

DESIGNING A PURE-PYTHON FRAMEWORK FOR DECLARATIVE DATA PIPELINE CONSTRUCTION

Santosh Kumar Maddali

Staff Data Engineer

ABSTRACT:

Most production data pipelines are written twice: the work itself is ordinary Python, and the order it runs in is written again as configuration, as graph objects wired by hand, or as operator declarations. The two copies are checked at different times by different tools, so they drift apart. This paper presents a pure-Python framework that removes the second copy. A pipeline is written as a plain Python function. The framework runs that function with stand-in values instead of real ones, watches which steps get called and what gets passed between them, and recovers the dependency graph from the code itself. The recovered graph is checked before any step runs, using the type annotations already present in the step signatures, and every step is given a fingerprint so that unchanged work is skipped on later runs. The framework uses only the Python standard library, with no compiled extensions, so a pipeline definition can be loaded and checked anywhere Python runs. A 457-line implementation was measured on pipelines of up to 4,993 steps. Building and checking a graph cost about 7 and 4 microseconds per step and both scale linearly, running a step through the framework adds 0.58 microseconds, a badly typed pipeline is rejected 121 times faster than it would fail during execution, fingerprinting reduced a repetitive 641-step graph to 13 distinct pieces of work, and the parallel executor reached a 22-fold speedup on a workload dominated by waiting. The checking usually gained by moving pipeline structure outside Python turns out to be available inside it, at a cost that is small in the workloads measured here.

Keywords:

data pipelines, declarative programming, dataflow, Python, static checking, fingerprinting, incremental computation, workflow orchestration

I. INTRODUCTION

A data pipeline is a set of steps that transform data, where some steps have to run before others. Almost every tool in this space agrees on the same shape: a graph of steps, an engine that runs them in dependency order, and ordinary code inside each step. What the tools do not agree on is where the graph comes from.

Most of them ask the engineer to write it out. Some use a builder interface, where you create step objects and connect them by hand. Some use a separate configuration file. Some use operator classes wired together in a script. In all three cases the pipeline exists twice: once as the code that does the work, and once as the description of how that code fits together.

Writing it twice costs something real. Rename a parameter, change a return type, or swap the order of two steps, and the description has to be updated to match. Nothing in the language forces that update to happen, and nothing checks it. The mistake shows up at run time, often after expensive earlier steps have already run and already written data somewhere. The problem is not that scheduling is hard. The problem is that the description of the pipeline is not checked the way the code is checked.

There is another option: let the tool work out the graph from the code, instead of asking a person to write it down. Machine learning libraries have done this for years, under the name tracing. The user writes a normal function. The library calls it with stand-in values, records every operation those stand-ins pass through, and ends up with a graph it can then check, optimize, and run [1-3]. Very little of this has carried over to data engineering, where a step is a table load or a join rather than a matrix operation, and where the things worth checking early are types and arguments rather than tensor shapes.

This paper asks what a framework like that should look like for data engineering work, under one hard constraint: pure Python, standard library only, no compiled extensions, and no external services. The constraint is not arbitrary. Teams often need a pipeline definition to be importable somewhere awkward: a locked-down build agent, a serverless runtime, a notebook sandbox, or a linter. A definition that can be read and checked anywhere Python runs is worth more than one that is slightly faster but only loads inside one specific system.

A. Contributions

The paper makes four contributions.

- 1) A way of writing pipelines in which the pipeline is an ordinary Python function and the graph is recovered from it rather than declared. Loops, conditionals, comprehensions, and helper functions all work normally, because they run while the graph is being recorded (Section IV).
- 2) A graph record and a set of checks that reuse the type annotations already sitting in the step signatures, catching incompatible types between connected steps, missing arguments, and unknown arguments before any step runs (Section V).
- 3) A fingerprint for each step, computed from its code, its arguments, and the fingerprints of the steps feeding it. One mechanism gives both incremental re-runs and the removal of repeated work (Section VI).
- 4) A 457-line implementation with measurements on graphs of up to 4,993 steps, covering build cost, per-step overhead, memory, the value of early checking, and parallel scaling (Section VIII).

