The Graph Follows Execution

What Survives Running, and What Must Be Written First

Dirk Harms-Merbitz
LinuxToaster.com
dhm@linuxtoaster.com

Abstract

Ask of any part of a system's graph: could this have been reconstructed from a record of a run that already happened? A process tree, a flame graph and a call graph answer yes, which is why each is a program you run afterwards rather than a file you edit first. A durable execution journal answers partly — the history is generated by running, but a journal can only record boundaries somebody declared. An approval gate answers no, and not merely because it must precede what it gates: what a gate encodes is what it would do on inputs that did not arrive, and an execution takes one path.

Recoverability is a property of the information, not a choice available to a designer. The move this paper makes follows from taking that seriously: existing frameworks classify nodes by function — router, judge, retry, checkpointer — and we classify the information those nodes carry by whether running the system can generate it. The familiar distinction between graphs that are written and graphs that are read is one consequence. We call these the *prescriptive* and *descriptive* uses, and argue that agent orchestration frameworks are caught between them. The claim is not that graphs cannot execute — Petri nets, statecharts, BPMN and every dataflow language execute, and the workflow systems in data engineering are right to be written in advance. It is narrower: a graph that records discovered structure should not have to exist before the discovery. Where a task's decomposition is itself the work, requiring the graph up front asks the developer to finish a search before the computation that would perform it has started.

Sorting a framework's own primitives supplies the evidence, and the three classes it yields are the three answers to the recoverability question. Some are operating system concerns with an owner since the 1970s — processes, message passing, lifetime, reaching a host. Some are gaps the kernel really does not fill, chiefly durability across process death, and durable execution engines are building the substrate for them. The rest — router, retry, judge, validator, approval gate — are not infrastructure. Each records a place where a model could not be trusted with something. We argue this last class is not disappearing but *churning*: individual nodes are indexed to a capability level and expire, while the class persists because rising capability is spent on wider scope. That makes it the fastest-moving part of the system, and Hydra's 1975 argument is precisely that the fastest-moving part is what must not be frozen into the same file as the slowest.

The design conclusion is not that agent systems should have no written graph but that they should have two, split where recoverability splits them: write the counterfactual part, recover the rest. A case study of `toast` shows the recovered half is constructible — the graph is a process tree and a message log rather than a file. We state what it does not get — no durable execution, no proof of behaviour before running, a shell as a prerequisite — and give predictions checkable against framework release histories.

Keywords: agent frameworks; orchestration; mechanism and policy; durable execution;

dataflow; Unix

1 Introduction

Draw the graph, then run it. LangGraph, AutoGen, CrewAI and the visual builders around them differ in almost everything else and agree on that. Nodes are steps, edges are control flow, a runtime walks the structure, and the developer's job is to have the structure right before anything executes.

Now consider a call graph. Nodes are functions, edges are control flow, and the shape is indistinguishable from the first. Nobody proposes writing one as source, and the reason is not aesthetic. Running the program produces it.

That is the question this paper is built on, and a reader can apply it immediately to anything:

> Could this have been reconstructed from a record of a run that already happened?

Call this the *recoverability* question, and note that the answer is a property of the content rather than a decision available to a designer. One sentence follows from it:

> Running a system produces some of the information in its graph and none of the rest. Which parts are which is not a choice.

Applied to real examples the question sorts information — not graphs — into three kinds.

Produced by execution.
Running the system generates it as a by-product. The process tree, the call graph, the sampled stacks behind a flame graph, the packets on the wire. This is why `pstree` and `gprof` are programs you run afterwards rather than files you edit beforehand.

Produced in part.
A durable execution journal is a record of what happened, so the history is generated by running. But a journal can only record boundaries somebody declared, and the declaration is not a by-product of anything. History generated, frame supplied.

Counterfactual.
Never present in any single execution, because it is not about an execution. An approval gate encodes what it would do across inputs, including the ones that never arrived. A run takes one path. A trace is one sample; a policy is a mapping, and you do not get the mapping from the sample.

None of that is about graphs. A graph is a container, and the reason arguments about graph frameworks go in circles is that a real one holds a mixture of all three kinds and draws them identically. The familiar distinction between graphs that are written and graphs that are read falls out once the contents are separated: a *descriptive* graph carries the first kind and can therefore be produced by execution and read afterwards; a *prescriptive* graph carries the third and must exist first, be read by a runtime, and make things happen — an Airflow DAG, a Dryad job, a LangGraph `StateGraph`. The notation does not distinguish them. Recoverability does, and it is not a property any feature comparison of these tools reports.

