

# Accelerating Python for real-time ML with Chalk

Chase Haddleton, Engineer, Chalk

right data  
right place  
right time

# why is real-time ML important?

Fraud detection, RecSys, portfolio risk (etc.) solutions need both

- Fresh features
- Low-latency < 10ms

Out of date or slow == real cost to the business

> Inevitably forced to build a bespoke (custom) solution

# why is real-time ML hard?

Need deep expertise across very different domains

- ML, low-latency systems, scale-out architectures

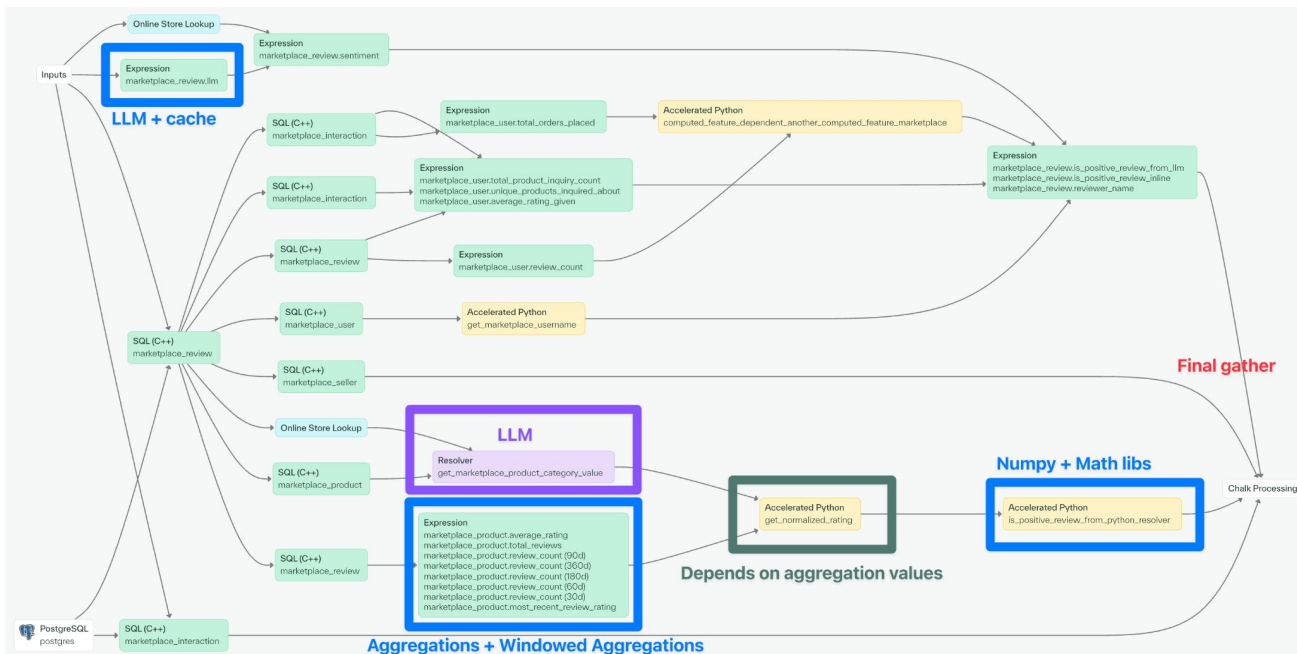
Sprawling tech, stitched into brittle pipelines

- Pytorch, SciKit, Sagemaker, Keras, Data wrangling
- Scala, Rust, dbt, Spark, Airflow, Iceberg
- Kubernetes, Terraform, KEDA, Workflow orchestration

In addition, ML teams want to

- Ship models quickly
- Incorporate new data sources and types (unstructured)
- React to real-time signals

# chalk query --in review.id=241 --out normalized\_rating --out <other\_features>



## Features

features needs to be easy for the user to express i.e. not YAML

```
@features
class Transaction:
    id: int

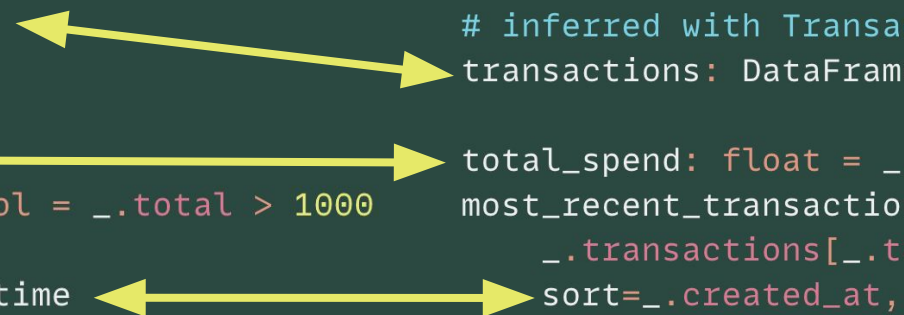
    # user_id: int
    user_id: User.id
    user: User

    total: float
    is_expensive: bool = _.total > 1000
    created_at: datetime

@features
class User:
    id: id
    name: str

    # inferred with Transaction.user_id
    transactions: DataFrame[Transaction]

    total_spend: float = _.transactions[_.total].sum()
    most_recent_transaction: float = F.max_by(
        _.transactions[_.total],
        sort=_.created_at,
    )
```





