A New Generation of Pipeline and Orchestration Tools

Tooling like dlt, metaflow, hamilton and kedro require that we don't reinvent the wheel.

colum.mccoole@analect.com
colum-mccoole-746b946a
analect/technical-docs-hierarchy

Docs: Level 2 AI/ML Value-chain

Data System Layers

In the 'Modular Data Tooling' section of this document pack, The Composable Codex envisaged three layers of a data-system - data, execution and UI (expression). The DAGWork's variant is presented below, where they include an *asset* layer that is responsible for structuring code into assets that are meaningful to the business and producing them and the *orchestration* layer that triggers their computation.

- **Data:** the physical representation of data, both inputs and outputs
- **Execution:** perform data transformations
- **Expression:** the language to write data transformations
- **Asset:** an object in persistent storage that captures some understanding of the world (e.g., a dataset, a database table, an ML model, dashboards)
- **Orchestration:** operational system for the creation of assets

Layer	Example tools
Orchestration	Airflow, Metaflow, Dagster, Prefect, Temporal
Asset	Hamilton, Kedro, dbt, dlt, SQLMesh, LangChain
Expression	pandas, SQL, polars, R, Ibis,
Execution	Spark, Snowflake, DuckDB, RAPIDS
Data	S3, Postgres, Snowflake, local files

Figure 1. DAGWork's 5-layer modular data-stack

Pipeline Orchestration using DAGs

Pipeline solutions like Airflow, Dagster, Hamilton and Kedro, all use a directed-acyclic-graph (DAG) to represent a pipeline, specifying how 'assets' (a node within the DAG) relate and depend upon eachother. Imperative systems require developers to expressly tell the orchestrator 'what to do'. Declarative ones, *declare* what can be computed and each function infers its dependencies.

Imperative orchestrators (Airflow) are efficient when there are only a few tasks and the recipe is linear. When the number of tasks grows, it becomes difficult to manually specify dependencies and maintain code.

Declarative orchestrators (Hamilton, Dagster) better manage a large number of tasks and complex dependencies by automating the DAG assembly. Consequently, it favors writing smaller functions resulting in code that's easier to read, test, debug, and maintain.

Common to all tools we profile here is the ability to run and test locally, but then the possibility of running them in production in the cloud, usually without any need to adjust the code much, if at all, making the efficient to work with.

dlt (data-load-tool) - github.com/dlt-hub/dlt

dlt contains all data-engineering best practices and handles:

- **schema evolution** - `dlt` automatically infers the initial schema for your first pipeline run. As the structure of data changes, such as the addition of new columns or changing data types, `dlt` handles these schema changes
- **data contracts** - can be used to raise an exception if data is not what you expected, allowing for discards at the row and value-level.
- **incremental loading** - is a crucial concept in data pipelines that involves loading only new or changed data instead of reloading the entire dataset.
- **performance management** - `dlt` provides several mechanisms and configuration options to manage performance and scale-up pipelines, including parallel execution, thread pools and async exection, memory buffers, iterators and chunking.
- **pre-built pipeline templates** - `dlt` comes with many verfied sources and destinations, so you use a command like `dlt init github postgres` to get the skeleton of a working pipeline pulling data from github and pushing it to postgres.
- `rest_api` **source toolkit** - a declarative and customisable REST interface that uses Python dictionaries instead of `YAML` or `JSON`. This choice allows more advanced developers to inject custom functionality, such as writing their own authorization methods for an API.

Metaflow - github.com/netflix/metaflow

Metaflow is a framework for defining data science and data engineering workflows with the the ability to define local experiments and scale those experiments to production jobs from a single API. Metaflow follows the dataflow paradigm which models a program as a directed graph of operations.

In particular, Metaflow is designed to be a cloud-native framework, relying on basic compute and storage abstractions provided by all major cloud providers.

When you run a flow without special decorators, the flow runs locally on your computer like any Python script or a notebook. If your job requires more resources than what is available on your workstation, simply annotate the step with a `@resources` decorator. The most advanced pattern of compute that Metaflow supports is distributed computing using the `@parallel` decorator.

Metaflow allows you to specify software dependencies as a part of the flow. You can either use a Docker image with necessary dependenices included, or layer them on top of a generic image on the fly using `@conda` or `@pypi` decorators. You can use the image argument in `@batch` and `@kubernetes` decorators to choose a suitable image on the fly, like an official pytorch image.

Kedro - github.com/kedro-org/kedro

Kedro is an open-source Python framework to create reproducible, maintainable, and modular data science code. It uses software engineering best practices to help you build production-ready data science pipelines. Kedro borrow heavily from the Cookiecutter Data Science approach by seeding new projects with pre-configured boilerplate code. Kedro is orchestrator-agnostic.

```
project-dir     # Parent directory of the template
├── conf        # Project configuration files
│   ├── base
│   │   ├── catalog.yml    # config ref load/store assets
│   │   └── parameters.yml # params to pass to pipeline at exec.
│   └── logging.yaml
├── data        # Local project data (not committed to vc)
├── docs        # Project documentation
├── notebooks   # Project-related Jupyter notebooks
├── README.md   # Project README
├── src         # Project source code
│   └── project_name
│       ├── pipelines
│       │   └── pipeline_name
│       │       ├── nodes.py    # python func to transform assets
│       │       └── pipeline.py # form pipeline from kedro nodes
│       ├── pipeline_registry.py # pipeline discovery from CLI
│       └── settings.py # default config for hooks and pipeline
exec.
└── pyproject.toml  # Identifies the project root; has config info
```

Hamilton - github.com/dagworks-Inc/hamilton

For comparison sake, the Kedro project above would look like this in Hamilton. It is much more light-weight. Hamilton allows you to write functions that are "pure" and then compose them together into a dataflow. Hamilton separates transformation logic from execution, allowing you to seamlessly scale via remote execution (AWS, Modal, etc.) and specialized computation engines (Spark, Ray, duckdb etc.).

```
project-name/
├── src/
│   └── project_name/
│       ├── __init__.py
│       ├── dataflow.py # define data transforms in pipeline
│       └── run.py # execute the pipeline (includes param, data-
catalog)
└── requirements.txt
```

The new Hamilton Kedro plugin allows you to connect your Kedro Pipeline to the Hamilton UI for rich observability and introspection features as well as metadata capture.