ChatGPT

# Apache Hamilton (Incubating) – Advanced Features Reference

This document is a technical reference for expert Python developers and AI agents using **Apache Hamilton** (incubating). It covers advanced features for constructing, configuring, and deploying Hamilton DAGs. Topics include **function modifiers**, dynamic DAG patterns, materialization, caching/versioning, execution backends, lifecycle hooks, LLM integration (and Apache Burr), I/O adapters, result builders, and developer tooling (UI, CLI, VSCode/LSP).

## Advanced Function Modifiers

Hamilton's **function modifiers** are Python decorators (prefixed `@`) that extend node behavior or create new nodes from a single function definition [1] [2]. Multiple modifiers can stack (order-insensitive for Hamilton's, but be cautious mixing with non-Hamilton decorators [3]). Below are key advanced modifiers, grouped by purpose:

### Metadata and Schema Annotations

- `@tag` – Attaches metadata tags to a node (does not alter computation) [4] [5]. Useful for categorizing nodes (e.g. owner, sensitivity) and later filtering or custom styling. *Example:* `@tag(owner="Data-Science", pii="False")` to tag a node with owner and PII status [1].
- **Query by Tag** – After building a Driver, use `driver.list_available_variables()` and filter by `node.tags` to select nodes by tag (e.g. execute only final outputs tagged `data_product="final"`) [6].
- `@schema` – Attaches lightweight type metadata to DataFrame outputs. Define expected schema via `@schema.output((col, type), ...)` to document column types [7] [8]. This currently only adds metadata (no validation) [9]. *Example:* `@schema.output(("a","int"), ("b","float"))` on a function returning a DataFrame adds schema info for columns *a* and *b* [10].

### Data Validation

- `@check_output` – Validates a node's return value against specified constraints (type, value range, null count, etc.) [11]. Failed checks can either log a warning (`importance="warn"`) or raise an error (`importance="fail"`) [11]. For example, `@check_output(data_type=np.int32, range=(0,100), importance="warn")` logs if output is not an int32 in [0,100] [11]. Multiple built-in validators exist (e.g. `allow_nans=False`, `min`, `max`, `between`, etc.) [11]. **Note:** Each `@check_output` adds an implicit `_raw` node in the DAG to represent the validated output [12] (you can hide these in visualizations by tagging them with `hamilton.data_quality.source_node` and styling as invisible [13]).
- **Custom Validators:** Use `@check_output_custom(*validators)` to apply user-defined validation classes (subclassing `BaseDefaultValidator`). Multiple custom validators can be provided in one decorator [14].

- **Pandera & Pydantic Integration:** If installed, you can pass a **Pandera** schema to `@check_output(schema=<DataFrameSchema>)` to validate DataFrame/Series outputs [15]. Similarly, Hamilton supports validating outputs as **Pydantic** models [16].
- `@cache` – Marks a function's result to be cached on disk for reuse in future runs. Accepts `format="<type>"` to specify storage format (e.g., `"parquet"`, `"json"`, `"csv"`, etc.) [17] [18]. By default, results are pickled; using a format leverages Hamilton's DataLoader/DataSaver under the hood (same supported formats as for materialization plugins) [19] [20]. Example: `@cache(format="parquet")` on a function will store its output as a Parquet file in the cache directory [18].

### Splitting Outputs into Multiple Nodes

- `@unpack_fields` – Splits a tuple return value into multiple output nodes [21] [22]. The decorator arguments are the names of the fields. The decorated function should return a tuple of at least that many elements. For example, `@unpack_fields("X_train","X_val","X_test") def split_data(X) -> Tuple[np.ndarray,...]` yields three separate nodes `X_train, X_val, X_test` while still allowing the original `split_data` node to produce the full tuple [23].
- `@extract_fields` – Similar to above, but for dict outputs [24] [25]. Provide either a dict of `{field: Type}` or a list of field names. The function must return a dict containing at least those keys. This can work with generic `dict`, `TypedDict`, etc. (If a TypedDict return type is used, you can omit explicit field names to extract all keys) [26] [27]. Each specified field becomes its own node, and the original function node can still return the full dict [28].
- `@extract_columns` – A specialized version of extract_fields for pandas/polars/Spark DataFrames [29]. Specify column names to extract, and the function should return a DataFrame containing those columns. Each column becomes a separate Series node [29] [30]. This provides column-level lineage and can save memory by avoiding passing large DataFrames between nodes [29].

*Note:* To add metadata to nodes created via extraction, you can stack `@tag_output` (works like @tag) to tag the split-out nodes [31].

### Creating Multiple Nodes from One Function

- `@parameterize` – Generates **multiple nodes** from one function, with different input bindings [32]. Define the new node names and a dict of dependencies for each. Use `hamilton.function_modifiers.value(val)` to insert a constant and `source(node_name)` to use an existing node as an input [33] [34]. Optionally, the docstring of the function can include placeholders (like `{param}`) that will be formatted with the specific parameter values for each generated node [33]. **Example:**

```
from hamilton.function_modifiers import parameterize, source, value
@parameterize(
    revenue_by_age     = {"df": source("df"), "groupby_col": value("age")},
    revenue_by_country = {"df": source("df"), "groupby_col":
value("country")},
    revenue_by_occupation = {"df": source("df"), "groupby_col":
value("occupation")},
)
```

```
def population_metrics(df: pd.DataFrame, groupby_col: str) -> dict:
    """Compute metrics grouped by {groupby_col}"""
    return df.groupby(groupby_col)
["revenue"].agg(["mean","min","max"]).to_dict()
```