II. RELATED WORK

A. Building a Graph From Code

Treating a computation as a graph of steps connected by data is an old idea. Kahn set out the semantics [4], and Johnston et al. survey the languages built on it [5]. Batch systems used the idea for parallelism: MapReduce fixed the graph at two stages [6], Dryad allowed any shape [7], Spark added recovery by replay [8], and later systems added query optimization [9] and merged batch with streaming [10], [11].

Two are close to the present design. FlumeJava lets the programmer write plain Java against collection objects; the library builds a graph, optimizes it, and only then runs it [12]. Dask does something similar in Python, offering array and dataframe objects whose operations build a task graph for a scheduler [13]. Both show a graph can be a side effect of writing normal code, but in both the deferral is attached to a particular collection type. The framework here defers at the level of any user function, so a node is whatever the engineer decided a step should be.

B. Orchestrators and Build Tools

Orchestrators care about scheduling, retries, and visibility rather than parallel data processing. Scientific workflow systems set the pattern [14], [15], and later tools describe dependencies in a rule file inspired by Make [16-18]. Luigi asks the author to declare each task class and what it requires [19], and Airflow's original operator interface has the author wire task objects together explicitly [20].

The closest prior systems are Airflow's TaskFlow interface [21] and Dagster's functional pipeline model [22]. Both recover data dependencies from compositions of decorated Python functions: a decorated function becomes a step, calling it returns a handle rather than a value, and passing that handle into another step records the dependency. Dagster goes further than TaskFlow and attaches types to those dependencies. The recording mechanism in Section IV is the same idea, so what is claimed here is not function-style graph construction. It is a dependency-free compilation stage that freezes the recovered graph, validates declared type compatibility before execution, and content-addresses nodes for reuse and deduplication.

The differences are specific. TaskFlow infers dependencies from XCom-backed outputs and moves values through them, and Airflow validates graph structure while parsing the definition file, cycles included, but does not use return and parameter annotations to check compatibility across an edge as Section V does. Dagster's types are enforced largely as the pipeline runs. Neither loads without its orchestrator, whereas the checks here run on a frozen graph before the first step, using only the standard library, so a definition can be checked in a linter or a pull request with no database or scheduler present.

Build tools face the same problem and have a sharper answer. Mokhov et al. split a build tool into a scheduler and a part deciding what needs rebuilding, and show that hashing the inputs gives correct minimal rebuilds [23]. Nix hashes everything an artifact depends on to get reproducible results [24], and self-adjusting computation explains how a small change propagates [25]. The design borrows that discipline: a step's fingerprint plays the same role as a build tool's cache key.

C. Deferred Execution in Python

Machine learning libraries made tracing popular as a way to get a graph out of Python. TensorFlow first required an explicit graph and later traced ordinary functions [1], PyTorch recorded operations as they happened [2], and JAX transforms traced functions [3]. Ray defers whole tasks and actors across machines [26]. Because these libraries trace numerical operations, what they check is shapes and device placement; the same trick at data engineering granularity recovers a different and, here, more useful set of facts.

Python's optional type annotations [27], [28] make those facts free to obtain, since an annotated step already says what it takes and returns. Building a small language inside a host language rather than inventing syntax is long standing [29], and is the approach here.

D. Checking Pipelines

What happens when pipeline structure goes unchecked is well documented. Sculley et al. name tangled pipelines and configuration sprawl as leading maintenance costs [30]. Production platforms answered with checking components [31], [32] and with tracking of what produced what [33], and provenance research supplies the vocabulary [34]. All of this checks the data moving through a pipeline; this framework checks the shape of the pipeline itself, before any data is touched, so the two are complementary.

TABLE I *Where the pipeline graph comes from, and what can be checked before anything runs.*

Approach	Where the graph comes from	What can be checked early
Configuration file	Written by hand, separately	The file format only
Task classes [19]	Declared requirements per task	Whether names exist
Rule files [16-18]	Declared file patterns	Whether paths connect
Deferred collections [12], [13]	Operations on typed collections	A fixed set of operators
Function recording [21], [22]	Recovered from the pipeline body	Structure and cycles at parse time, but no annotation based edge check
Tracing function (this work)	Recovered from the pipeline body	Types, arguments, cycles

Table I places the design among the alternatives. What sets it apart is not that the graph is recovered from the code, since the row above does that too. It is what happens next: the recovered graph is frozen, checked against the annotations already in the source, and content-addressed, all before the first step runs and all without loading an orchestrator.

