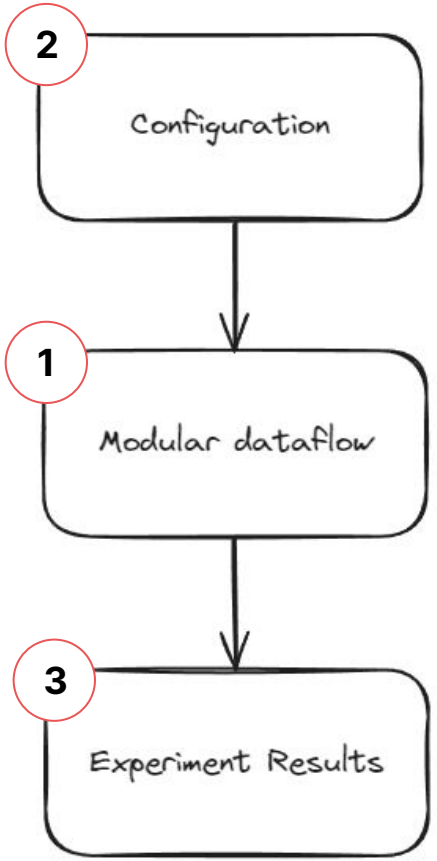
Modular Dataflows
& Experiment Management
for Machine Learning Evaluation

Modular dataflows for research

Context

- Train ML models for time series forecasting
- Benchmark 200+ models and conduct statistical tests
- Reproducibility is paramount, requiring lineage for code and data



Who Am I ?

- My name is Thierry, I'm based in Montréal, QC, Canada
- Using Hamilton since ~2021-11, it launched in 2021-10
- Working with Stefan and Elijah since 2023-06
- Previous experiences:

    teaching ML/DS, AI consulting, HR tech SaaS

Creating

Modular Dataflows

Level 0: spaghetti

```python
import numpy as np
import pandas as pd

def main():
    df = pd.read_parquet("raw_data.parquet")

    pairwise_diff = np.diff(df[["loc_x", "loc_y"]].values, axis=0)
    pairwise_dists = np.sqrt((pairwise_diff ** 2).sum(axis=1))
    df["location_jump"] = np.insert(pairwise_dists, 0, np.nan)
    timedelta = df.timestamp.diff().dt.total_seconds()
    df["location_jump_speed"] = df["location_jump"] / timedelta

    return df
```

- A single function

Limitations

- Hard-coded values
- Assumptions about columns
- Unrelated steps within same scope

Level 1: functions

```python
import numpy as np
import pandas as pd

def location_jump(df):
    pairwise_diff = np.diff(df[["loc_x", "loc_y"]].values, axis=0)
    pairwise_dists = np.sqrt((pairwise_diff ** 2).sum(axis=1))
    return np.insert(pairwise_dists, 0, np.nan)

def location_jump_speed(df):
    timedelta = df.timestamp.diff().dt.total_seconds()
    return df["location_jump"] / timedelta

def main():
    df = pd.read_parquet("raw_data.parquet")
    df["location_jump"] = location_jump(df)
    df["location_jump_speed"] = location_jump_speed(df)
    return df
```

- Functions to define features
- Passing DataFrames to add columns

Limitations

- Implicit dependencies between features
- **Imperative**: need to manually order functions in **main()**

Level 2: checking schemas

```python
import numpy as np
import pandas as pd

@check_columns(["loc_x", "loc_y"])
def location_jump(df):
    pairwise_diff = np.diff(df[["loc_x", "loc_y"]].values, axis=0)
    pairwise_dists = np.sqrt((pairwise_diff ** 2).sum(axis=1))
    return np.insert(pairwise_dists, 0, np.nan)

@check_columns(["location_jump"])
def location_jump_speed(df):
    timedelta = df.timestamp.diff().dt.total_seconds()
    return df["location_jump"] / timedelta

def main():
    df = pd.read_parquet("raw_data.parquet")
    df["location_jump"] = location_jump(df)
    df["location_jump_speed"] = location_jump_speed(df)
    return df
```

- Make dependencies explicit via custom decorator.
- Exactly what **pandera** does with **@check_input**