This creates three nodes (`revenue_by_age`, `revenue_by_country`, `revenue_by_occupation`) using the same logic but different groupby column inputs [34] [35]. You can mix constants and dataflow sources as shown above. - **Simplified Parameterization:** If you only need one kind of substitution, use shortcuts: - `@parameterize_values(**params)` – like `@parameterize` but when all differing inputs are constant values. - `@parameterize_sources(**params)` – similar, when all differing inputs are existing node sources. - `@parameterize_extract_columns` – for creating multiple nodes from one DataFrame-returning function, each extracting a different column. Combines parameterization with `@extract_columns` internally. - `@does(func)` – *Function replacement.* Runs an external function over all inputs of the decorated function [36] [37]. The `func` passed must accept **kwargs (so it can handle any number of inputs uniformly)** [36] [38]**. Essentially,** `@does(f)` **makes the decorated function a thin wrapper that feeds its parameters into** `f`**. For example, given** `def sum_series(**series: pd.Series) -> pd.Series: ...`**, using** `@does(sum_series)` **on a function with two series inputs will produce a node that outputs the sum of those series without having to implement the body again** [39] [40]**. This helps reuse generic logic (e.g. summing, identity transform) across different node definitions. -** `@subdag` **–** ***Sub-DAG inclusion.*** **Allows embedding a pre-defined sub-DAG (a set of Hamilton functions) as a single node. You supply a module or function group to load as a subworkflow. Each function decorated with** `@subdag` **acts as a gateway to a separate Hamilton function collection. (*Usage details of* `@subdag` ***are omitted for brevity, as*** `@parameterized_subdag` ***is a more powerful variant.*****) -** `@parameterized_subdag` **– Dynamically create** multiple sub-DAGs** from one function template [41] [42]. Use this when you want to run the same pipeline logic for different configurations or data sources (e.g. per region or multiple datasets) [41]. You provide: - `load_from`: one or more modules or callables defining the subdag (similar to adding modules in the Driver) [43]. - `**parameterization`: keyword args defining each subdag instance by name, each a dict of `"inputs": {...}, "config": {...}` overrides for that subdag [44] [45]. - Optionally global `inputs` or `config` that apply to all instances [46].

For example, if `feature_modules` contains a set of feature engineering functions, one could do:

```
@parameterized_subdag(feature_modules,
    ds1={"inputs": {"data": value("source1.csv")}},
    ds2={"inputs": {"data": value("source2.csv")}},
    ds3={"inputs": {"data": value("source3.csv")}, "config": {"filter":
"even_ids"}}
)
def features_union(feature_df: pd.DataFrame) -> pd.DataFrame:
    return feature_df
```

This will create three parallel sub-DAGs (for `ds1`, `ds2`, `ds3`) all using the same `feature_modules` logic but with different input data and an extra config in the third [44] [47]. The outputs of each subdag can then be collected or used downstream. *(Under the hood, this is syntactic sugar for repeating a set of functions; it's an advanced feature to reduce boilerplate when many identical DAG slices are needed [48].)*

- `@pipe_input` / `@pipe_output` **(Pipe Family)** – A family of decorators for chaining transformations *within* the DAG [49]. They allow representing a sequence of operations as distinct nodes, solving cases where one might otherwise rewrite or chain a variable. For instance, `@pipe_input(f1, f2, ..., on_input="data")` can create a pipeline where each of `f1, f2, ...` is applied sequentially to an input, each as its own node in the DAG [49] [50]. The **pipe family** supports namespacing of the generated nodes and is analogous to using pandas `.pipe()` or iterative DataFrame transformations, but yields intermediate nodes that are individually addressable [49] [51]. (As of Hamilton 2.0, `@pipe` is deprecated in favor of `@pipe_input` / `@pipe_output` for clarity [52].)
- *Use case:* If you have a series of steps that always occur in order (e.g., data cleansing functions), pipe decorators let you define them in one place and automatically expose each step as a separate node. It also makes it easy to apply the same function multiple times in a chain with different parameters [53].
- `@resolve` – A **power-user** decorator that defers application of another decorator until the DAG is being built (after config is known) [54] [55]. This is used when the structure of the DAG itself needs to depend on runtime configuration. For example, you might not know until runtime which inputs to connect to a function. With `@resolve(when=ResolveAt.CONFIG_AVAILABLE, decorate_with=lambda ...: some_decorator)`, you can supply a lambda that returns a decorator (like a `@parameterize_sources` with config-dependent values) [56] [57]. Hamilton will invoke this when building the Driver (after `.with_config()` is applied) [58] [59]. *Important:* Power-user mode must be enabled (`hamilton.enable_power_user_mode=True` in the config) for `@resolve` to work [60]. This feature is intended for scenarios where even the dependencies between nodes are dynamic based on config – use sparingly as it can reduce DAG readability [61].

### Conditional Node Inclusion

- `@config.when(...)` – Allows *conditionally including* a function in the DAG based on a configuration value [62] [63]. Use `@config.when(key="value")` to mark a function that should only be loaded if `with_config({key: "value"})` is provided to the Driver. Typically, you define multiple implementations of a concept with different suffixes, each gated by a different config. Hamilton will include exactly one matching function (and drop the suffix in the node name) [64]. For example, two implementations of `base_model`:

```
@config.when(task="binary_classification")
def base_model__binary() -> XGBClassifier: ...
@config.when(task="continuous_regression")
def base_model__regression() -> XGBRegressor: ...
```

  In the DAG, whichever `task` value is in the config yields a `base_model` node of the appropriate type [65] [66]. If no config or an unrecognized value is given, no node is created and any dependent node would fail (unless a default is provided).

- `when_not` / `when_in` **variants:** `@config.when_not(key="value")` includes the function if the config **does not** equal the value (useful to provide a default) [67]. `@config.when_in(key=[...])` and `when_not_in` check membership in a list of values [68] [69]. These can be combined to handle multiple cases and a fallback.
- **Usage:** You must pass the corresponding config via `Builder.with_config({...})` when constructing the Driver [70]. The config keys in `with_config` decide which `@config.when` functions are loaded (note: this config is about DAG *structure*, not to be confused with `inputs` or `overrides` passed at execution time [71]).