# Resolvers

Chalk expressions

Python

SQL

```
1 def compute_total(  
2     subtotal: Transaction.subtotal,           # float  
3     discount: Transaction.discount,           # float | None  
4     tax_rate: Transaction.locale.sales_tax_rate, # float  
5     is_taxable: Transaction.taxable,         # bool  
6 ) → Transaction.total:  
7  
8     if discount is not None:  
9         # Apply any discount, but don't allow it to become negative.  
10        subtotal = max(subtotal - discount, 0.0)  
11  
12    if not is_taxable or tax_rate is None:  
13        return subtotal  
14  
15    return subtotal * (1.0 + tax_rate)  
16
```

```
@features  
class User:  
    id: id  
    name: str  
  
    # inferred with Transaction.user_id  
    transactions: DataFrame[Transaction]  
  
    total_spend: float = _.transactions[_.total].sum()  
    most_recent_transaction: float = F.max_by(  
        _.transactions[_.total],  
        sort=_.created_at,  
    )
```

enrich\_transaction.chalk.sql

```
-- resolves: Transaction  
-- source: bigquery  
select  
    txn_id as id,  
    desc as description  
    receipt_id,  
from enriched_transaction_memos
```

The diagram illustrates a hybrid LLM + SQL architecture for recommendation systems. It shows the flow of data from various sources through different processing layers to generate recommendations.

**Data Sources:**

- PostgreSQL postgres
- Online Store Lookup
- Inputs

**Processing Layers:**

- SQL (C++) / Expression Layer:** This layer processes raw data into structured expressions. Examples include:
  - SQL (C++) marketplace\_review
  - SQL (C++) marketplace\_user
  - SQL (C++) marketplace\_seller
  - SQL (C++) marketplace\_product
  - SQL (C++) marketplace\_review
  - SQL (C++) marketplace\_interaction
  - Expression marketplace\_review\_sentiment
  - Expression marketplace\_user\_total\_orders\_placed
  - Expression marketplace\_user\_total\_product\_inquiry\_count
  - Expression marketplace\_user\_unique\_products\_inquired\_about
  - Expression marketplace\_user\_average\_rating\_given
  - Expression marketplace\_user\_review\_count
  - Expression marketplace\_product\_category\_value
  - Expression marketplace\_product\_average\_rating
  - Expression marketplace\_product\_total\_reviews
  - Expression marketplace\_product\_review\_count (90d)
  - Expression marketplace\_product\_review\_count (360d)
  - Expression marketplace\_product\_review\_count (90d)
  - Expression marketplace\_product\_review\_count (30d)
  - Expression marketplace\_product\_most\_recent\_review\_rating
- Aggregations + Windowed Aggregations (Purple Box):** This layer handles complex aggregations and windowed calculations. It includes:
  - SQL (C++) marketplace\_review
  - SQL (C++) marketplace\_user
  - SQL (C++) marketplace\_seller
  - SQL (C++) marketplace\_product
  - SQL (C++) marketplace\_review
  - SQL (C++) marketplace\_interaction
- LLM (Green Box):** The Large Language Model layer, which includes:
  - Resolver get\_marketplace\_product\_category\_value
  - Expression marketplace\_review\_is\_positive\_review\_from\_llm
  - Expression marketplace\_review\_is\_positive\_review\_online
  - Expression marketplace\_review\_reviewer\_name
- Accelerated Python (Yellow Box):** This layer uses Python for accelerated computations. Examples include:
  - Accelerated Python computed\_feature\_dependent\_another\_computed\_feature\_marketplace
  - Accelerated Python get\_marketplace\_username
  - Accelerated Python get\_normalized\_rating
  - Accelerated Python is\_positive\_review\_from\_python\_resolver
- Numpy + Math libs (Blue Box):** This layer uses Numpy and Math libraries for numerical computations. It includes:
  - Accelerated Python is\_positive\_review\_from\_python\_resolver

**Summary of Components:**

- 18 Chalk expressions
- 9 SQL resolvers
- 5 Python resolvers
- 4 Redis online stores

**Dependencies:**

- Accelerated Python (get\_normalized\_rating) depends on aggregation values.
- Accelerated Python (is\_positive\_review\_from\_python\_resolver) depends on Numpy + Math libs.

