Using Dataflow for local ML batch inference

Lessons Learned

Agenda

Who We Are

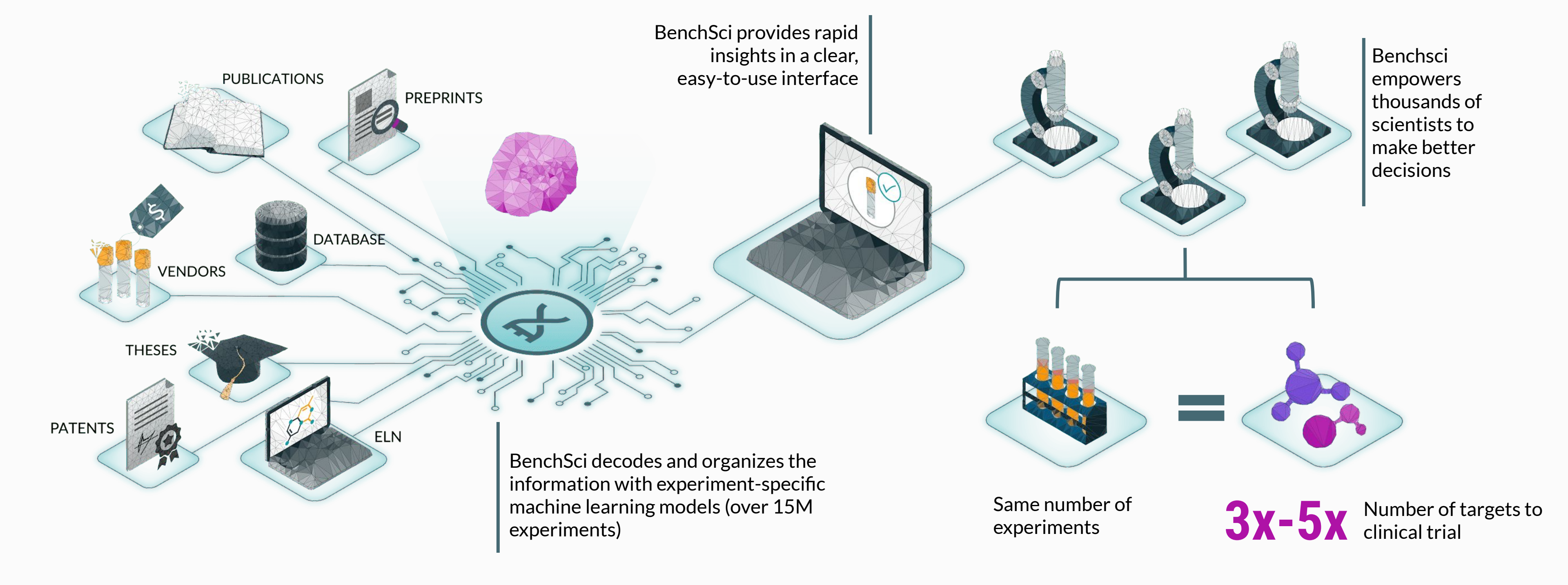
Inference on Dataflow

Heavy Initialization & Memory Management

Multiple Models

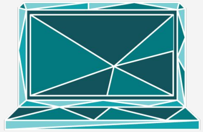
Introduction

AI-Assisted Preclinical Experimental Design platform



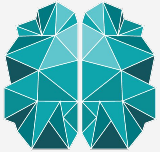
Our approach to decode the world's experiments follows a 3 step process: **collect, extract, and contextualize**

1 Data Collection & Curation



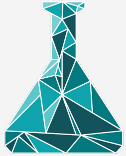
- Documents containing experimental data
- Reagents
- Bioinformatics