### External Data I/O in DAG

- `@dataloader` – Marks a function as a **DataLoader** that reads from an external source. Such functions should return a tuple `(data, metadata)` – the data object plus optional metadata about the load [72] [73]. Hamilton treats the tuple so that the actual node value is `data` (e.g. a DataFrame), and metadata is stored for lineage/tracking. *Example:*

```
@dataloader()
def raw_df(data_path: str) -> tuple[pd.DataFrame, dict]:
    df = pd.read_parquet(data_path)
    return df, utils.get_file_and_dataframe_metadata(data_path, df)
```

  This creates a `raw_df` node that loads a DataFrame from a parquet file and attaches metadata (like file path, row count, etc.) [72]. DataLoader nodes are typically **not cached** by default (since they represent external reads), unless explicitly configured otherwise.
- `@datasaver` – Marks a function as a **DataSaver** that writes out to external storage. The function should return a metadata dict about the save operation [73]. For example:

```
@datasaver()
def save_model(model: XGBModel, model_dir: str) -> dict:
    model.save_model(f"{model_dir}/model.json")
    return utils.get_file_metadata(f"{model_dir}/model.json")
```

  Here `save_model` produces no meaningful in-memory output (returns metadata only), but as a node it will trigger the side-effect of saving the model when executed [74].

- These `@dataloader` / `@datasaver` decorators let you include I/O in the DAG with full observability and without writing repetitive loader/saver code for each data source [75] [76]. They also enable caching & hooks around I/O (since they are normal nodes in the graph).

- `@load_from` / `@save_to` – Pre-built **materialization decorators** for common formats. These are actually provided as attributes: e.g. `@load_from.json(path="file.json")` will auto-create a node that loads that JSON file and feeds it into the decorated function [77]. The decorated function in this case takes the loaded data as a parameter (named by default based on the materializer). The type of the parameter should match the expected data type (e.g. `dict` for JSON) [78]. Conversely, `@save_to.<format>(path="...", output_name_="...")` will save the decorated function's

result to the given path and register a node with the specified output name representing the saved artifact [79]. For example:

```
@load_from.parquet(path=source("raw_data_path"))
def normalized_data(raw_data: pd.DataFrame) -> pd.DataFrame: ...
```

This will create an upstream loader node (reading the parquet) and provide its DataFrame to `normalized_data` [80]. You can even use `source()` to have the file path itself be provided via `inputs` at runtime. When multiple loads feed one function, use `inject_="name"` to distinguish them [81] (the loaded values will be passed as separate params). Similarly:

```
@save_to.json(path=source("metrics_path"), output_name_="metrics_to_json")
def eval_metric(x: np.ndarray, y: np.ndarray) -> dict: ...
```

will create a `metrics_to_json` node that, when executed, saves the `eval_metric` output to the given JSON path [82]. (You *must* provide `output_name_` for save_to nodes so they can be requested from the driver [79].)

**Materialization vs Execution:** Normally, `driver.execute(final_vars)` runs the DAG entirely in-memory. In contrast, one can use `driver.materialize(final_vars)` to explicitly trigger DataLoader/DataSaver side-effects and drop pure in-memory results when not needed. Essentially, `.execute()` returns computed values, whereas `.materialize()` emphasizes external persistence (often returning just metadata) [83]. In practice, `execute()` suffices for most use-cases, and loaders/savers will still run as part of execution. The `materialize` API is useful for programmatically constructing large I/O flows via `with_materializers` (see below).

## Dynamic DAG Patterns: Parallelization and Conditional Execution

Apache Hamilton supports **dynamic execution** patterns for parallelism and task grouping. There are two approaches to parallel DAG execution:

1. **Using Graph Adapters** – Simply swap in an adapter that runs each node on a parallel execution engine (thread pool, Dask, Ray, etc.) [84]. This requires no changes to your function definitions – the parallelism is handled at execution time.

2. **Using `Parallelizable[T]` and `Collect[T]` types** – Write your functions to generate and collect dynamic nodes at runtime (similar to a map-reduce pattern) [84] [85]. This approach uses the new **Dynamic DAG Executor**.

### Graph Adapter Parallelism

Using a parallel adapter is straightforward: when building the driver, attach an adapter via `Builder.with_adapter(...)`. For example, to use a thread pool:

```
from hamilton.plugins.h_threadpool import FutureAdapter
dr =
driver.Builder().with_modules(my_module).with_adapter(FutureAdapter()).build()
result = dr.execute(["my_variable"], inputs={...})
```

This will execute the DAG by submitting each node function to a `ThreadPoolExecutor` (great for I/O-bound tasks) [86] [87]. Hamilton provides adapters for **Ray**, **Dask**, **Spark**, etc., which you can install via extras (e.g. `sf-hamilton[ray]`) [88] [89]. These adapters wrap each node computation in a remote execution, returning futures that Hamilton waits on or collects as needed [90] [91]. The caveat is that serialization overhead can sometimes outweigh parallel gains, so benchmark accordingly [92] [93].

**Available Execution Adapters:** (import from `hamilton.plugins` module) - `h_threadpool.FutureAdapter()` – Uses Python ThreadPoolExecutor (local threads) [94]. - `h_dask.DaskGraphAdapter()` – Executes nodes as Dask delayed tasks on a Dask scheduler [91]. - `h_ray.RayGraphAdapter()` – Executes nodes as Ray remote functions (for multi-core or cluster scaling) [95] [96]. - `h_spark.PySparkUDFGraphAdapter()` – Runs nodes as UDFs in Spark (for Spark DataFrame workflows). - `h_spark.SparkKoalasGraphAdapter()` – (Deprecated Koalas support, use PySpark or Pandas API on Spark). - `h_async.AsyncGraphAdapter()` – Uses Python `asyncio` for async functions. - `CachingGraphAdapter` – Wraps another adapter to incorporate caching behavior. - (Plus any custom adapters you write by extending the GraphAdapter interface.)