III. DESIGN GOALS

Five goals shaped the design. Later sections refer back to them.

G1. One copy. The structure of the pipeline must come from the code that does the work. There must be no second description that can fall out of step with the first.

G2. Errors before effects. Anything that can be caught without running a step should be reported before the first step runs. That means wrong types between steps, missing arguments, unknown arguments, and cycles.

G3. Plain Python. An engineer should be able to use loops, conditionals, comprehensions, and helper functions when assembling a pipeline, and should be able to test a step by calling it directly. No new syntax, and no operator tricks that confuse a debugger.

G4. No dependencies. The part that defines and checks a pipeline must use only the standard library, so a definition can be imported and checked anywhere Python runs.

G5. Running is a separate decision. Scheduling, parallelism, caching, and retries should be chosen after the graph exists, without touching the definition. The definition says what is computed; the executor decides how.

IV. HOW A PIPELINE IS WRITTEN

A. Two Decorators

Fig. 1 summarises the whole path from decorated functions to results. This section covers its left hand end.

The whole interface is two decorators. @task turns a plain function into a pipeline step and records its signature and type annotations. @pipeline marks a function whose body says how the steps fit together. Listing 1 shows a complete example.

Listing 1. A pipeline written as an ordinary Python function. The body is never run for its values. It is watched.

```
@task
def load(path: str) -> Table:
    return Table.read_parquet(path)

@task
def normalize(t: Table, tz: str = "UTC") -> Table:
    return t.with_timestamps(tz)

@task
def join(left: Table, right: Table) -> Table:
    return left.join(right, on="entity_id")

@pipeline
def daily_rollup(events: str, entities: str):
    facts = normalize(load(events), tz="UTC")
    return join(facts, load(entities))

graph = daily_rollup.compile() # build + check
ThreadedExecutor(width=8).run(
    graph, events="file:///e/", entities="file:///n/")
```

The body of daily_rollup reads as ordinary Python and can be understood as such. It never runs for its values. Calling compile() gives each pipeline parameter a stand-in value and then calls the body with those stand-ins. Every call to a decorated step notices that recording is switched on, writes down a node instead of doing the work, and hands back a handle. A handle carries nothing but a node number, so passing it to the next step is what records an edge.

B. What Comes With Recording

Because the body really does run while the graph is recorded, normal Python helps rather than gets in the way. A for loop over source tables becomes that many parallel branches, an if on a configuration flag picks a branch while recording and leaves nothing in the graph, and a helper that assembles a common piece is just called. This is how G3 is met without the framework implementing any control flow of its own.

The matching restriction is that a decision cannot depend on a value the pipeline itself computes, since that value does not exist while recording, a common limitation of tracing-based systems [3]. In practice pipeline shape is decided by configuration and by the catalog rather than by the contents of rows, so the limit rarely bites; Section X covers the cases where it does.

Outside recording a decorated step is just a function, so testing one needs no framework, harness, or scheduler.

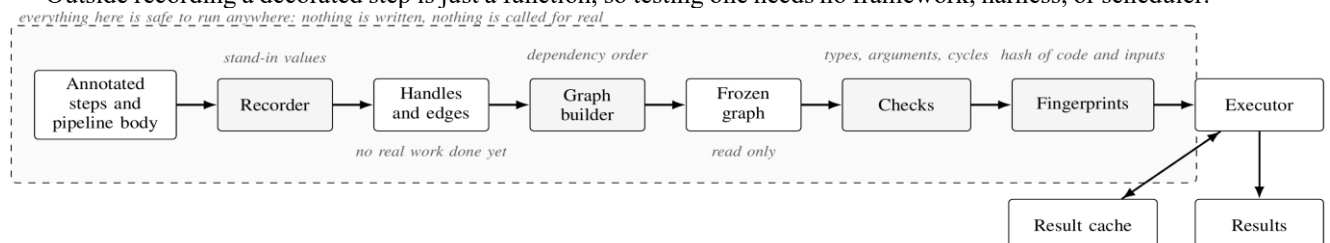


Fig. 1. How the framework works end to end. Everything to the left of the executor is safe and dependency free: the pipeline body is watched, turned into a frozen graph, checked, and fingerprinted. Only the executor touches the outside world, which is why the whole build and check stage can run inside a linter, a build agent, or any other locked-down environment.

