dagprofiler

DAG Task Standard Specification

A standard framework for authoring profiler-compatible DAG (Directed Acyclic Graph) tasks with automatic compute and communication profiling.

Yoonjae Hwang, Bhaskar Krishnamachari
ANRG — Autonomous Networks Research Group
Viterbi School of Engineering
University of Southern California
Technical Report, February 12, 2026

Table of Contents

1. Introduction

dagprofiler is a standalone, pip-installable Python package that defines the directed acyclic graph (DAG) Task Standard — a formal contract between application developers, the profiler, and schedulers. It is a generalized, domain-independent package.

The package provides four core capabilities:

- **Automatic compute profiling** — measures per-task execution time and estimates instruction counts
- **Communication measurement** — captures per-edge data transfer volume in bytes and bits
- **Reproducible profiles** — stores execution profiles in a portable JSON format with full metadata
- **Scheduler independence** — works with any scheduler (SAGA, Kubernetes, Ray, etc.) and any network type

Any application that can be expressed as a DAG of computation tasks can use dagprofiler to profile its execution characteristics and feed the results into scheduling algorithms.

2. Package Structure

The package follows standard Python packaging conventions using pyproject.toml and hatchling as the build system:

```
dagprofiler/
├── pyproject.toml                # Package metadata and build config
├── dagprofiler/
│   ├── __init__.py              # Public API exports
│   ├── task.py                  # Generic Task ABC (base class)
│   ├── config.py                # Dynamic configuration container
│   ├── dag.py                   # DAG executor with profiling
│   ├── metrics.py               # TaskMetrics, EdgeMetrics dataclasses
│   ├── profile.py               # Profile storage/loading (JSON)
│   ├── schema.py                # JSON schema definitions
│   └── decorators.py            # @task decorator for simple usage
└── tests/                       # Comprehensive test suite
```

Module	Purpose
`task.py`	Abstract base class for all DAG tasks
`config.py`	Dynamic configuration container (key-value parameters)
`dag.py`	DAG executor that orchestrates task execution and profiling
`metrics.py`	TaskMetrics, EdgeMetrics, DAGMetrics dataclasses
`profile.py`	Profile class for save/load/compare operations
`schema.py`	JSON schema for profile validation
`decorators.py`	@task() decorator for function-based task definition

3. The DAG Task Standard

The DAG Task Standard defines three core requirements that any task must satisfy to be compatible with dagprofiler. These requirements form the contract between application developers and the profiling/scheduling infrastructure.

3.1 Requirement 1: Declare I/O Schema

Every task must declare its input fields, output fields, and configuration dependencies as class-level attributes:

```
class MyTask(Task):
    inputs = ["data_in"]          # Expected input field names
    outputs = ["data_out"]         # Produced output field names
    config_deps = ["N", "scale"]   # Config parameters used in estimation
```

These declarations enable automatic edge routing, validation, and documentation. The DAG executor uses them to determine which outputs from upstream tasks should be passed as inputs to downstream tasks.

3.2 Requirement 2: Implement the Compute Interface

Tasks must implement a compute() method that receives deserialized inputs and a configuration object, and returns a dictionary of named outputs:

```
def compute(self, data_in: np.ndarray, cfg: Config) -> Dict[str, Any]:
    result = process(data_in)
    return {"data_out": result}
```

The method signature uses keyword arguments matching the declared inputs. The return dictionary must contain all declared outputs. Serialization and deserialization are handled automatically by the framework.

3.3 Requirement 3: Provide a Cost Model (Optional)

Tasks may optionally provide a static method that returns a parameterized instruction count estimate:

```
@staticmethod
def estimate_instructions(cfg: Config) -> int:
    """Return estimated instruction count based on config."""
    return cfg.N * 100  # O(N) cost model
```

If provided, these estimates are included in the profile as node weights, enabling schedulers to make informed placement and partitioning decisions without running the actual computation.