You specify an adapter in the builder; you can also combine adapters (e.g., a caching adapter with a thread pool) by providing multiple via `Builder.with_adapters(adapter1, adapter2, ...)`. The **DefaultGraphExecutor** (no parallelism) is used if no adapter is given.

### Dynamic Node Generation with Parallelizable/Collect

Hamilton's **dynamic DAG** feature allows certain functions to emit **multiple outputs at runtime**, which subsequent nodes can aggregate. This is done by typing a function's return as `Parallelizable[X]` (meaning it yields a *stream* of X values) and corresponding downstream input as `Collect[X]` (meaning it expects a *collection* of X values) [97] [98]. When the DAG runs, the dynamic executor will **expand** the Parallelizable node into multiple executions (one per yielded value), then aggregate their results into a list for the Collect node.

**Enabling dynamic execution:** Call `builder.enable_dynamic_execution(allow_experimental_mode=True)` when building the driver [99]. This switches Hamilton to the V2 executor needed for Parallelizable/Collect handling [100] [101]. You should also attach a **remote executor** (e.g. a process pool) for the parallel tasks and a **local executor** for non-parallel parts via `with_remote_executor(...)` and `with_local_executor(...)` [102] [103] (if not, the builder will default to each node per task grouping on its own).

**Example:**

```
from hamilton.htypes import Parallelizable, Collect
```

```python
def url() -> Parallelizable[str]:
    # Yields multiple URLs (one per task)
    for u in list_all_urls():
        yield u

def url_loaded(url: str) -> str:
    return load_content(url)  # fetch content for each URL

def count_words(url_loaded: str) -> int:
    return len(url_loaded.split())

def total_words(count_words: Collect[int]) -> int:
    return sum(count_words)
```

In this DAG, `url()` produces a sequence of URLs. Hamilton will treat `url_loaded(url: str)` and `count_words(url_loaded: str)` as a *task group* to be repeated for each yielded URL in parallel [98] [104]. Each URL goes through `url_loaded` and `count_words` (these two are executed together as one unit per URL, likely on the remote executor), and then `total_words` receives the **collected list** of all `count_words` outputs and reduces them [105] [106].

**Task Grouping:** In dynamic mode, by default each Parallelizable-Collect block forms a task group (executed in parallel across yielded elements), while other nodes run as individual tasks [107]. You can customize grouping via the `TaskGroupingHook` if needed.

**Caveats:** Currently only one level of Parallelizable→Collect is supported (no nesting) [108], and for multiprocessing, serialization is via pickle (with its known limitations) [108]. Typically, using a robust backend like Dask or Ray for heavy parallelism is advised.

## Materialization API and I/O Adapters

Hamilton decouples **materialization** (data input/output) from core logic. Materialization APIs provide flexible ways to load initial inputs and persist outputs without hardcoding I/O in your functions, improving reuse and observability [75] [109].

**Approaches to Materialization:**

- *Simple in-code I/O:* You can always perform I/O in regular functions (e.g. call `pd.read_csv()` inside a node). However, this scatters I/O logic across the DAG and can become repetitive for many sources [110].
- *Function Modifiers:* Using `@dataloader` and `@datasaver` (or `@load_from`/`@save_to`) decorators integrates I/O as first-class nodes (with standardized implementations for common formats) [111] [77]. This improves observability – e.g., a DataLoader node is part of DAG lineage and can be cached, retried, etc. [109].
- *Builder Materializers:* You can define I/O actions at Driver construction via `Builder.with_materializers(*materializer_objects)`. The `hamilton.io.materialization` module provides factory helpers like

`from_.<format>(target=node, path="...")` or `to.<format>(id="name", dependencies=[...], path="...")` to create materializer specs [112] [113]. For example:

```
from hamilton.io import materialization as mat
mats = [
    mat.from_.parquet(target="raw_df", path=data_path),
    mat.to.json(id="model__json", dependencies=["model"],
path=f"{model_dir}/model.json")
]
dr =
driver.Builder().with_modules(modules...).with_materializers(*mats).build()
dr.execute(["model", "model__json"], inputs={...})
```

This will insert a `raw_df` loader that reads the parquet file, and a `model__json` saver that writes the `model` output to JSON [112] [114]. The `execute` call requests both the model and the saving node to run [115]. Using builder materializers is akin to "attaching" load/save steps without modifying the function definitions (good for environment-specific concerns, e.g., in production attach a saver that writes to S3).

**Data Adapters and Custom I/O:** Hamilton's design allows adding new I/O formats or backends through **DataAdapter** classes. A DataAdapter in Hamilton usually pairs a **DataLoader** and **DataSaver** implementation for a certain storage or format. The reference docs list built-in adapters and how to register your own (e.g., you could implement a DataLoader/Saver for a SQL database or an API). As long as a custom adapter provides a `load()` or `save()` method and is wired via `with_materializers` or decorators, it can be integrated. (More details in "Using Data Adapters" reference.)

**Difference `.execute` vs `.materialize`:** If using the `driver.materialize(...)` method, Hamilton will execute all required nodes but return only the *side-effect* results (e.g., metadata from savers) instead of all node values. This is useful when you primarily care about the fact that data was saved, not the in-memory data itself [83]. In contrast, `driver.execute(...)` returns the in-memory outputs as a dict. Both will trigger DataLoader/DataSaver operations; `materialize` is just a semantic hint to focus on persistence.

## Caching and Data Versioning

Hamilton's caching system allows reuse of previous run results to skip recomputation of unchanged nodes [116]. Caching is particularly powerful during iterative development or in long DAGs where upstream data doesn't change often.