V. BUILDING AND CHECKING THE GRAPH

A. The Graph Record

Recording produces handles and edges in the order the steps were called. The builder turns that into a frozen record: one entry per node plus a dependency order. Each entry holds a node number, the step name, whether it is an input or a step, and the function to call. It also records the plain values given to the step, which of its parameters are fed by which upstream nodes, and the types declared going in and coming out. Entries are read only once built, so the graph can be shared between threads and cached without copying.

A handle can only point at a node that already exists, so an edge always points backwards and a recorded graph cannot contain a cycle. Algorithm 1 gives the procedure. The dependency sort is kept anyway, because graphs stitched together from separately recorded pieces are not covered by that argument, and a cycle check is cheap.

Algorithm 1 Recovering the graph from a pipeline function

```

1: input: a pipeline function and its parameters
2: start with an empty graph
3: for all parameters of the pipeline do
4:   add an input node, and make a stand-in handle for it
5: end for
6: switch recording on for the current thread
7: call the pipeline body, passing the stand-in handles
8:   {steps write themselves down instead of running}
9: switch recording off
10: for all step calls that were recorded do
11:   split the arguments into plain values and handles
12:   add a node, with one incoming edge per handle
13: end for
14: freeze the nodes and sort them into dependency order
15: reject if the sort cannot reach every node, since that means a cycle
16: run the checks described below
17: return the graph

```

B. What Gets Checked

Checking walks the graph in dependency order and enforces three things per node, which is what G2 asks for.

Arguments. Every parameter without a default has to be filled, either by a plain value or by an upstream step. Any argument naming a parameter the step does not have is rejected. These are the mistakes a builder interface leaves until the scheduler tries to call the function.

Types. For every edge, the framework compares the type the upstream step says it returns against the type the downstream step says it wants. The edge is allowed if the first is the same as the second, or a subclass of it. Two cases are always allowed. If either side is unannotated or declared as Any, the edge passes, which is the usual convention for optional annotations in Python [27], [28]. If both sides are generic types such as list or dict, only the outer type is compared, so a step returning list[int] satisfies a step asking for list.

Only edges are checked. A literal written straight into a call, such as tz=123 where the parameter is annotated str, is not compared against its annotation, and Union, Optional, protocols, and type variables are admitted without analysis. Checking literals is easy to add and left for later.

Being lenient here is deliberate. A checker that rejects a valid pipeline is worse than one that occasionally lets a bad one through, because the first stops work while the second just leaves you where you already were.

Cycles. Handled by the dependency sort described above.

None of this needs credentials, a network, or data. That is what makes it usable in a linter, a pre-commit hook, or a pull request check, and it follows directly from G4.

VI. FINGERPRINTS AND EXECUTION

A. What a Fingerprint Is

Every node gets a fingerprint: a SHA-256 hash over four things. The first is the node's kind and step name. The second is a hash of the step's source text. The third is the plain values passed to it. The fourth is the fingerprints of the steps feeding into it. Because upstream fingerprints are folded in, a node's fingerprint stands for its declared dependencies: its own code, the values written into the call, and everything upstream of it. It does not stand for

ambient state. A step that reads a module-level constant, calls a helper defined elsewhere, consults an environment variable, or depends on an installed library version can change behaviour with its fingerprint unchanged; Section X returns to this. Plain values become bytes through repr, stable for the strings, numbers, and simple containers that configuration consists of, but not for an object falling back on the default repr, since that embeds a memory address. A production implementation would need a declared encoding. Working the fingerprints out for a whole graph costs one pass, since each is remembered once computed.

This is the same idea a build tool uses to decide what needs rebuilding [23], [24], applied to pipeline steps. Two useful things follow, and the evaluation measures both.

Re-running only what changed. Edit one step and its source hash changes, so its fingerprint changes, so the fingerprints of everything downstream of it change too. Nothing else moves, so every other node keeps its key and comes back from the cache. The set that gets recomputed is the smallest correct one, as long as steps are honest functions of their declared inputs.

Repeated work done once. Two nodes running the same code on the same values with the same upstream fingerprints get the same key, so the second one comes back from the cache without ever running. Repeated work disappears with no optimizer pass involved, purely because the key describes contents rather than position in the graph.