4. Core Module Specifications

4.1 task.py — Task Abstract Base Class

The Task class is the foundational abstract base class. All DAG tasks inherit from it. Subclasses define the following:

Attribute / Method	Description
`inputs: List[str]`	Expected input field names
`outputs: List[str]`	Produced output field names
`config_deps: List[str]`	Config parameters used in estimation
`compute(**kwargs, cfg)`	The user-defined computation
`estimate_instructions(cfg)`	Optional static cost model returning instruction count
`serialize(data) -> bytes`	Default: pickle (HIGHEST_PROTOCOL)
`deserialize(data) -> Any`	Default: pickle.loads()

Task.run() Execution Pipeline

When Task.run() is called, the following steps execute in sequence:

1. Measure bytes in — count total size of serialized input data
2. Deserialize inputs — convert bytes to native objects (time measured)

3. Execute compute() — run the user-defined computation (time measured)
4. Validate outputs — ensure all declared output fields are present in the return dict
5. Serialize outputs — convert native objects to bytes (time measured)
6. Measure bytes out — count total size of serialized output data
7. Return result dict containing "outputs" (serialized), "outputs_raw" (native), and "metrics" (TaskMetrics)

4.2 config.py — Dynamic Configuration

The Config class is a dynamic container that holds arbitrary key-value configuration parameters. It supports multiple access patterns:

```
cfg = Config(N_UE=10000, N_CELL=7, N_SLICE=5)

cfg.N_UE                  # Attribute access
cfg["N_CELL"]             # Dictionary access
cfg.get("N_SLICE", 5)     # With default value
cfg.to_dict()             # Export as plain dict
cfg.copy(N_UE=20000)      # Create modified copy
```

Config objects are immutable after construction. The copy() method returns a new Config with the specified overrides applied, leaving the original unchanged.

4.3 dag.py — DAG Executor

The DAG class orchestrates the execution of multiple tasks connected by directed edges. It handles topological ordering, data routing, and metric collection.

Constructor Parameters:

- `tasks: Dict[str, Task]` — named task instances
- `edges: List[Tuple]` — (source, target) or (source, target, [fields])
- `config: Config` — shared configuration object

DAG.run(seed=None) Execution Steps:

1. Topological sort using Kahn's algorithm (with cycle detection)
2. Edge field resolution — auto-detect which outputs route to which inputs by matching field names
3. Sequential task execution in topological order
4. For each task: compute edge sizes, route inputs, execute task, capture TaskMetrics, store raw outputs for downstream use
5. Aggregate all TaskMetrics and EdgeMetrics into DAGMetrics
6. Return a Profile object containing the complete execution record

4.4 metrics.py — Metrics Dataclasses

TaskMetrics — captures per-task execution characteristics:

Field	Description
`task: str`	Task name identifier
`compute_time_ms: float`	User code execution time in milliseconds
`serialize_time_ms: float`	Output serialization time in milliseconds
`deserialize_time_ms: float`	Input deserialization time in milliseconds

Field	Description
`bytes_in: int`	Total input data size in bytes
`bytes_out: int`	Total output data size in bytes

EdgeMetrics — captures per-edge data transfer:

Field	Description
`source: str`	Source task name
`target: str`	Target task name
`data_fields: List[str]`	Output fields transferred on this edge
`size_bytes: int`	Actual bytes transferred
`size_bits: int`	Auto-calculated as bytes × 8

DAGMetrics — aggregate metrics for the full DAG execution:

Field	Description
`task_metrics: List[TaskMetrics]`	All per-task metrics
`edge_metrics: List[EdgeMetrics]`	All per-edge metrics
`node_weights: Dict[str, int]`	Task name → instruction estimate
`edge_weights: Dict[str, int]`	"src->dst" → bits transferred
`total_time_ms: float`	Wall-clock execution time

4.5 profile.py — Profile Storage

The Profile class encapsulates a complete DAG execution record and provides persistence operations:

- `save(path: str)` — write the profile to a JSON file
- `Profile.load(path: str)` — class method to load a profile from JSON
- `compare(other: Profile)` — compare two profiles and return a diff of metrics

Profiles are designed to be portable, version-tagged, and self-describing. They include the dagprofiler version, seed, hostname, and timestamp to enable exact reproducibility.