Limitations

- Still imperative

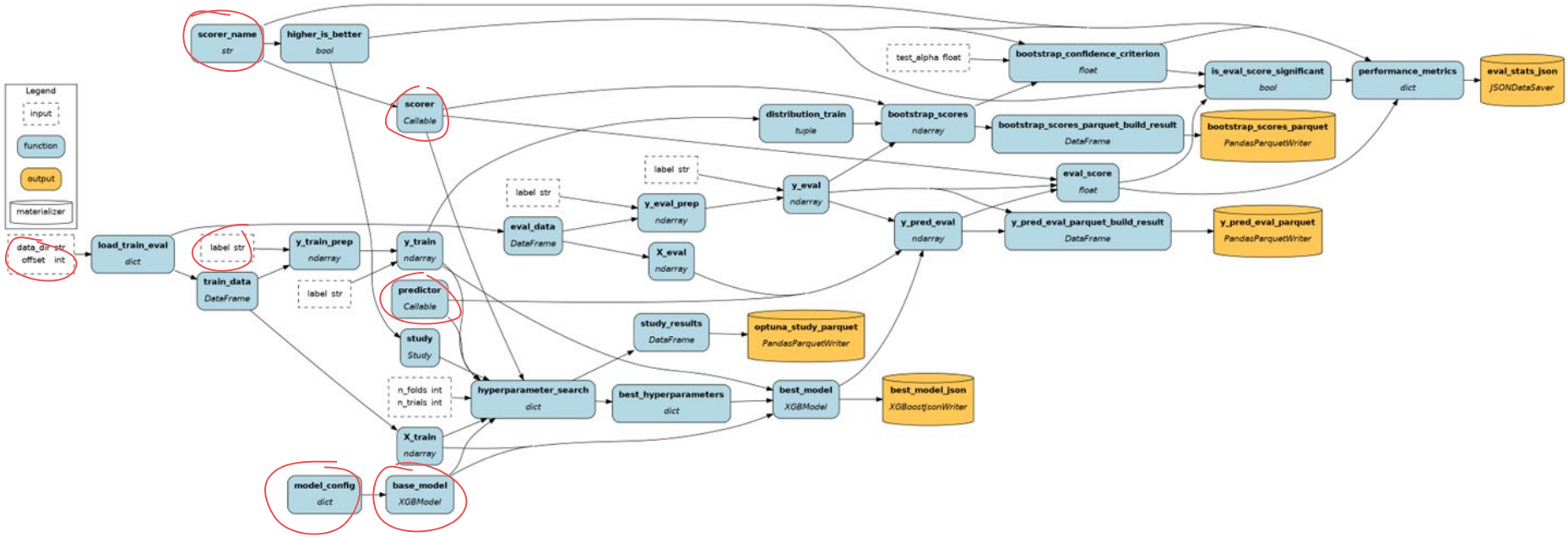
Level 3: Hamilton

```python
# features.py
import numpy as np
import pandas as pd
from hamilton.function_modifiers import extract_columns

@extract_columns("loc_x", "loc_y", "timestamp")
def raw_data(file_path: str) -> pd.DataFrame:
    return pd.read_parquet(file_path)

def location_jump(loc_x: pd.Series, loc_y: pd.Series) -> pd.Series:
    """Distance between the positions at two consecutive timestamps"""
    return ...

def location_jump_speed(
    location_jump: pd.Series,
    timestamp: pd.Series
) -> pd.Series:
    """Speed between consecutive locations"""
    return ...

# run.py
from hamilton import driver, base
import features

def main():
    dr = (
        driver.Builder()
        .with_modules(features)
        .with_adapters(base.PandasDataFrameResult())
        .build()
    )
    features = ["timestamp", "location_jump", "location_jump_speed"]
    df = dr.execute(
        features,
        inputs={"file_path": "raw_data.parquet"}
    )
```

- Forces the use of type hints and encourages docstrings
- Dependencies are explicit
- Separation between inputs and transformations
- **Declarative** script! Simply request what you want