The assumption about honest functions is the price. A step that reads a table someone else is writing to produces a result its fingerprint does not describe. Caching is therefore something the executor turns on rather than a default, in line with G5: the pipeline definition is the same either way.

B. Executors

Two are provided. The sequential executor walks the dependency order and calls each node with its arguments filled in. The parallel executor counts how many inputs each node still awaits, sends any node whose count reaches zero to a worker pool, and decrements its successors as results arrive. This picks up all the parallelism the graph contains without the author describing any of it, which is the payoff of recovering the graph rather than settling for a script that runs top to bottom.

The workers are threads, so parallelism helps work that spends its time waiting, such as input/output-bound extract and load steps [35]. Steps that are busy rather than waiting need a process pool. The same interface allows that, since the graph can be pickled whenever its steps are defined at module level, but the current prototype does not include one.

VII. IMPLEMENTATION

The implementation is 457 lines of Python in one module, with nothing imported from outside the standard library. It uses inspect for signatures, typing.get_type_hints for annotations, hashlib for fingerprints, dataclasses for the node entries, threading for the recording switch, concurrent.futures for the worker pool, and importlib to restore step functions when a graph is unpickled.

The recording switch is per thread, so pipelines can be built at the same time in a multithreaded service without interfering. Source hashes are computed once when a step is decorated and kept on the function, so fingerprinting a large graph does not keep going back to disk. Where the source cannot be read, the digest falls back to a value unique to that step, so two dynamically built steps that happen to share a name can never collide on one cache key and return each other's cached result.

VIII. EVALUATION

A. Setup

Everything was measured on a single virtual CPU of an Intel Xeon at 2.10 GHz with 3 GiB of memory, running Python 3.10.6 on Linux. Figures are medians of seven runs unless stated otherwise, with standard deviations given for the main series. No warm-up run is thrown away. What follows measures the framework itself rather than a production pipeline, and is a microbenchmark study.

The test pipelines are built from a common source feeding several parallel chains, which are then combined back together in pairs, so the graphs are both wide and deep. Five questions are asked.

RQ1. How does the cost of building a graph grow with its size?

RQ2. How much does the framework add to each step compared with calling the functions directly?

RQ3. What does checking early save compared with finding the same problem at run time?

RQ4. How much work do fingerprints avoid?

RQ5. How well does the parallel executor use the graph?

B. RQ1: Build Cost

Table II gives the cost of recording, checking, and fingerprinting graphs from 17 to 4,993 nodes. Fig. 2 plots the same numbers against a straight line.

TABLE II Cost of building a graph, by size. Each column is a separately measured median of seven runs, so *Compile*, which records and checks in a single call, is not the arithmetic sum of the *Record* and *Check* columns; *Hash* is a further pass over the finished graph. The per-node cost barely moves across two and a half orders of magnitude.

Nodes	Record (ms)	Check (ms)	Hash (ms)	Compile (ms)	Record (μ s/node)
17	0.149	0.081	0.035	0.269	8.78
49	0.359	0.200	0.070	0.527	7.32
97	0.610	0.366	0.131	0.947	6.28
497	3.070	1.854	0.674	4.624	6.18
993	6.231	4.085	1.336	10.447	6.27
1,993	13.339	7.274	2.633	23.007	6.69
4,993	34.830	18.047	6.782	50.094	6.98

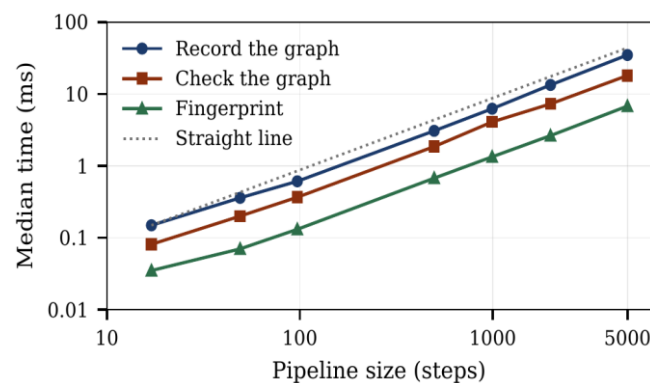


Fig. 2. The cost of building a graph grows in a straight line with its size, in all three stages. The dotted line extends the smallest measurement assuming perfectly straight growth.