**How Caching Works:** When enabled, after each execution Hamilton stores node results in a **cache store** (by default a local directory like `.hamilton_cache`) keyed by a **data version hash** [117]. On the next run, before computing a node, Hamilton will check if a cached result exists for the *same inputs and code*. If so, it loads that instead of recomputing [118] [119].

- **Data Version Hash:** Hamilton computes a hash for each node's inputs (and the node's own function code version) to uniquely identify a "version" of the output [118] [120]. The hashing leverages the **Data**

**Versioning** module, which can hash many Python types (numbers, strings, tuples, pandas DataFrames, numpy arrays, etc.) via `functools.singledispatch` [121] [122]. Complex types are hashed in a deterministic way (e.g. order-insensitive for dicts [123], recursive for nested structures, special handling for numpy/pandas to incorporate array content [124] [125]). If a type isn't directly supported, it falls back to a general strategy or can be extended via plugins (the `h_databackends` registry covers common library types like Pandas without needing to import them) [126] [127].

- **Code Version:** The cache key also reflects the function's source code (or an identifier of it). If you change a function's logic, its cached results won't be used, ensuring you don't get stale data from old code runs.
- **Cache "Stores":** The default store writes cache files to disk (under a project-specific or global folder). You can configure the cache location: by default it's `.hamilton_cache` in the current working directory, but you can set a custom path via config or environment variable (see **Storage** settings). There's also an in-memory store for caching within a single session (useful for quick re-exec without disk I/O). The cache can even persist in-memory values across driver invocations via `cache.persist()/load()` calls.
- **Enabling Caching:** Use `Builder.with_cache()` to enable caching with default settings [128]. You can also specify `with_cache(CacheConfig(...))` for advanced options (like a custom cache directory or policy). Alternatively, decorate specific functions with `@cache` (as described above) to always cache those nodes. By default, if caching is enabled on the driver, **all** nodes are cached unless marked otherwise. You can control behavior per node with `@cache` (to specify format or to force caching on/off).
- **Cache Behavior:** By default, if inputs and code match a previous run, the node is skipped (`get_result::hit` log) and loaded from cache [118] [119]. If inputs differ or no cache exists, the node is executed (`execute_node`) and result stored. You can inspect which nodes were cached vs recomputed by examining `driver.cache` metadata:
- `driver.cache.last_run_id` gives the latest execution ID, and `driver.cache.behaviors[run_id]` shows each node's cache usage (e.g., `DEFAULT` used normal caching, `RECOMPUTE` forced recompute, `DISABLE` was not cached) [129].
- `driver.cache.data_versions[run_id]` has the hash keys for each node's output in that run.
- `driver.cache.view_run(run_id)` pretty-prints a summary of a run's cache usage (as seen in logs).
- **Forcing Recompute or Disable Cache:** You can mark a node to always recompute by `@cache(..., stale=True)` or disable caching by `@cache(enabled=False)`, etc., if needed (depending on API version).
- **Cache Format:** As mentioned, the default cache store uses pickle files. You can change the format per node with `@cache(format="...")` to use, say, Parquet for DataFrames (see the example where `processed_data` node is cached to Parquet [18]). Hamilton will actually store a small pickle that points to the Parquet file and a DataLoader to retrieve it [130] – meaning when loading from cache it knows to read the Parquet.
- **Clearing Cache:** You can manually clear cache via `driver.cache.clear_storage()` (or `Dataflow.clear_storage()` in newer API) [131] for a given dataflow, which removes cached files.

*Tip:* Hamilton's caching is designed to be transparent. It significantly speeds up iterative analysis as unchanged parts of the DAG are skipped. It's also safe for production if inputs carry timestamps or version identifiers – ensuring new data triggers recompute while old unchanged data uses cache.

## Execution Backends (Graph Adapters)

**Graph Adapters** abstract how the Hamilton DAG is executed. The default is in-memory Python, but Hamilton offers adapters to integrate with various engines or frameworks for scaling or integration purposes [89] [92]. We introduced the parallel ones above (ThreadPool, Ray, Dask). Here we summarize the available adapters and their configurations:

- **SimplePythonGraphAdapter** – Default single-threaded executor in pure Python. Simply computes node functions in topological order.
- **SimplePythonDataFrameGraphAdapter** – Similar to above but returns a pandas DataFrame if possible (uses `PandasDataFrameResult` result builder under the hood) [132] [133]. Essentially, if all outputs are Pandas Series or scalars, you get a combined DataFrame result.
- **HamiltonGraphAdapter** – The "v2" executor enabling dynamic DAG. Internally used when `enable_dynamic_execution()` is called. It works with *lifecycle hooks* to group tasks, manage futures, etc. (Most users don't need to directly invoke this; use the builder toggles).
- **AsyncGraphAdapter** (`h_async.AsyncGraphAdapter`) – Runs each node as a Python `asyncio` coroutine, allowing non-blocking concurrency for I/O-bound tasks. Ensure your node functions use `await` for I/O for this to be beneficial.
- **ThreadPool Adapter** (`h_threadpool.FutureAdapter`) – Uses a local thread pool (see above). Config: can specify max workers via Python's `ThreadPoolExecutor` defaults or environment.
- **Multiprocessing Adapter** – (Hamilton doesn't have a specific named adapter in core, but using the dynamic execution with `executors.MultiProcessingExecutor` is possible [103]).
- **Dask Adapter** (`h_dask.DaskGraphAdapter`) – Offloads node execution to Dask. Typically, you need a Dask client or cluster; the adapter will create Dask delayed tasks for each node.
- **Ray Adapter** (`h_ray.RayGraphAdapter`) – Offloads node execution to Ray remote functions. Accepts `ray_init_config` (dict passed to `ray.init`) and a `shutdown_on_completion` flag in its constructor [134] [96]. Usage:

```
adapter = RayGraphAdapter(result_builder=base.ResultMixin(),
                          ray_init_config={"address": "auto"},
                          shutdown_ray_on_completion=False)
dr = driver.Driver(config, my_module, adapter=adapter)
```