- 18 Chalk expressions (UDFs)
- 9 SQL resolvers
- 5 Python resolvers (4 UDFs)
- 4 Redis online store lookups

## Numpy + Math libs

Depends on aggregation values



# Symbolic Python Interpreter

many Python resolvers are relatively straightforward

```
1 def compute_total(  
2     subtotal: Transaction.subtotal,           # float  
3     discount: Transaction.discount,           # float | None  
4     tax_rate: Transaction.locale.sales_tax_rate, # float  
5     is_taxable: Transaction.taxable,         # bool  
6 ) → Transaction.total:  
7  
8     if discount is not None:  
9         # Apply any discount, but don't allow it to become negative.  
10        subtotal = max(subtotal - discount, 0.0)  
11  
12    if not is_taxable or tax_rate is None:  
13        return subtotal  
14  
15    return subtotal * (1.0 + tax_rate)  
16
```

# Symbolic Python Interpreter

Converts Scalar Python Resolver → Expression

- Resolvers already have a strict arrow schema, as do expressions
- **Convert at plan time**
- Avoids Python completely at query time

## Python Resolver → Expression (example)

```
if discount is not None:
    subtotal = max(subtotal - discount, 0.0)

if not is_taxable or tax_rate is None:
    return subtotal

return subtotal * (1.0 + tax_rate)
```

```
if(
  or(
    not(col("I")),
    is_null(col("T")),
  ),
  if(
    not(is_null(col("D"))),
    max(col("S") - col("D"), lit(0.0)),
    col("S"),
  ),
  if(
    not(is_null(col("D"))),
    max(col("S") - col("D"), lit(0.0)),
    col("S"),
  ) * (1.0 + col("T")),
)
```

Rewrites de-duplicate common sub-expressions

## Python Resolver → Expression (example)

SymbolicValue:

- Velox Expression + Python Type
- Initially col(...) expressions
- Immutable, side-effect free
  - Always refer to original value

*Environment*

```
discount    = col("D")/float
subtotal    = col("S")/float
is_taxable  = col("I")/float
tax_rate    = col("T")/float?
```



*Environment*

```
discount    = abc(...)
subtotal    = max(col("S") - 10.0, 0.0)
is_taxable  = ghi(...)
tax_rate    = jkl(...)
```

## Symbolic Python Interpreter

Fallback to running Python with process pools

- For unsupported functions e.g. `openai.responses.create`
- Functions with runtime-validated preconditions

“Gas Limit” estimates total amount of work (at query time + runtime) to determine whether conversion is a good idea

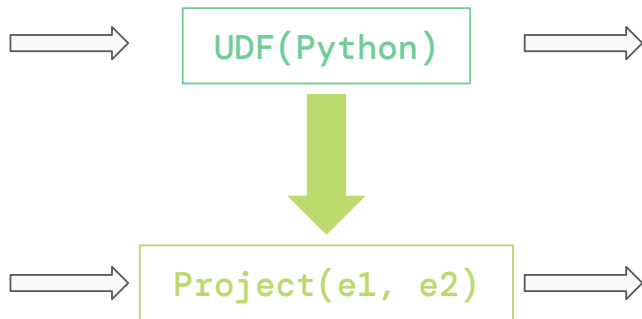
# Symbolic Python Interpreter

Python as a DSL for writing expressions!

- Can run as regular Python functions outside of Chalk
- Lift Python semantics verbatim into Velox

Benefits

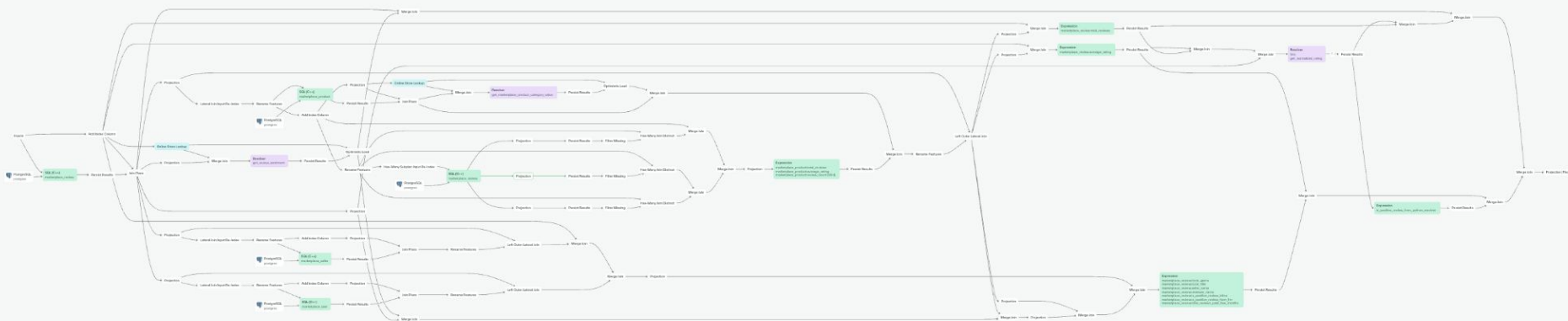
- No GIL
- No heavy per-object heap representation
- Falls back if it encounters any unhandled expression/function
- Becomes SIMD!





# Logical Plan

10+ Python Resolvers, 5 Postgres lookups, 2 Redis Online Store lookups, 68ms\* response time



# Velox Plan