4.6 decorators.py — @task Decorator

For simple tasks, the @task decorator provides a function-based alternative to subclassing:

```
@task(inputs=["data"], outputs=["result"], estimate_fn=lambda cfg: cfg.N * 10)
def process(data, cfg):
    return {"result": data * 2}
```

The decorator automatically creates a Task subclass with the appropriate declarations and wraps the function as the compute() method.

4.7 schema.py — JSON Schema

Defines the JSON Schema used to validate profile files. This ensures that profiles produced by different versions of dagprofiler or different implementations remain structurally compatible and can be consumed by any downstream scheduler.

5. Profile JSON Format

The profile output format is scheduler-agnostic and designed for maximum portability. Every profile contains complete metadata, configuration, DAG structure, weights, and execution metrics:

```json
{
  "metadata": {
    "timestamp": "2025-01-30T17:45:00Z",
    "dagprofiler_version": "0.1.0",
    "seed": 42,
    "hostname": "worker-01"
  },
  "configuration": {
    "N_UE": 10000,
    "N_CELL": 7,
    "N_SLICE": 5
  },
  "dag_structure": {
    "nodes": ["T0", "T1", "T2"],
    "edges": [
      {"source": "T0", "target": "T1"},
      {"source": "T0", "target": "T2"}
    ],
    "execution_order": ["T0", "T1", "T2"]
  },
  "node_weights": {
    "T0": 3760000,
    "T1": 54150
  },
  "edge_weights": {
    "T0->T1": 4400000,
    "T0->T2": 560000
  },
  "execution_metrics": [
    {
      "task": "T0",
      "compute_time_ms": 5.627,
      "serialize_time_ms": 0.129,
      "deserialize_time_ms": 0.0,
      "bytes_in": 0,
      "bytes_out": 600395
    }
  ],
  "total_time_ms": 45.2
}
```

Profile Sections:

- **metadata** — timestamp, dagprofiler version, random seed, and hostname for reproducibility
- **configuration** — all Config key-value pairs used during execution
- **dag_structure** — node list, edge list, and the computed execution order
- **node_weights** — per-task instruction estimates (from estimate_instructions)
- **edge_weights** — per-edge communication volume in bits
- **execution_metrics** — per-task timing and data size measurements
- **total_time_ms** — wall-clock time for the entire DAG execution

6. Key Algorithms

6.1 Topological Sort

The DAG executor uses Kahn's algorithm to compute a deterministic execution order. The algorithm:

- Initializes a queue with all nodes having zero in-degree
- Repeatedly removes a node from the queue, appends it to the order, and decrements the in-degree of its successors
- If any nodes remain with non-zero in-degree after the queue is exhausted, a cycle exists and a ValueError is raised

This guarantees that every task executes only after all of its upstream dependencies have completed.

6.2 Edge Field Resolution

When edges are specified without explicit field lists, the executor auto-detects which outputs should route to which inputs:

- For each edge (source, target), inspect source.outputs and target.inputs
- Compute the intersection of output names and input names
- If the intersection is non-empty, route only the matching fields
- If no names match, fall back to routing all source outputs

Explicit field lists in the edge tuple override this automatic resolution, giving developers full control when needed.

6.3 Byte Counting

The byte counting algorithm measures the serialized data size for each value in a data dictionary:

```
@staticmethod
def _count_bytes(data: Dict[str, Any]) -> int:
    total = 0
    for value in data.values():
        if isinstance(value, bytes):
            total += len(value)
        elif isinstance(value, np.ndarray):
            total += value.nbytes
    return total
```

For bytes objects, the length is counted directly. For numpy arrays, the nbytes property gives the exact memory footprint. This ensures accurate communication volume measurements.