2 Extraction



- Biomedical Entity Recognition Models
- Computer Vision
- ETL pipelines

3 Contextualization



- Context tagger models
- Data QA
- Conversion to Platform

Tech Hierarchy

1. **Beam & DataFlow Runner**
   Data mining pipelines

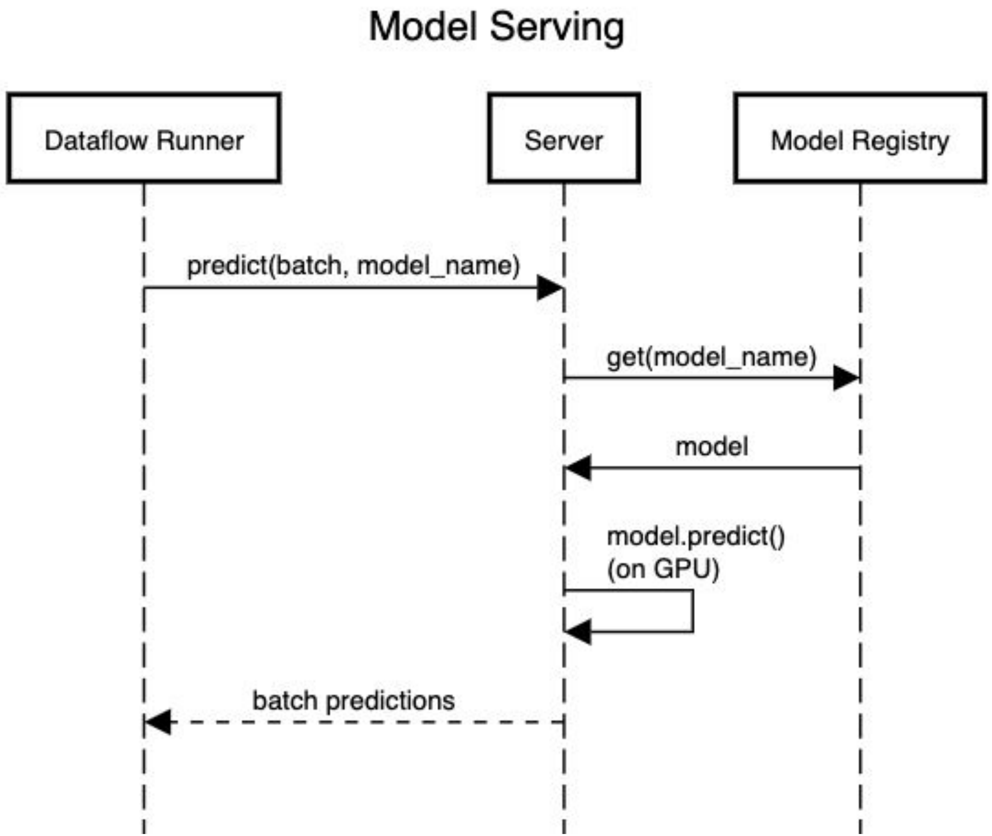
2. **BigQuery**
   Data Warehouse for our parsed papers and ML datasets

3. **Colocation Servers (replaced by inference on Dataflow)**
   Where Inference happens (using Beam's external services pattern)

Before: Colocation Servers

Dataflow as Client
"Pattern: Calling external services for data enrichment"

Colo as the Server
The API that receives the requests and serves the models.



Colocation Servers: Pain Points

Scalability

- Our models had to run in sequence in order for colo to handle them properly.
- Our ML inference step, our biggest bottleneck in our full run pipelines then, would soon made our jobs take more than 24 hours.
- Couldn’t increase/decrease our resources depending on the models and its hardware requirements

Maintainability

- Many manual steps involved
- Deploying a new model took on average ~ 3 weeks

Reproducibility

- Custom scripts and manual efforts to setup and run the pipeline

Inference on Dataflow

Common Patterns

Remote Inference

Serialized data is sent to an API endpoint

- Separation of concerns
- Managed training services

Local Inference

Initialization + Internal Call to inference Code

- Integration with preprocessing pipelines
- Low Latency and CPU utilization
- Using Beam features

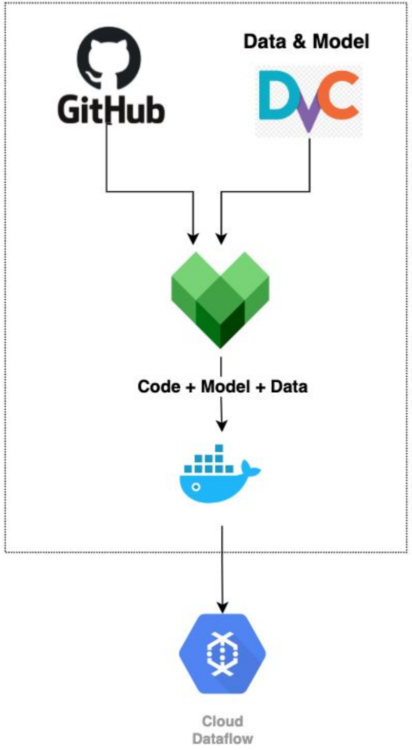
Inference with Dataflow Runner

Custom Containers

- Wrapping code dependencies (and model artifacts.)
- Customizing the execution environment (GPU libraries, ...).
- Copy over the necessary artifacts from a default Apache Beam base image