Adding Configurability

Training an XGBoost model



Configuration dimensions

Experiment Dimension	Execution `inputs`	Driver `config`	Driver `modules`
3 forecast horizon (one dataset each)	set `offset` to compute (or select) the dataset		
2 model architecture (XGBoost, LSTM)			one dataflow (.py file) per model type
4 learning tasks (binary, multiclass, ordinal, continuous)		set `task` config to one of the 4 tasks	
10 target variables	set `label` to select a column from the dataset		
2 execution mode (development, evaluation)		set `mode` to change dataset loading behavior	

Configuration at the Driver level

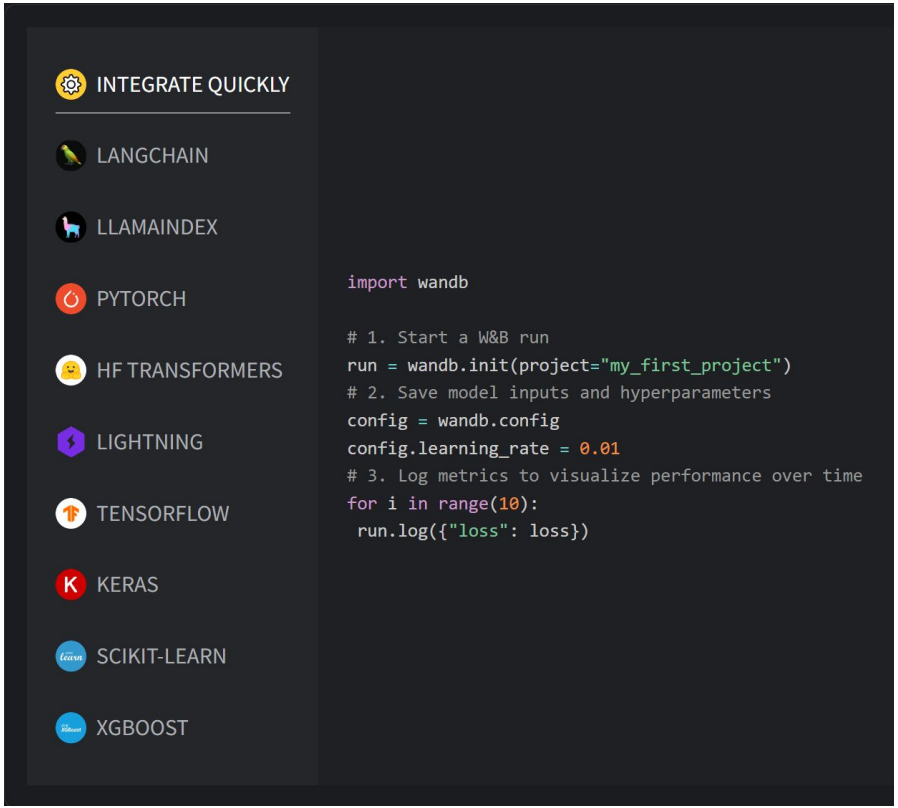
```python
from hamilton import driver
import tabular_model
import model_evaluation

def main(cfg):
    dr = (
        driver.Builder()
        .with_modules(tabular_model, model_evaluation)
        .with_config({"task": cfg.task, "mode": cfg.mode})
        .build()
    )

    inputs = dict(
        data_dir=cfg.data_dir,
        offset=cfg.offset,
        label=cfg.label,
        n_trials=cfg.n_trials,
        n_folds=cfg.n_folds,
    )
    final_vars = ["best_model", "performance_metrics"]
    dr.execute(final_vars, inputs=inputs)
```

- Pass config at the “driver level”
  - Driver **config**
  - Driver **modules**
  - **inputs**
  - **overrides**

- Breakdown large dataflows into multiple scripts
  - prepare_data.py
  - train_model.py
  - benchmark.py

Tracking Experiment Results

MLFlow and Weights&Biases



- Direct integrations with many ML libraries
- Provides a web UI to explore results
- Facilitate collaboration

Limitations

- Integrations are actually confusing
- Hard to trace the code producing an artifact
- Weak code-artifact versioning