  This will connect to an existing Ray cluster or start one with given config [96] [135]. The `result_builder` decides the return type (see ResultBuilders below). Note that Ray requires all data to be serializable – large pandas data may not get special treatment (Ray doesn't auto-shard DataFrames) [136].
- **Spark Adapter** (`h_spark.PySparkUDFGraphAdapter`) – Integrates with PySpark by turning Hamilton functions into UDFs applied on Spark DataFrames. This allows you to write transformation logic in Hamilton, then execute it in Spark's distributed manner. There is also `SparkKoalasGraphAdapter` for the older pandas-on-Spark API (Koalas).
- **CachingGraphAdapter** – Wraps another adapter to introduce caching at the adapter level. In practice, the standard caching mechanism (via `with_cache`) is often sufficient, but this adapter ensures that even in a distributed context, caching rules are applied.

To use adapters, pass them via `Builder.with_adapter(s)` or directly to `driver.Driver(adapter=my_adapter)` (noting that the Builder pattern is preferred for dynamic and multiple adapters). You can compose multiple adapters: e.g., `with_adapters(CachingGraphAdapter(), RayGraphAdapter(...))` to cache with Ray execution.

## Lifecycle Hooks and Monitoring (Lifecycle API)

One of Hamilton's most powerful advanced features is the **Lifecycle API**, which lets you hook into various stages of DAG execution for logging, debugging, or integration with external monitoring systems [137] [138]. These hooks are implemented as adapter classes (often called *Lifecycle Adapters* or *hooks*), and you attach them via `Builder.with_adapters(...)` similar to GraphAdapters.

Key categories of lifecycle hooks:

- **Node Execution Hooks (NodeExecutionHook / NodeExecutionMethod)** – These allow running custom code *before or after each node executes*. For example, `PrintLn` is a built-in hook that simply prints each node's input and output at runtime (for debugging) [139]. `SlackNotifier` sends a Slack message if a node fails (or succeeds) [137] [138]. `PDBDebugger` enters an interactive debugger if a node errors, pausing execution for inspection [140]. These hooks typically subclass `NodeExecutionHook` or `NodeExecutionMethod` and override methods like `run_before_node_execution` or `run_after_node_execution`.
- *Usage:* For example, to get Slack alerts, install `sf-hamilton[slack]` and add `h_slack.SlackNotifier(api_key="XXX", channel="alerts")` to your adapters [137] [141]. Then when `.execute()` runs, any node completion or error triggers a Slack message (by default, SlackNotifier posts after each node execution, indicating success or failure) [142] [143].
- **Graph Execution Hooks (GraphExecutionHook)** – These run at the DAG level (before/after the entire execute run). Could be used to initialize resources or log overall success/failure.
- **Edge Hooks (EdgeConnectionHook / StaticValidator)** – These can intercept the wiring of node outputs to inputs. Hamilton uses a StaticValidator hook internally to check that types match on connections (and one can extend it). An example plugin is `SparkInputValidator` which ensures that certain DataFrame types align between nodes for Spark [144].
- **Task Hooks (for dynamic execution)** – When using the dynamic executor (Parallelizable), hooks like `TaskSubmissionHook`, `TaskExecutionHook`, `TaskReturnHook`, `TaskGroupingHook` allow injecting behavior when tasks are submitted, started, finished, or grouped [145]. These are more specialized, e.g., for custom task scheduling or grouping logic beyond the defaults.
- **Result Builders (ResultBuilder / ResultMixin)** – These define how the final outputs of `execute()` are assembled. They are part of lifecycle because they intercept the result construction. Hamilton provides:
- **GenericResultBuilder** (default): returns a Python dict of `{node: value}`.
- **PandasDataFrameResult**: builds a DataFrame from outputs if possible (e.g. if outputs are Series) [146] [147].
- **StrictIndexTypePandasDataFrameResult**: like above but errors if Series indices don't align [148] [149].
- **NumpyResult**: perhaps stacks outputs into a numpy array (when shapes align).
- **PolarsResult**, **DaskResult**: similar concept for those libraries.
- You can create a **Custom ResultBuilder** by subclassing `lifecycle.ResultBuilder` and implementing `build_result(**outputs)` to assemble the outputs as desired [132] [147].

- To use a result builder, pass it to a GraphAdapter (many adapters accept a `result_builder` in their constructor) or use a helper adapter like `base.SimplePythonGraphAdapter(result_builder=my_builder)` [132] [150].

- **Telemetry / Monitoring Adapters:** These gather execution metadata. Examples:

- **ProgressBar hooks:** `h_tqdm.ProgressBar` and `h_rich.RichProgressBar` provide nice console progress bars for DAG execution (especially on long runs) [151] [152]. Attach them to get a live indication of which node is running.
- **Datadog Tracer (`h_ddog.DDOGTracer`)** – Sends execution timings to Datadog APM, so you can monitor DAG performance in production [153].
- **OpenLineage Adapter (`h_openlineage.OpenLineageAdapter`)** – Integrates with the OpenLineage standard for data pipeline lineage tracking [154]. It emits events about node executions and dataset inputs/outputs so that an external lineage service can log what happened.
- **MLflow Tracker (`h_mlflow.MLFlowTracker`)** – Automatically logs parameters, metrics, or artifacts of node outputs to MLflow, useful for experiment tracking.
- **Slack Notifier (`h_slack.SlackNotifier`)** – As discussed, sends Slack messages on node events [137] [138].
- **Narwhals** (NarwhalsAdapter & NarwhalsDataFrameResultBuilder) – Integration with an internal tool "Narwhals" for result tracking (less commonly used externally).
- **GracefulErrorAdapter** – Modifies how errors are handled – for instance, instead of halting on first error, it might collect errors from multiple nodes or provide a formatted error report.