All three stages grow in a straight line. Once past a hundred nodes, recording settles between 6.2 and 7.0 microseconds per node, checking between 3.6 and 4.1, and fingerprinting around 1.4. A 5,000-node pipeline is recorded and checked in 50 milliseconds and fingerprinted in a further 7, which fits inside a linter or a pre-commit hook. The graph for 4,993 nodes needs 2.44 MiB, or 512 bytes per node, so keeping several large definitions in memory is not a concern.

C. RQ2: Cost Per Step

A 993-node graph of trivial steps was run through the sequential executor and compared with hand-written Python making the same sequence of calls. The framework took 0.708 ms, with a standard deviation of 0.140 ms, against 0.136 ms for the direct version. That works out to 0.58 microseconds added per step.

The ratio of 5 to 1 comes from steps that do almost nothing, and it is the wrong way to look at it for this kind of work, where steps spend their time on input, output, and queries. Against a step that takes one millisecond, 0.58 microseconds is 0.06 percent. Against a step that takes a second, it does not register. The absolute number is the one that matters, and in these workloads the framework is far from the bottleneck once steps do real work.

D. RQ3: The Value of Checking Early

A 202-step pipeline was built with a type error at the last edge, where a step returning an integer feeds a step expecting a string. Each earlier step was given a one millisecond cost to stand in for modest input and output. Checking rejected the pipeline in 1.84 ms. Running it hit the same problem after 222.74 ms, by which point 201 steps had run and their results were thrown away. That is 121 times more work wasted.

The multiplier grows with both the length of the pipeline and the cost of each step, and those are exactly the numbers that grow in production. The measurement also undersells the benefit: in a real pipeline the discarded work is not just lost time but visible effects, such as half-written tables and used-up API quota, so catching the problem first leaves nothing to clean up.

E. RQ4: What Fingerprints Save

Two effects were measured separately.

Re-running only what changed. A 321-node graph of 64 branches four steps deep, each step doing a fixed amount of work, was run once to fill the cache. One value in one branch was then changed and the pipeline rebuilt. Of 320 steps, 10 ran again: the edited step and everything downstream of it, a recompute fraction of 3.1 percent. Wall clock time fell from 11.40 ms, with a standard deviation of 1.05 ms, to 5.39 ms, an improvement of 2.1 times. The gap between running 32 times fewer steps and finishing 2.1 times faster is the cost of rebuilding and looking things up in the cache, which caching does not remove and which dominates when steps are cheap. With steps of realistic cost the two ratios converge.

Repeated work done once. A 641-node graph of 128 identical branches turned out to hold only 13 distinct fingerprints, a reduction of 98.0 percent, and running it called 12 step bodies. Work a graph keyed by position would do 128 times gets done once. This was not designed in; it falls out of how the fingerprint is defined, and is strongest on the repetitive, machine-generated graphs that templated pipeline code produces.

F. RQ5: Parallel Scaling

Thirty-two independent steps, each waiting 10 ms, were run at executor widths from 1 to 32. Fig. 3 plots the speedup against the sequential baseline of 325.0 ms.

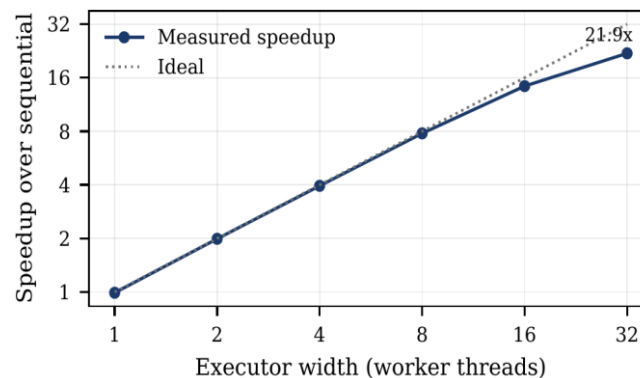


Fig. 3. Speedup of the parallel executor on work that spends its time waiting. It tracks the ideal line closely up to width 16 and reaches 22 times at width 32, on a single virtual CPU, because a waiting step is not using it.

