days_since_subscription_renewal > 45

feature: frequent_complainer = support_tickets_last_90d > 3
feature: unresolved_issues = open_support_tickets > 0

feature: viewed_products_last_week = product_views_last_7d > 10
feature: marketing_engaged = marketing_email_clicks_last_30d > 3
```

Note: String literals like `"active"` are supported in expressions.

## 4 Implementation Details

FDSL is implemented using ANTLR for parsing the grammar (defined in `fdsl.g4`), which generates a parse tree. A custom listener (`CustomListener.py`) traverses this tree to construct an Abstract Syntax Tree (AST). The AST is then traversed in post-order to extract tokens for code generation.

Code generators for each target (`PythonGenerator.py`, `SQLGenerator.py`, `YAMLGenerator.py`, `JSONGenerator.py`) use a stack-based approach on the post-order traversal:

- Operators pop operands and push expressions.
- `FeatureDefinition` nodes trigger assignment emission.

This ensures accurate reconstruction of expressions while adapting to target-specific syntax (e.g., Python uses `&` for AND, SQL uses `CASE WHEN` for boolean features).

YAML drift checks are currently mocked based on feature name patterns, as historical data for real statistical computations is unavailable.

# 5 Outputs Generated Automatically

From the DSL input, FDSL generates the following outputs.

## 5.1 1. Python (for Training in Pandas)

For the simple example:

```
df["is_active"] = (df["days_since_last_login"] < 30)
df["low_spender"] = (df["total_spent"] < 100)
df["no_recent_support"] = (df["support_tickets_last_90d"] == 0)
```

For the complex example, similar lines are generated for each feature.

## 5.2 2. SQL (for Serving)

Boolean features are expressed as `CASE WHEN` clauses. For the simple example:

```
CASE WHEN (days_since_last_login < 30) THEN TRUE ELSE FALSE END
    ↪ AS is_active,
CASE WHEN (total_spent < 100) THEN TRUE ELSE FALSE END AS
    ↪ low_spender,
CASE WHEN (support_tickets_last_90d = 0) THEN TRUE ELSE FALSE END
    ↪  AS no_recent_support
```

Note: The SQL output has been updated to use equality `=` instead of `==`, and `<>` for `!=`.

## 5.3 3. YAML (for Monitoring, Mocked)

Drift checks are simulated. For the simple example:

```
features:
  - name: is_active
    drift_check: ks_test
    threshold: 0.05
  - name: low_spender
    drift_check: js_divergence
    threshold: 0.1
  - name: no_recent_support
    drift_check: psi
    threshold: 0.07
```

### 5.4 4. JSON (for API Schemas)

Captures input variables and features. For the simple example:

```
{
  "input": {
    "days_since_last_login": "number",
    "total_spent": "number",
    "support_tickets_last_90d": "number"
  },
  "features": [
    "is_active",
    "low_spender",
    "no_recent_support"
  ]
}
```

For the complex example, the schema includes all unique inputs with type `number` (future work: type inference).

## 6 Known Limitations

- Limited support for advanced functions or aggregations.
- YAML drift checks are mocked and not data-driven.
- No automatic type inference in JSON (defaults to `number`).
- Potential parse errors for unquoted strings in some contexts.

## 7 Future Improvements and Extensions

To enhance FDSL, consider the following:

- **Type Inference and Validation**: Analyze expressions to infer types (e.g., string, boolean) for JSON schemas and add runtime checks.
- **Real Drift Monitoring**: Integrate with libraries like Evidently or Great Expectations to compute actual drift metrics from historical data.
- **Extended Grammar**: Support aggregations (e.g., `sum`, `avg`), conditional expressions (`if-then`), and metadata annotations (e.g., comments for descriptions).
- **Additional Targets**: Generate Spark SQL, TensorFlow transformations, or HTML documentation for features.
- **Integration with Tools**: Connect to feature stores (e.g., Feast) or CI/CD pipelines for automated deployment.
- **Testing Framework**: Add unit tests for parsers and generators, plus validation against sample datasets.
- **User-Friendly CLI**: Develop a command-line tool for parsing, generating, and visualizing ASTs.

These enhancements would make FDSL a robust tool for enterprise ML pipelines.