**Attaching Hooks:** All these are added via `Builder.with_adapters(...)`. You simply instantiate the adapter and pass it in. For example:

```
dr = driver.Builder()\
      .with_modules(mods)\
      .with_config(cfg)\
      .with_adapters(
          h_tqdm.ProgressBar(),
          lifecycle.FunctionInputOutputTypeChecker(),  # strict type check
          h_slack.SlackNotifier(api_key="XXX", channel="#alerts")
      ).build()
```

This would build a driver that shows a tqdm progress bar, validates input/output types strictly, and sends Slack alerts on failures. Many lifecycle adapters have settings (e.g., SlackNotifier can be configured for only failures or certain channels; OpenLineageAdapter might take job info, etc. – see their docs).

**Note:** Lifecycle adapters are composable – you can attach multiple. Under the hood, Hamilton's Driver will wrap node execution calls to invoke all `pre_node_execute` hooks before a node runs, then call all `post_node_execute` hooks after, etc. [142] [143]. This system makes it easy to add cross-cutting concerns like logging, without cluttering your business logic.

## Integration with LLM/Agentic Systems (LLM Workflows & Apache Burr)

Hamilton excels at declaratively defining dataflow pipelines, including those that involve LLMs or other AI components. Many steps in an "agentic" LLM application (retrieval, prompts, data transforms) can be Hamilton nodes [155] [156]. The benefits of using Hamilton for LLM pipelines include: easy visualization of the flow, unit testing components in isolation, swapping implementations via config (e.g. different vector DBs or LLM providers), adding standard data validation/caching to any step, and getting intermediate outputs for debugging with minimal effort [157] [158].

**Direct LLM integration:** You can treat calls to LLM APIs or tools as just functions in Hamilton. For example, one node could call OpenAI's API given a prompt (with the prompt constructed by upstream nodes), another could vectorize a document via a library, another could perform a similarity search on embeddings, etc. By structuring these as Hamilton nodes, you gain the ability to rapidly re-wire the flow: e.g., switch out the embedding model by changing one function via `@config.when`, or run the pipeline for multiple inputs via dynamic execution.

Hamilton's docs provide examples like using "OpenAI function calling with a knowledge base" and a "PDF Summarizer" as Hamilton dataflows [159]. In practice, you might build an LLM app with Hamilton handling the *deterministic* dataflow (data prep, calling LLM, post-processing) and then possibly integrate a controller for *non-deterministic/looping* decisions.

**Agentic loops and Apache Burr:** Hamilton by itself is static – it doesn't natively support loops or conditional branches within the DAG execution (all branching must be unrolled at graph build time). For full agent behavior (where an LLM decides to take actions in a loop until some goal is met), consider using **Apache Burr** (incubating). Burr is a sister library focused on agentic workflows, which can call into Hamilton for structured tools. In other words, you can use Hamilton to define the tools/pipelines, and Burr to orchestrate the decision loop using those tools.

As the Hamilton team notes: *"Apache Hamilton is great for DAGs, but if you need loops or conditional logic to create an LLM agent or a simulation, take a look at our sister library Apache Burr."* [160]. Burr can integrate Hamilton drivers as actions an agent can take. For example, one could register a Hamilton dataflow to generate an answer given some context; the Burr agent can decide when to invoke that and how to iterate.

**Using Burr with Hamilton:** Burr provides an interface to build agents that leverage functions (tools). Hamilton functions can be easily exposed as such tools since they are just Python callables. The integration might involve a Burr agent calling `driver.execute` on a set of Hamilton nodes as one step. Conversely, Hamilton itself can call LLMs (e.g., a node that wraps Burr or LangChain for a single step). The combination yields a powerful setup: Hamilton ensures each deterministic piece is maintainable and testable, Burr handles control flow around those pieces.

In summary, for LLM applications: - Use Hamilton to define clear transformations: prompt creation, API calls, parsing responses, vector DB queries, etc. [155] [161]. This makes each part verifiable and reusable. - If you have a simple flow (no looping), Hamilton alone is sufficient – you get a clear DAG you can visualize and monitor. - If the app requires the agent to decide and loop (e.g., iterative question answering), pair Hamilton with Burr. Burr will own the loop and state, Hamilton will provide the execution of each action. This keeps

the agent logic separate from the transformation logic, aligning with Hamilton's philosophy of separation of concerns.

*(For more on Burr, see the Apache Burr project. Burr, like Hamilton, is incubating at ASF and designed to complement Hamilton for agentic AI workflows.)*

## Developer Tooling: UI, CLI, and IDE Support

Hamilton comes with a rich set of tools to improve the development and operational experience:

- **Hamilton UI:** A web-based application for visualizing and tracking your dataflows. The Apache Hamilton UI provides a live view of the DAG, a catalog of nodes (with documentation from docstrings and tags), and execution monitoring (it can show which nodes ran, their runtime, and output previews) [162]. It's useful for collaboration – your team can inspect the pipeline and see lineage and results without diving into code. The UI can run in **Local Mode** (launch a server for your DAG on localhost) or **Deployed Mode** (host it for broader access) [163] [164]. Key features include searching nodes, viewing upstream/downstream, and comparing runs.
- *Usage:* To use the UI, install Hamilton with the `ui` and `sdk` extras: `pip install "sf-hamilton[ui,sdk]"` [165]. Then you can run `hamilton ui [MODULES...]` via CLI or use the SDK to start the UI server programmatically. The UI will crawl your module(s) for Hamilton functions and present the DAG. In local mode, it even auto-reloads as you edit code.

- The UI is especially powerful combined with telemetry hooks (e.g., OpenLineageAdapter will surface data lineage metadata in the UI). It also allows triggering new executions with different inputs from a web form, and then inspecting results. (Refer to Hamilton's UI docs for setup and capabilities.)