The speedup stays close to ideal: 1.99 at width 2, 3.95 at width 4, 7.79 at width 8, 14.36 at width 16, and 21.95 at width 32, where the cost of handing out work starts to show against a 10 ms step. Efficiency stays above 89 percent through width 16. This holds on a single virtual CPU because a step that is waiting is not using it, which makes threads an appropriate default for input/output-bound steps. Nothing in the pipeline described this parallelism. It came out of the recovered graph.

G. What These Numbers Do Not Show

The test pipelines are synthetic and isolate the framework's own cost rather than imitating a production pipeline, so the measured per-step overhead is an upper bound on the share belonging to the framework. Everything ran on one machine with one virtual CPU, so absolute timings will differ elsewhere, though the straight-line growth and the comparisons between cases should not. The result on repeated work depends on the graph being repetitive, so it describes templated pipelines rather than arbitrary ones. Finally, there is no comparison against a production orchestrator. Such a comparison would mix the cost of defining a pipeline with the cost of a scheduler, a database, and serialization, none of which this design pays, and building a fair test for it is left for later.

IX. DISCUSSION

The measurements support one narrow claim well. The checking usually obtained by moving pipeline structure into a separate file or a builder interface is available inside Python, at a cost of microseconds per node to build the graph and under a microsecond per step to run it, with no compiled extension or external service.

Three decisions carried most of the weight. First, deferring at the level of any user function rather than a collection type lets the graph match how the engineer already split up the problem, which is the main structural difference from earlier deferred execution work in Python [12], [13]. Second, reusing type annotations instead of

inventing a schema language means the strength of the checking follows a habit the team already has, and no second artifact appears, which is what G1 asks for. Third, keeping the build and check stage free of side effects lets it run where execution is impossible, which is where structural mistakes are cheapest to fix.

The link to build tools is worth stating plainly. Once results are keyed by fingerprint, running a pipeline is a build and the executor decides what to rebuild [23]. Re-running only what changed, and doing repeated work once, become consequences of the key rather than features to write, which is a much smaller thing to get right.

X. LIMITATIONS AND FUTURE WORK

Shape that depends on data. Graph shape is fixed while recording, so a pipeline whose branches depend on a value it computes cannot be written directly. The usual answers apply: lift the decision into a pipeline parameter, or add an explicit branch node with both sides recorded. The second is planned.

How deep the type checking goes. Checking compares declared types and is lenient about generics and missing annotations, ignoring protocols, type variables, and what is inside a generic. Handing the edge check to an external type checker while keeping graph recovery here is a natural next step.

Outside state. Fingerprints describe a step's own source and its declared arguments, not everything it can reach. A step reading a table someone else is changing is not keyed correctly, nor is one depending on a module level constant, a helper defined elsewhere, an environment variable, or a library version. Letting a step declare its external inputs, and folding a table version or a watermark into the fingerprint, would close that gap and is the most valuable thing to do next.

Running across machines. The graph pickles cleanly, provided its steps are defined at module level, and the executor is separate from it, so a process pool or a distributed back end along the lines of Ray [26] needs only a new executor, with no change to how pipelines are written. That is the point of G5.

Evidence from real pipelines. The biggest gap is a study of production pipelines measuring how many defects these checks would catch before deployment. That, and a fair comparison with existing orchestrators, are the main items of future work.

XI. CONCLUSION

Pipeline structure is usually written twice and checked once. This paper argued that the second writing is unnecessary. A graph of steps can be recovered from an ordinary Python function by running it with stand-in values, checked against the type annotations already in the code, and fingerprinted so that unchanged work is not repeated. A 457-line implementation with no dependencies outside the standard library records, checks, and fingerprints a 5,000-node pipeline in 57 milliseconds, adds 0.58 microseconds per step, rejects a badly typed pipeline 121 times faster than running it would find the same fault, reduces a repetitive 641-node graph to 13 distinct pieces of work, and reaches a 22-fold speedup on waiting-bound work with nothing written down about parallelism. The checking that seems to require leaving Python does not, and the price of keeping it inside is small in the workloads measured here.