What this is not an argument against. Graphs execute. Petri nets execute [14], statecharts execute [6], BPMN executes, every dataflow language executes [9], and the workflow engines of data engineering are right to be authored in advance. Prescriptive graphs are a fine and well-founded idea, and where the structure of a job is known before it runs they are the correct tool. The claim here is narrower and, we think, harder to argue with: *recoverable information should not have to be written down first.* Where the decomposition of a task is itself part of the task — audit this codebase, design this system, find every instance of a problem nobody has characterised — the decomposition is recoverable and nothing else is available anyway, so requiring it up front asks the developer to finish a search before the computation that would perform it has started.

The frameworks are already moving. This is not a critique from outside. LangGraph's `Command` type performs routing without a predefined edge, deciding the next node at run time [10]; checkpointers and `interrupt` exist because runs outlive processes. A graph framework has added a way to not have the graph in advance. We read that as convergence, and Section 5.1 returns to it.

Sorting the boxes. The evidence comes from the frameworks' own vocabulary. Take a list of node and primitive types and sort it, and three different kinds of thing come out:

Mechanism the operating system already provides.
Spawning work, passing messages, parent and child, lifetime, scheduling, reaching another host. These have had an owner since the 1970s. A framework that implements them again has moved a solved problem back up into application code.

Mechanism the operating system does not provide.
Survival across process death, exactly-once external effects, a pause that outlives the machine it started on. A process table is volatile and a process tree does not survive a reboot. This is a real gap, it is not what a kernel does, and durable execution engines [16, 2] are the substrate being built to fill it.

Policy that compensates for an untrusted model.
Router, retry, judge, validator, critic, fallback, approval gate. These are not infrastructure. Each one is a place where a developer stopped trusting the computation and wrote down what to do instead.

The three have different owners and different rates of change, and a graph language draws all three with the same boxes and the same arrows in one file. Systems design named that error in 1975: Hydra separated mechanism from policy as a design principle, on the grounds that the two change at different rates and mixing them makes the slow thing inherit the churn of the fast one [11].

These are not two separate arguments, because the three classes are the three answers to the recoverability question. M-OS is what execution produces. M-GAP is the mixed case: the journal is generated, the boundaries it journals are declared. P is counterfactual and therefore present in no execution at all. So the question of whether a graph should be written or read is not answerable about the graph at all. It decomposes into which class each box is in, and for any real system the answer is mixed. Section 3.4 draws the design consequence: not one graph and not none, but two — write what cannot be recovered, recover the rest.

That is the move, and it is worth stating in the form a reader might repeat:

> Existing frameworks classify nodes by function. This paper classifies the information those nodes carry, according to whether execution can generate it.

Everything below is a consequence of applying that classification, and we list the consequences rather than presenting them as independent claims, because they are not independent and a reader should not have to weigh them one at a time:

1. **Recoverability as the organising variable** (Sections 3.4, 6 and 5). Graphs are usually compared by what they can express. We classify the *information* in them by whether execution produces it, which yields the descriptive/prescriptive distinction as a consequence rather than an assumption, sorts call graphs and process trees onto one side and workflow DAGs onto the other, and explains why agent frameworks sit awkwardly between.
2. **A three-way classification of framework primitives** (Section 3), applied to a documented framework's own vocabulary. The middle class — mechanism the kernel genuinely lacks — is what makes this more than a complaint. It concedes the strongest counterargument and locates it.
3. **An argument that the policy class churns rather than shrinks** (Section 4). Constraint requires enumerating what a system might do; review examines what it did. The first grows with the scope you are willing to grant, and scope is what improving capability buys you. Individual policy nodes are indexed to a capability level and expire; the class persists. That combination — unstable membership, stable existence — is exactly what should not be frozen into a language, and it is the shape of the knowledge acquisition bottleneck that constrained the first expert system wave [3, 7].
4. **An existence proof** (Section 7). `toast` is not offered as a better framework. It is offered as evidence that a system's graph can be reconstructed rather than authored — that the descriptive arrangement is buildable for recursive decomposition across hosts — together with an account of what that costs, which is more than a position of this shape usually admits.
5. **Predictions** (Section 8) that can be checked against framework release histories rather than argued about.

