Metaflow Up-for-Grabs

Fun & useful projects that don't need a deep knowledge of the internals of Metaflow to implement.

DevOps

Metaflow load testing / performance dashboard

It would be great to have a set of performance/load tests as a part of the test suit, a'la https://speed.pypy.org/. We could start by creating a test set manually.

How hard would it be to support Metaflow on Windows natively (no WSL)?

https://github.com/Netflix/metaflow/issues/60

Best practices for model hosting

Explore and document how to deploy models

- Sagemaker
- AWS Lambda + API Gateway
- ECS
- Something else

Explore executing local tasks inside lightweight containers

Today, in the local model Metaflow executed tasks as OS-native processes. It would be fun & profitable to explore and POC how we could execute even local tasks inside lightweight containers. There are a number of options (see this excellent article for an overview):

- Docker (not exactly lightweight)
- Firecracker
- Many others

As a result, we could support limiting @resources even for local processes.

Explore better monitoring options

Today, it is hard to monitor per-task resource consumption (CPU/memory/disk) etc. The default CloudWatch setup is clunky. We could explore better options

- Better CloudWatch config than the default
- DataDog
- Grafana
- Something else

Explore using Bazel with Metaflow for dependency management

Bazel could be a decent alternative to @conda or Docker for some. Even though UX is clunky, it is one of very few tools that can provide an environment for building your artifacts that is hermetic, reproducible and crossplatform. There is a good number of major companies using Bazel today, and a few startups around it that promise to make it easier.

Integrations

How to visualize Metaflow runs with Streamlit

Process data, train models in Metaflow, provide a dashboard in Streamlit

VSCode plugin for Metaflow

Provide a DAG visualization, artifact visualization, remote code execution etc. all inside VSCode. Even just having a nice "Run Flow" button could go a long way, not everyone is into CLI tools.

Data

How to use Metaflow with dbt (and/or Snowflake)

Explore and document best practices

How to use Metaflow with Great Expectations

Explore and document best practices

How to use Metaflow with Spark/EMR/AWS Data Wrangler

Use an existing Python library (e.g. AWS Data Wrangler) to execute e.g. SparkSQL queries, fetch results using metaflow.S3. Explore and document best practices.
https://github.com/Netflix/metaflow/issues/335

Modeling

Metaflow with HuggingFace

Explore and document best practices for NLP

Metaflow with TensorBoard

Explore and document best practices for training a model with Metaflow, monitoring the process with TensorBoard

Core Metaflow

Support for other linters than PyLint, e.g. Black

https://github.com/Netflix/metaflow/issues/218

Custom decorator for memory/CPU profiling

https://github.com/Netflix/metaflow/issues/432

Custom @notify decorator to send notifications by email/Slack

https://github.com/Netflix/metaflow/issues/443

Metaflow explain

A subcommand that renders a web page, shows it in a local web browser, that visualizes a DAG of the current flow, information about decorator, syntax-highlighted source code etc. Consider it a fancier version of the “show” subcommand.

Artifact visualization in a notebook

Imagine a function that you can use in a notebook to quickly visualize all data produced by a task. It takes a Metaflow Task object as an argument

```
visualize(Run('HelloFlow/343')['start'].task)
```

Iterates over all data artifacts produced by the task, sniffs their type, and displays them accordingly: Dataframes as dataframes, Python native types as usual etc. A quick way to eyeball the results of a run!

Easier debugging with @debug

If a task is slow or gets stuck on @batch or another remote compute layer, it can be hard to figure out what's going on. We could introduce a @debug decorator that adds plenty of profiling in the background (what processes are running, how much memory is consumed etc) and prints the results to stdout, as well as injects debug prints in the user code to highlight what lines are being executed.

If a task gets stuck, it can be interrupted and run again with enhanced debugging using `resume --with debug`. See this Slack thread for a motivating use case.