7. Serialization Strategy

dagprofiler uses a pluggable serialization system with sensible defaults:

- **Default format:** Python pickle using pickle.HIGHEST_PROTOCOL for maximum efficiency
- **Customizable:** Override Task.serialize() and Task.deserialize() for custom formats such as protobuf, msgpack, or Arrow IPC
- **Numpy support:** Automatically measures nbytes for numpy arrays, providing accurate memory footprint data

The serialization boundary is important: all data entering and leaving a task passes through serialize/deserialize. This means the profiler measures the true communication cost, not just in-memory object sizes.

8. Design Principles

dagprofiler is built on seven core design principles:

1. Declarative over Procedural — I/O declarations eliminate manual routing logic and enable automatic validation
2. Generic not Domain-Specific — works for any DAG in any domain, not tied to 5G or networking
3. Automatic Profiling — metrics are collected transparently without modifying application code
4. Reproducible — profiles include seed, config, timestamp, and hostname for exact replay
5. Scheduler-Agnostic — output format is a portable JSON consumable by any scheduler
6. Extensible — custom serialization, cost models, and task types can be plugged in
7. Testable — comprehensive test suite validates all components independently

9. Integration Example: SliceScheduler Migration

The following example illustrates the migration from the old hardcoded task pattern to the DAG Task Standard.

9.1 Before: Hardcoded Task Pattern

```
class T0CollectState(Task):
    def _deserialize(self, inputs):
        ...
    def _compute(self, data):
        ...
    def _serialize(self, outputs):
        ...
```

In this pattern, each task manually implements serialization, deserialization, and routing. There are no I/O declarations, no automatic profiling, and no cost models.

9.2 After: DAG Task Standard

```
class T0CollectState(Task):
    inputs = []
    outputs = ["Q", "CQI", "SINR", "cell", "slice", "features"]
    config_deps = ["N_UE", "N_CELL", "N_SLICE", "K"]

    @staticmethod
    def estimate_instructions(cfg: Config) -> int:
        N_UE = cfg.N_UE
        return N_UE * 5 + N_UE * 5 * 5 + N_UE * 3

    def compute(self, cfg=None):
        # Generate simulated network state
        Q = generate_queue_lengths(cfg.N_UE)
        CQI = generate_cqi(cfg.N_UE, cfg.N_CELL)
        SINR = generate_sinr(cfg.N_UE)
        return {
            "Q": Q, "CQI": CQI, "SINR": SINR,
            "cell": assign_cells(cfg),
            "slice": assign_slices(cfg),
            "features": build_features(cfg)
        }
```

The migrated version is declarative, automatically profiled, and includes a parameterized cost model. The framework handles all serialization, routing, and metric collection.

10. Testing Strategy

dagprofiler includes a comprehensive test suite covering all components:

1. Task creation and validation — verify that I/O declarations are enforced
2. Task execution with metrics — confirm that run() produces correct TaskMetrics
3. Instruction estimation — validate cost model integration
4. DAG topology — test topological sort correctness and cycle detection
5. Edge resolution and routing — verify automatic and explicit field routing
6. Profile save/load/compare — test JSON serialization round-trip and diff
7. Decorator-based task definition — ensure @task produces valid Task subclasses

11. Deployment

dagprofiler follows standard Python packaging practices:

- **Build:** `python -m build` (uses hatchling as the build backend)
- **Publish:** `twine upload dist/*` (upload to PyPI)
- **Install:** `pip install dagprofiler`
- **Dependency:** SliceScheduler declares `dependencies = ["dagprofiler>=0.1.0"]` in its pyproject.toml

The package has minimal dependencies (numpy for array support) and is compatible with Python 3.9+.