This is a position paper. It contains no user study and no benchmark. What it does contain is a classification that anybody can apply to a framework they use and disagree with item by item, and predictions specific enough to be wrong.

2 Background and Related Work

2.1 Graphs that are read, not written

The graph of a program's execution is old and well understood as an output. `gprof` produced call graphs from instrumented runs in 1982 [4]. Flame graphs render sampled stacks after the fact [5]. `pstree` draws the process hierarchy, `lsof` lists the descriptors connecting it, and a packet capture reconstructs the edges between machines. In each case the artefact describes what happened, and its value comes from that: a call graph is worth reading precisely because nobody wrote it.

2.2 Graphs that are written, not read

The opposite tradition is equally old. Dataflow languages made the graph the program [9]. Scientific workflow systems — Taverna, Kepler and their descendants — gave domain scientists a graph to compose [17], and Airflow and its peers made the directed acyclic graph the unit of scheduling in data engineering. Dryad expressed distributed computation as a graph of sequential programs [8]. None of these are mistakes. They are all cases in which the structure is known before execution, which is exactly the condition Section 6 identifies as the one that makes writing the graph reasonable.

2.3 Durable execution

The strongest objection to the position taken here has an established literature. Execution normally lives inside a process on a machine, and ends when that process ends [16]. Durable execution systems virtualise it: Temporal journals completed steps and replays deterministic workflow code against the history to reconstruct state in a new process after a crash; Restate and DBOS offer the same guarantee with different packaging [16, 2]. Agent frameworks have adopted the pattern directly. LangGraph's checkpointers persist state after every step to SQLite or Postgres, and its `interrupt` primitive pauses a run, saves state and waits for an external resume that may arrive from a different process days later [10].

A process tree is not durable. Anyone who says the kernel already owns the agent graph has to explain what happens on reboot, and "the kernel already has it" is not an answer.

2.4 Agent frameworks

Interleaving reasoning with tool invocation [19] generalised quickly into frameworks that arrange multiple such loops. LangGraph exposes state, nodes, edges, conditional edges, checkpointers and interrupts, along with a `Command` type that performs routing without a predefined edge [10]. That last primitive is worth noting early: a graph framework has added a way to decide the next node at run time, which is a small admission that the structure is not always available in advance. Agents that operate against a repository with little human intervention [18] sit at the far end of the same trend.

2.5 Mechanism and policy

Hydra established separation of mechanism from policy as a kernel design principle: the kernel provides mechanisms almost exclusively, and policies that manipulate them live in user-level software outside it [11]. The argument was about flexibility, and it rests on a rate observation — policies change often and mechanisms rarely, so embedding one in the other propagates churn downward.

2.6 Expert systems and the knowledge acquisition bottleneck

Rule-based expert systems worked in domains where the rules could be written down, and the limit they met was not accuracy but authorship. Feigenbaum named knowledge acquisition as the central problem of building them [3], and the engineering literature that followed treated eliciting and maintaining rules as the dominant cost [7]. Section 4 argues that a growing catalogue of validator, judge and approval nodes is the same cost in different notation.

2.7 Conway's Law

Conway observed that organizations produce designs that copy their own communication structures [1]. The usual reading is organizational. Section 5.1 applies it at the scale of one author and one afternoon.

2.8 Relation to the companion paper

A companion paper argues that the level of delegation to a language model belongs to the task rather than the system, and that the operator's remaining decision is also the last place their understanding of the work is refreshed [12]. That paper is about where the person stands; this one is about where the structure comes from. They share a case study, a premise — that composition, scheduling and state belong to the shell rather than to a product — and a move. Both replace a capability question with a question about information, and they do it from opposite ends. The companion paper classifies tasks by where information about the work reaches the person: it tracks acquisition. This one classifies a system's structure by where the information in it originates: it tracks production. Instead of asking how autonomous a system is, ask where the operator's understanding was last refreshed; instead of asking what a graph notation can express, ask which of the information in it running can generate. Either paper can be read without the other.

3 Three Classes of Node

Take the vocabulary a framework documents and sort it. We use LangGraph because its primitives are named and stable enough to cite [10]; the exercise is meant to be repeatable against any framework, and disagreement about individual assignments is the useful kind.