REFERENCES

- [1] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, 2016, pp. 265-283.
- [2] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Vancouver, BC, Canada, 2019, pp. 8024-8035.
- [3] J. Bradbury *et al.*, "JAX: Composable transformations of Python and NumPy programs," 2018. [Online]. Available: <https://github.com/google/jax>. Accessed: Oct. 4, 2021.
- [4] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congress*, Stockholm, Sweden, 1974, pp. 471-475.
- [5] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1-34, Mar. 2004.
- [6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107-113, Jan. 2008.
- [7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys European Conf. Computer Systems*, Lisbon, Portugal, 2007, pp. 59-72.

- [8] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2012, pp. 15-28.
- [9] M. Armbrust *et al.*, "Spark SQL: Relational data processing in Spark," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Melbourne, Australia, 2015, pp. 1383-1394.
- [10] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792-1803, Aug. 2015.
- [11] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Systems Principles (SOSP)*, Farmington, PA, USA, 2013, pp. 439-455.
- [12] C. Chambers *et al.*, "FlumeJava: Easy, efficient data-parallel pipelines," in *Proc. 31st ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, Toronto, ON, Canada, 2010, pp. 363-375.
- [13] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proc. 14th Python in Science Conf. (SciPy)*, Austin, TX, USA, 2015, pp. 126-132.
- [14] B. Ludäscher *et al.*, "Scientific workflow management and the Kepler system," *Concurrency Computat.: Pract. Exper.*, vol. 18, no. 10, pp. 1039-1065, Aug. 2006.
- [15] E. Deelman *et al.*, "Pegasus, a workflow management system for science automation," *Future Gener. Comput. Syst.*, vol. 46, pp. 17-35, May 2015.
- [16] J. Köster and S. Rahmann, "Snakemake: A scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520-2522, Oct. 2012.
- [17] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnol.*, vol. 35, no. 4, pp. 316-319, Apr. 2017.
- [18] S. I. Feldman, "Make: A program for maintaining computer programs," *Softw.: Pract. Exper.*, vol. 9, no. 4, pp. 255-265, Apr. 1979.
- [19] Spotify, "Luigi: A Python module for building complex pipelines of batch jobs," 2021. [Online]. Available: <https://github.com/spotify/luigi>. Accessed: Nov. 15, 2021.
- [20] Apache Software Foundation, "Apache Airflow documentation," 2021. [Online]. Available: <https://airflow.apache.org/docs/>. Accessed: Nov. 15, 2021.
- [21] Apache Software Foundation, "Airflow TaskFlow API tutorial," 2021. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html. Accessed: Nov. 15, 2021.
- [22] Elementl, "Dagster: An orchestration platform for the development, production, and observation of data assets," 2021. [Online]. Available: <https://github.com/dagster-io/dagster>. Accessed: Nov. 15, 2021.
- [23] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Art. 79, pp. 1-29, Jul. 2018.
- [24] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proc. 18th USENIX Conf. System Administration (LISA)*, Atlanta, GA, USA, 2004, pp. 79-92.
- [25] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*, Portland, OR, USA, 2002, pp. 247-259.
- [26] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *Proc. 13th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, USA, 2018, pp. 561-577.
- [27] G. van Rossum, J. Lehtosalo, and Ł. Langa, "PEP 484: Type hints," Python Software Foundation, Sep. 2014. [Online]. Available: <https://peps.python.org/pep-0484/>. Accessed: Oct. 4, 2021.
- [28] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *Proc. 7th Workshop Scheme and Functional Programming*, Portland, OR, USA, 2006, pp. 81-92.
- [29] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, Art. 196, Dec. 1996.
- [30] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28*, Montreal, QC, Canada, 2015, pp. 2503-2511.
- [31] D. Baylor *et al.*, "TFX: A TensorFlow-based production-scale machine learning platform," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Halifax, NS, Canada, 2017, pp. 1387-1395.
- [32] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data validation for machine learning," in *Proc. 2nd SysML Conf.*, Stanford, CA, USA, 2019.

IJETRM

INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)

<https://ijetrm.com/>

- [33] M. Zaharia *et al.*, "Accelerating the machine learning lifecycle with MLflow," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39-45, Dec. 2018.
- [34] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM SIGMOD Record*, vol. 34, no. 3, pp. 31-36, Sep. 2005.
- [35] D. Beazley, "Understanding the Python GIL," in *Proc. PyCon*, Atlanta, GA, USA, Feb. 2010.