GPUs

- Nvidia Drivers
- Count & Type per VM
    - --experiment "worker_accelerator=...”



After: Inference on Dataflow

Cloud Dataflow Runner

- Machine Type: n1-standard-4
- GPU: NVIDIA T4
- Autoscaling up to 500 GPUs
- Takes < 6 hours to do inference on ~100M rows on 14 Deep Learning models including two SciBERT models in parallel

CI/CD

- Fully reproducible inference jobs; re-running inference on the exact same environment
- End-to-end runs from scratch using a single trigger
- Automated E2E testing on DataFlow on every PR

Heavy Initialization & Memory Management

- Input Management
- Shared Model
- Worker Parallelism Control

Input Management

Reduce the initialization overhead by batching and understand the input rows before calling inference

✓ Batching

- BatchElements(min_batch_size,, max_batch_size, …)
- GroupIntoBatches
- CombineFn
- …

✓ Sorting Inputs & Dynamic Padding

- More deterministic as we sort our sentences by sentence lengths, with the largest sentences be predicted first.
- A bit more efficient as we now batch on similar-sized inputs.

Shared Model

“Shared class for managing a single instance of an object shared by multiple threads within the same process.“

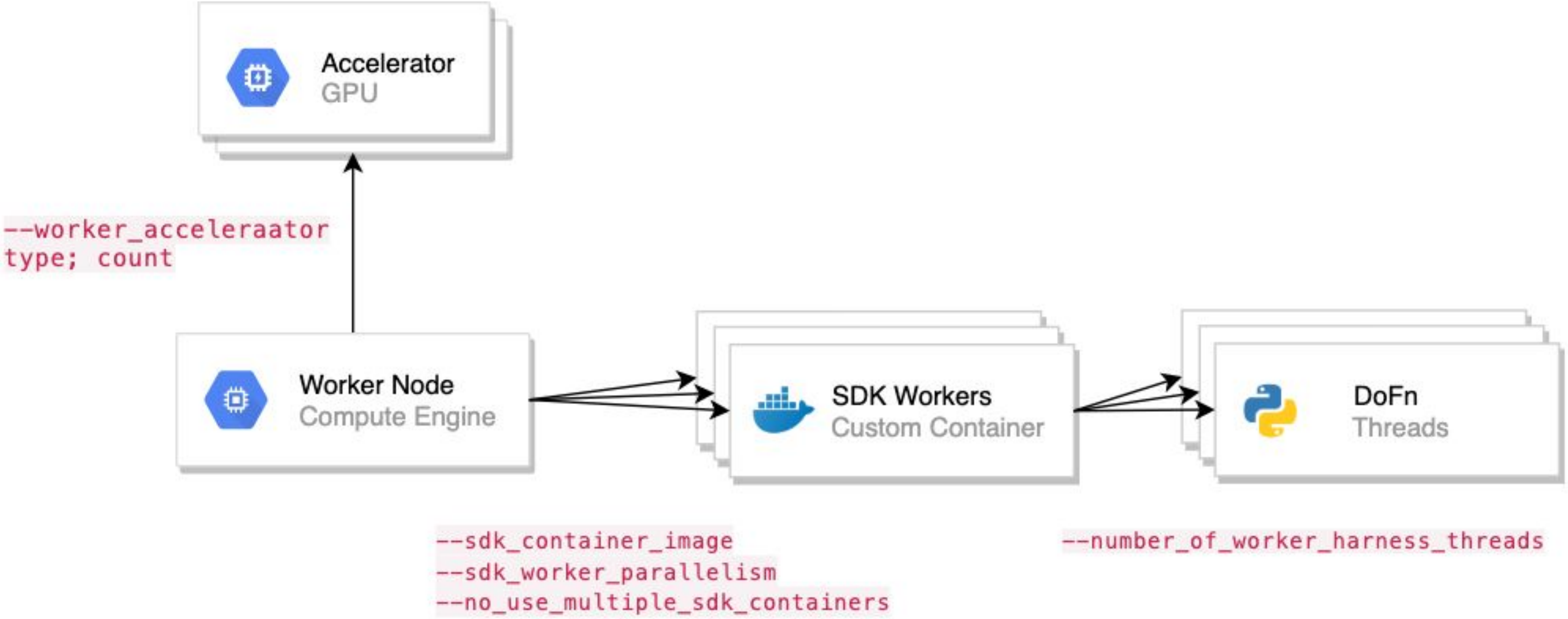
- ✓ Shared by all threads of each worker process.
- ✓ Doesn’t save you from OOM without Worker Thread Control
- ✓ Use @setup DoFn *even* if you don’t use Shared
- ✓ More important if model artifacts are not wrapped into the container at build time and are downloaded from Model Registry at run time

```
def setup(self):
    def load_model():
        return MyModelLoader.load(self.model_name)

    self.model = self._shared_handle.acquire(load_model,
                                             tag=self.model_name)

def process(self, batch, *args, **kwargs):
    for predicted in self.model.predict(batch):
        yield predicted
```