3.1 M-OS: already owned

The kernel has maintained a live execution graph since the 1970s [15, 13]. The process table is the node list. The parent-child relation is the topology. File descriptors are the edges and sockets are the edges that reach other machines. It is scheduled, routed, isolated and paged by code that every program ever written has helped to test.

A framework that reimplements node lifetime, message passing and concurrency above that has not built new infrastructure. It has moved fifty-year-old infrastructure up one level, into a slower language, with less testing, and made it available only to programs written against that framework.

3.2 M-GAP: genuinely missing

The honest part. Processes die. The process table does not survive a reboot, a pipe does not journal what crossed it, and `wait` does not remember. An agent task that runs for three days, pauses for a human approval overnight and must not send the same email twice needs guarantees no kernel offers [16, 2].

So this class is not framework bloat. It is a real substrate concern that was missing, and the field is responding correctly: durable execution engines are being built as substrate, underneath application code, exactly where the Hydra argument says mechanism belongs. The criticism this paper makes of M-GAP is therefore narrow, and it is about location rather than existence

Primitive	Class	Recoverable?	Who owns it
Node (unit of work)	M-OS	yes	process; `fork`/`exec`
Edge (control transfer)	M-OS	yes	pipe, exit status, wait
Subgraph / nesting	M-OS	yes	process tree
Message passing	M-OS	yes	pipes, sockets, IPC
Concurrent execution	M-OS	yes	scheduler
Reaching another host	M-OS	yes	network stack
Checkpointer / persistence	M-GAP	boundaries first	durable execution engines
Resume after crash	M-GAP	boundaries first	journal and replay [16]
`interrupt` spanning days	M-GAP	boundaries first	durable pause; outlives the process
Exactly-once external effect	M-GAP	boundaries first	journalled side effects
Conditional edge / router	P	no	the developer, anticipating branches
Retry node	P	no	the developer, anticipating failure
Validator, judge, critic	P	no	the developer, anticipating wrong output
Approval gate	P	no	the developer, anticipating cost
Fallback, exception handler	P	no	the developer, anticipating the unanticipated

Table 1: Framework primitives sorted three ways, with the column that connects the classification to the direction claim. *Recoverable* asks whether this part of the graph can be reconstructed from a trace rather than written in advance. M-OS can: the operating system produces it as a by-product of running. M-GAP can only in part — a journal records the step boundaries somebody declared, so the boundaries must be prescribed even though the history is descriptive. P cannot, in principle: a judge has to exist before the output it judges. Section 3.4 draws the consequence.

— when durability is expressed as node types in the same diagram as `judge` and `validator`, a substrate concern has been mixed with an application one, and the substrate inherits the churn.

3.3 P: policy, and what it is compensating for

Watch the order in which these arrive in a real system. Nobody designs a validator on day one because validators are elegant. The model produced nonsense, so a validator went in. It took the wrong branch, so a router. It gave up early, so a retry. It failed to notice its own error, so a judge. It did something expensive, so an approval gate.

Each of these marks a place where the designer stopped trusting the computation and started trusting themselves. That is not a criticism of the designers, and in many cases it was the correct call. It is a countable property of an architecture, and worth counting, because Section 4 argues the count only moves one way.

3.4 Applying the recoverability question

The three classes are the three answers to the question of Section 1: could this have been reconstructed from a record of a run that already happened? That is what the *Recoverable?*

column of Table 1 records, and it is why the classification and the direction claim are one argument rather than two neighbours.

M-OS is recoverable.
The operating system produces this graph as a by-product of running: the process table is the node list, the parent-child relation is the topology, descriptors and sockets are the edges. Nobody has to write it, which is precisely why `pstree` is a program you run afterwards rather than a file you edit beforehand. Authoring this part is optional, and frameworks that author it have chosen the prescriptive form for content that did not require it.

M-GAP is half recoverable.
A journal is a descriptive artefact — it records what happened — but it can only record boundaries somebody defined. Temporal requires nondeterministic work to be marked as activities in advance so that replay is sound [16]; LangGraph persists state at node boundaries that the developer declared [10]. The history is descriptive and the step boundaries are prescriptive, and durability genuinely requires both. This is the class that shows the direction is not simply a matter of taste.