- **Command-Line Interface (CLI):** Hamilton provides a CLI tool `hamilton` that can perform various tasks on your DAG code:

- `hamilton build MODULES...` – Build a DAG from the given modules and output a visualization (Graphviz .dot or image) or some summary.
- `hamilton view MODULES...` – Launches the UI for the specified modules (similar to `hamilton ui`).
- `hamilton diff MODULES...` – Compare two sets of modules or two versions of a DAG (to see what nodes changed, added, removed).
- `hamilton version MODULES...` – Compute a stable "version" hash of the DAG (based on function code and dependencies), useful for verifying if code changes affect the DAG structure [166].

These commands help integrate Hamilton into CI/CD – e.g., you can diff DAGs between git commits to ensure changes are expected. Run `hamilton --help` or see the CLI reference for full options.

- **VSCode Extension & Language Server:** Hamilton has an official VSCode extension (powered by the Hamilton Language Server) to enhance the editing experience [167] [168]. Features include:
- *Automatic Dataflow Visualization:* As you edit a Python file with Hamilton functions, the extension can display the DAG of that file in real-time [169]. This helps you see the impact of changes immediately. You can rotate or reposition the graph and even visualize partial DAGs.

- *Intelligent Autocomplete:* When writing a function definition, the extension suggests existing node names as you type parameters, and will insert the correct type if available [170]. It understands your DAG context, so it can autocomplete with valid upstream nodes.
- *Hover & Docstrings:* Hovering over a node name shows its type and docstring, even if defined via decorators (it can resolve nodes created by @parameterize, etc.) [171].
- *Outline & Navigation:* VSCode's outline view lists Hamilton nodes separately, and you can jump to a node's definition or find references (e.g., which functions depend on a given node) [172].
- *VSCode Walkthrough:* The extension provides a guided walkthrough to set up a project with Hamilton, accessible via the Hamilton side panel [173].

The extension is experimental but rapidly evolving [174]. Install it from the VSCode Marketplace (search "Apache Hamilton") [167] [168]. It significantly boosts productivity by making the DAG a first-class part of your coding workflow.

- **Jupyter Notebook Integration:** Hamilton can be used in notebooks as well – while there isn't a full UI in notebooks, you can still define functions and construct a Driver. There is mention of a potential Jupyter extension in the roadmap [175]. In practice, many use the `%load_ext hamilton` magic (if available) or simply define functions in a cell and then do `driver.execute` to get results in the notebook.

- **Pre-commit Hooks:** Hamilton's `extension autoloading` and provided pre-commit hook can enforce style (like ensuring function names match their return types) and even auto-generate some documentation indices. These help maintain consistency in large codebases (see docs on *pre-commit hooks* for details).

In summary, Apache Hamilton's tooling ecosystem (UI, CLI, IDE integration) is geared towards making DAG development and maintenance as interactive and transparent as possible. By leveraging these tools, you get immediate feedback (via visual DAGs and autocompletion) and can more easily collaborate and deploy Hamilton-based pipelines.

---

**References:** This reference guide is based on Apache Hamilton's official documentation and examples, including the Concepts and Reference sections of the Hamilton incubator site [1] [32], materialization how-tos [72] [74], caching tutorial [17] [18], dynamic execution discussions [98] [104], and the Apache Hamilton PyPI description [160]. For further details on any topic, consult the Apache Hamilton docs or the Hamilton community Slack.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 62 63 64 65 66 67 68 69 70 71 77 78 79 80 81 82 83 111 151 152 153 154 Function modifiers - Hamilton
https://hamilton.apache.org/concepts/function-modifiers/

17 18 19 20 117 118 119 120 128 129 130 Caching - Hamilton
https://hamilton.apache.org/how-tos/caching-tutorial/

36 37 38 39 40 does - Hamilton
https://hamilton.apache.org/reference/decorators/does/

41 42 43 44 45 46 47 48 parameterized_subdag - Hamilton
https://hamilton.apache.org/reference/decorators/parameterize_subdag/

49 50 51 52 53 pipe family - Hamilton
https://hamilton.staged.apache.org/reference/decorators/pipe/

54 55 56 57 58 59 60 61 resolve - Hamilton
https://hamilton.incubator.apache.org/reference/decorators/resolve/

72 73 74 75 76 109 110 112 113 114 115 Materialization - Hamilton
https://hamilton.staged.apache.org/concepts/materialization/

84 85 86 87 90 91 94 97 98 99 100 101 102 103 104 105 106 107 108 Dynamic DAGs/Parallel Execution - Hamilton
https://hamilton.apache.org/concepts/parallel-task/

88 89 92 93 95 96 134 135 136 h_ray.RayGraphAdapter - Hamilton
https://hamilton.apache.org/reference/graph-adapters/RayGraphAdapter/

116 Caching - Hamilton - The Apache Software Foundation
https://hamilton.apache.org/concepts/caching/

121 122 123 124 125 126 127 Data versioning - Hamilton
https://hamilton.apache.org/reference/caching/data-versioning/

131 163 164 Concepts - Hamilton
https://hamilton.apache.org/concepts/

132 133 146 147 148 149 150 Pandas - Hamilton
https://hamilton.apache.org/reference/result-builders/Pandas/

137 138 139 140 141 142 143 144 145 plugins.h_slack.SlackNotifier - Hamilton
https://hamilton.staged.apache.org/reference/lifecycle-hooks/SlackNotifierHook/

155 156 157 158 159 161 LLM workflows - Hamilton
https://hamilton.apache.org/how-tos/llm-workflows/

160 162 165 sf-hamilton · PyPI
https://pypi.org/project/sf-hamilton/

166 Command line interface - Hamilton
https://hamilton.apache.org/how-tos/cli-reference/

167 168 169 170 171 172 173 174 175 Apache Hamilton VSCode - Hamilton
https://hamilton.apache.org/hamilton-vscode/vscode_extension/