Node level

```python
def best_model(
    X_train: np.ndarray,
    y_train: np.ndarray,
    base_model: xgboost.XGBModel,
    best_hyperparameters: dict,
    artifact_path: str,
    model_name: str,
) -> xgboost.XGBModel:
    model = base_model
    model = model.set_params(
        early_stopping_rounds=None,
        **best_hyperparameters,
    )
    model.fit(X_train, y_train)

    mlflow.xgboost.log_model(
        model,
        artifact_path=artifact_path,
        registered_model_name=model_name
    )
    return model
```

Driver level

```python
from hamilton import driver
import tabular_model
import model_evaluation

def main(cfg):
    dr = (
        driver.Builder()
        .with_modules(tabular_model, model_evaluation)
        .with_config({"task": cfg.task, "mode": cfg.mode})
        .build()
    )
    inputs = dict(
        data_dir=cfg.data_dir,
        offset=cfg.offset,
        label=cfg.label,
        n_trials=cfg.n_trials,
        n_folds=cfg.n_folds,
    )
    final_vars = ["best_model", "performance_metrics"]
    results = dr.execute(final_vars, inputs=inputs)

    best_model = results["best_model"]

    with mlflow.start_run():
        run.set_tags(inputs)
        mlflow.log_params(best_model.params)
        mlflow.xgboost.log_model(
            best_model,
            artifact_path=cfg.artifact_path,
            registered_model_name=cfg.model_name,
        )
        mlflow.log_metrics(results["performance_metrics"])
```

Building the ExperimentTracker adapter

```python
from hamilton import driver
from hamilton.io.materialization import to
from hamilton.plugins.h_experiments import ExperimentTracker
import tabular_model
import model_evaluation

def main(cfg):
    dr = (
        driver.Builder()
        .with_modules(tabular_model, model_evaluation)
        .with_config({"task": cfg.task, "mode": cfg.mode})
        .with_adapters(
            ExperimentTracker(
                experiment_name=cfg.experiment_name,
                base_directory="./experiments",
            )
        )
        .build()
    )
    inputs = dict(
        data_dir=cfg.data_dir,
        offset=cfg.offset,
        label=cfg.label,
        n_trials=cfg.n_trials,
        n_folds=cfg.n_folds,
    )
    materializers = [
        to.json(
            id="best_model_json",
            dependencies=["best_model"],
            path="./xgboost_model.json"
        ),
        to.json(
            id="eval_stats_json",
            dependencies=["performance_metrics"],
            path="./eval_stats.json"
        ),
    ]
    dr.materialize(*materializers, inputs=inputs)
```

- artifacts == Hamilton nodes
- Specify artifacts with .**materialize()**
- Artifact versioned according to executed code
- Bundled with an extensible UI

Quick

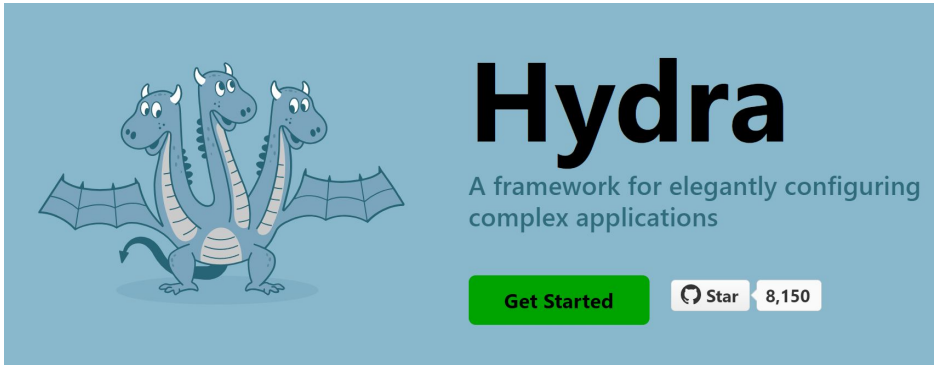
Live Demo

Extras

Extra: Project structure

- Separate “packages” from script/notebook execution
- Separate input and output data
- Breakdown your project into several dataflows

Things I've tried



- Hydra is a yaml-first config system
- Popular for deep learning research

- Metaflow is a Python orchestrator
- Easy to deploy yourself on AWS

METAFLOW

Everything you need to develop data science, ML, and AI apps