P is not recoverable, and the reason is deeper than ordering.
The obvious objection is temporal — a judge must exist before the output it judges — and it is true but not the strongest form. The strongest form is that policy is *counterfactual*. What an approval gate encodes is not what happened but what it would do across inputs, including the ones that did not arrive. An execution takes one path, so a trace is a single sample and a policy is a mapping — and a function is not recoverable from one evaluation of it. No quantity of tracing yields a decision rule over cases that never occurred. This is not a limitation of current instrumentation but a statement about what a record of events contains.

So the question "should this graph be written or read?" is not answerable at the level of the graph. It decomposes into the question of which class each box is in, and a real system's answer is mixed.

A note on vocabulary. We say *recoverable* and stop there. The temptation is to reach for conservation language, and it should be resisted: nothing here is a claim in the Shannon, thermodynamic, reversible-computation or algorithmic-information sense, and borrowing those words would import formal commitments the argument does not make and cannot discharge. Recoverability is narrower, it is decidable case by case, and it is exactly as strong as what follows requires.

The design consequence. What follows is not that agent systems should have no written graph. It is that they should have two, split where recoverability splits them: write the counterfactual part — policy, and the step boundaries durability requires — and recover everything else by running. Current frameworks author all three classes in one file, which forces recoverable content into written form for no reason except that it shares a diagram with content that had to be written.

Stated without the vocabulary: do not maintain by hand information the runtime already produces. That is an old principle rather than a new one, and the paper's contribution is a test for deciding which information qualifies.

3.5 Two questions

The classification also reduces to two questions that can be asked of any node in any framework, and between them they account for very nearly all of them:

> Does this exist because the operating system does not provide something?
> Does this exist because I do not trust the model?

The first question has a further split — whether the operating system does not provide it, or does provide it and the framework built its own — which is the M-GAP / M-OS line. The second question does not admit a substrate answer at all, and that is the subject of the next section.

4 The Policy Class Churns

There are two ways to get dependable work out of a worker you cannot fully trust. You can constrain what they are allowed to do, or you can review what they did. Graph frameworks are an instance of the first. The second is older, and the two do not scale the same way.

To constrain, you enumerate: every branch the thing might take, every way it might fail, every case worth stopping for. The list grows with the scope you are prepared to grant. To review, you look at what came out — one result, however it was arrived at.

constraint	$\propto$ the space of things it might do
review	$\propto$ the things it did

One of those is a search space and the other is a sample from it, and the gap between them widens in the direction the field is moving. This is the part worth stating carefully, because it is easy to state wrongly. Improving capability does not make a model fail more often; if it did, the constraint burden would fall. It makes you hand the model more scope, and the enumeration is over scope. The failure rate can decline while the authoring burden rises, and that is the claim.

4.1 The same bottleneck, in different notation

Rule-based expert systems met this limit. The rules were not wrong. They had to keep being written, by people, for each situation somebody anticipated, and the writing never finished — knowledge acquisition, named as the field's central problem in 1977 [3] and treated as the dominant cost of building such systems thereafter [7].

An inference engine surrounded by hand-written rules about when to believe it is an expert system. The novelty in the current instance is that the inference engine is a language model, and it is a real novelty — the engine now generalises, which is what the old ones could not do. But the surrounding rules are authored the same way, for the same reason, with the same growth condition.

4.2 Why this class cannot be absorbed downward

Section 8 treats this as a prediction rather than a certainty, but the structural point can be stated here. Mechanism gets absorbed by a substrate because the substrate can do it correctly for everybody: register allocation went into the compiler, retransmission into the kernel, leader election into a library and then into a rented service. Absorption requires a stable specification of the thing to be owned.

A `judge` node has no such specification. It exists because a particular model, on a particular class of task, at a particular moment, could not be relied on to check its own work. There is nothing for a substrate to take ownership of, because the concern is indexed to a capability level. The node's removal condition is not better infrastructure. It is a better model.

4.3 Two claims that are easy to run together

The previous paragraph invites an inference we want to block, because we made it in an earlier draft and it does not hold. If every policy node has a capability-indexed removal condition, and capability keeps rising, it is tempting to conclude that the policy class is temporary and will drain away. It will not, and our own argument is the reason.