Worker Parallelism Control



Multiple Models

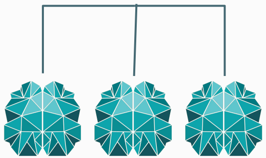
- Pipeline Branches vs. Independent Pipelines

Pipeline Branches vs. Independent Pipelines

Or a hybrid approach

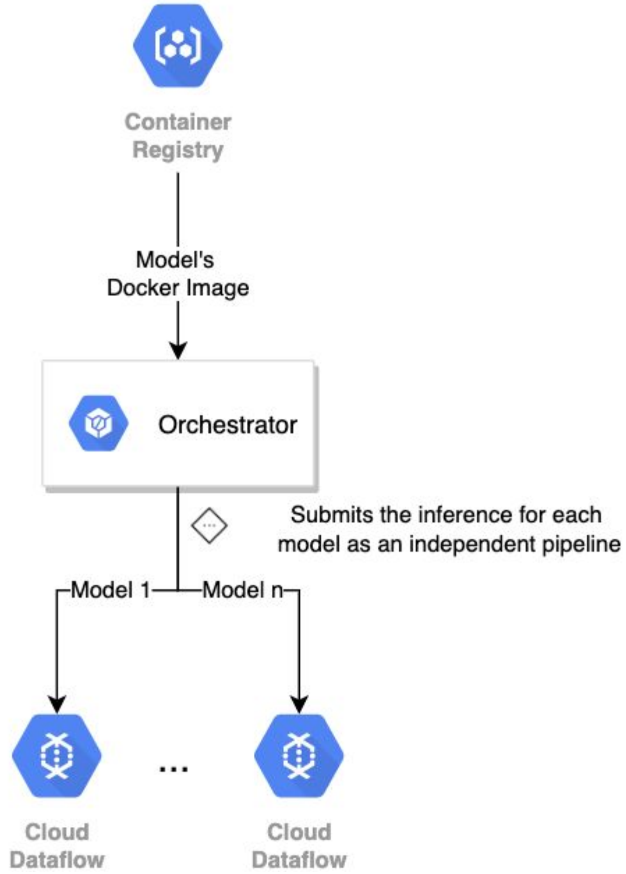
One Pipeline

- *Branches* vs. *Sequential* Inference
  - Keep an eye on memory management
- Good for coupled models with same:
  - Architecture
  - Input
  - Dependencies
  - Hardware Configuration



Independent Pipelines

- More flexibility in terms of configuration:
  - Model-specific container
  - GPU resources
  - Disk Space
- No inter-pipeline resource management



What Next?

- Optimization for Worker Parallelism
- Managing independent pipelines
- Multi-SDK, Multi-container Support
- Defining hardware configuration per PTransform

Summary

- Inference on Dataflow as a Feasible Option
- Heavy Initialization
- Memory Management
- Worker Parallelism

Q&A