The two claims are different:

1. **Each node is temporary.** A validator written because a 2024 model hallucinated citations has a removal condition, and a model that does not hallucinate citations removes it. This we do claim.
2. **The class is temporary.** The catalogue as a whole shrinks toward nothing as models improve. This we do *not* claim, and Section 4 argues against it. Rising capability is spent on wider, more expensive and more consequential scope. A model trusted to draft a paragraph needed a style check; a model trusted to move money needs an approval gate. The old node retires and a more consequential one is written.

What follows is not disappearance but *churn*: unstable membership, stable existence. Each entry expires; the register does not. And churn, rather than size, is what makes this class the wrong thing to put in a graph language. Hydra's argument was never that policy is unimportant — it was that policy changes at a different rate from mechanism, so embedding one in the other propagates the fast thing's churn into the slow thing [11]. A framework that expresses `judge` and `socket` as the same kind of box, in the same file, has arranged for a validator's obsolescence to be a change to the program's structure.

This is a weaker conclusion than the one we started with, and a more defensible one. It also sharpens the prediction in Section 8: what should be observable is not a shrinking node catalogue but a high rate of addition and deprecation within a roughly stable one.

5 Decomposition as Search

Ask for something with a known shape and a graph is a good way to say it. Now ask for this:

```
$ toast "audit this kernel for security bugs"
```

There is no correct decomposition. By subsystem? By bug class? By file age, by contributor, by the way memory moves through the tree? Nobody knows, and the reason nobody knows is that finding out is most of the problem.

Requiring the graph in advance asks the developer to finish that search before the computation that would perform it has started. The decomposition is not the setup. It is the work, and it is discovered by doing the work. A system that cannot restructure its own approach halfway through can only solve problems somebody had already solved on paper.

5.1 Whose decomposition is it?

Conway's Law says organizations produce designs that copy their own communication structures [1]. Usually this is an accident and something teams work to design around. A hand-drawn agent graph is the same effect at the scale of one person on one afternoon, and it is not accidental — the decomposition is authored deliberately and shipped as the program.

That decomposition is captured at the moment its author understood the problem least, before any of it had run, and then executed unchanged for as long as the system lives. For a known workflow this is fine and even desirable: a shared object a team can argue over, diff and version. For a task whose structure is the unknown, it freezes the wrong thing.

5.2 The framework's own admission

The evidence flagged in Section 1 belongs here. LangGraph's `Command` type performs routing without a predefined edge, deciding the next node at run time [10]; a graph framework has added a way to not have the graph in advance. Read uncharitably this is a concession. Read the way we intend it, it is convergence: the pressure toward late-bound structure is real and is being felt from inside, and the interesting question is how far a prescriptive notation can travel in that direction before it is doing something else.

6 Graphs Are Read After the Fact

Here is a graph. Nodes, edges, control transferring between them:

```
main
  parse_config
    malloc
    read
    free
  connect
    socket
    malloc
  serve
    malloc
    write
    free
```

Nobody proposes writing that as the source, and not because it is uninformative. It is one of the most informative things available when something has gone wrong. It is produced by running the program, and the execution came first.

An agent graph has the same shape and the opposite direction. Flame graphs, process trees, packet captures and dependency graphs are all generated after execution, and each is useful because it describes what happened rather than what somebody expected to happen [4, 5]. Turn one of them into a language you write in beforehand and the property that made it worth having is the thing you gave up.

7 Case Study: The Graph as a Process Tree

`toast` is a command-line tool in which no graph is written. It reads standard input, writes standard output, edits a named file in place when given one, sets an exit code, and reads dot-files

Figure 1: The same object, two directions. Where the structure of a task is known before it runs, writing it down is reasonable and the left-hand arrangement is correct. Where finding the structure is most of the task, the left-hand arrangement requires the search to be finished before the search can start.

from the working directory for context. Composition, scheduling, persistence and concurrency are the shell's job. A `.tools` allowlist, one command per line, states what the process may invoke; putting `toast` on that list lets `toast` call `toast`.

Start one process:

```
$ toast "find every security issue in this repository"
```

It decides the job splits four ways and invokes itself four times. One of those finds another machine on the message bus, which starts two workers of its own. One checkpoints to a durable log. Another asks a third machine for a second opinion.

What is the graph? It does not exist yet. It is being created by the computation as the computation runs, and when everything finishes it can be reconstructed exactly — from the process tree, the messages on the bus, and a log that recorded not only what each step changed but why. The graph is the trace. It was never the program.

7.1 What this design does not get

This is where a position paper usually stops, and stopping here would misrepresent the trade.

No durable execution. Everything in Section 3.2 is missing. Processes die, the tree dies with them, and a three-day task that must survive a reboot and must not repeat an external side effect is not served by this design as described. That is the honest gap, and closing it means adding a journal — which is to say, adopting the substrate the durable execution engines are building, not arguing that the kernel already had it.

No proof before running. A trace tells you what happened. It cannot tell you what could have happened. Where a system must be certified before it executes — avionics, medical devices, anything with a regulator in the room — a written graph is not a stylistic preference but the requirement, and no quality of trace substitutes for it.

A shell as a prerequisite. The design purchases its properties with fluency most users of AI tooling do not have, and we do not know whether the properties survive a graphical interface.

Harder in the small. Each process decides locally whether to decompose, ask, checkpoint, retry or stop, and the orchestration is what those decisions add up to. That is a distributed system rather than a workflow engine: easier to grow, harder to reason about at any one moment, and subject to every failure mode distributed systems have.

7.2 What it does establish

That the mechanical half of an agent graph can be supplied entirely by the operating system, for a class of tasks that includes recursive decomposition across hosts; that the structure can be recovered afterwards in enough detail to debug; and that a system with no written graph is buildable at all, which is the minimum an argument of this shape owes the reader.

8 Predictions

The three classes should have three different futures. Stated so they can fail:

1. **M-OS moves down or disappears.** Framework primitives that duplicate kernel services will be increasingly delegated rather than reimplemented. Checkable against release notes and dependency graphs. Falsified if frameworks continue to grow their own schedulers, transports and process models.
2. **M-GAP consolidates into substrate.** Durability, replay and exactly-once effects will move out of agent frameworks into engines specialising in them, with frameworks integrating rather than implementing. Partly visible already [2]. Falsified if agent frameworks converge on their own durable runtimes instead.
3. **The P class churns without shrinking.** The prediction most likely to be wrong, and the most interesting to check. Across successive releases of a framework, count the trust-compensating node types — router, retry, judge, validator, critic, fallback, approval — and separately count additions and deprecations. We predict a high rate of both against a roughly stable total, and no downward trend in the total as the capability of the models these frameworks are used with rises. Falsified if the catalogues shrink; also falsified, though differently, if membership turns out to be stable, which would make the class ordinary infrastructure and undermine Section 4.3.
4. **The graph migrates to the output side.** Today's frameworks make good debugging tools, and the visualisation work transfers directly. We expect the picture to survive as a trace view — read when something has gone wrong — rather than as a language.

Prediction 3 is the load-bearing one, and it has two ways to fail rather than one. A shrinking catalogue would mean the class is draining and the mechanism/policy objection loses its urgency. A stable catalogue with stable membership would mean these nodes are settled infrastructure and belong in a language after all. The argument here needs the third case: the total holding while the entries turn over.

9 Limitations

One framework's vocabulary. Table 1 classifies primitives from documentation at a point in time. Frameworks change quickly, some assignments are arguable, and the exercise is offered

as one anybody can repeat and dispute rather than as a survey. A proper version would classify several frameworks across releases, which is also what would test prediction 3.

The boundary cases are real. A conditional edge that routes on a deterministic business rule is not compensating for an untrusted model; it is ordinary control flow, and we have classified it as policy anyway. A retry against a rate-limited API is infrastructure, not distrust. The classes are cleaner in the argument than in a codebase.

Two contributions, one paper. A reader may reasonably hold that the descriptive/prescriptive distinction and the three-way classification are independently publishable, since each survives the rejection of the other. We think Section 3.4 is a real dependency rather than a bridge of convenience — the classes decide which parts of a graph can point which way, and the design consequence follows from the pair and from neither alone. A reader who finds that dependency unpersuasive is holding a paper with two ideas in it, and should weigh them separately.

No empirical evaluation. Nothing here measures development time, defect rates or system reliability under either arrangement. The asymmetry in Section 4 is an argument about growth rates, not a measurement of them.

The position is narrow, deliberately. A graph is an excellent representation of a workflow whose structure is known, and the only right answer where behaviour must be proved before execution. The claim is that a graph is the wrong *primary* abstraction for a system whose decomposition is part of the computation. That is a mismatch between an abstraction and a class of problem, not a defect in the abstraction.

Competing interests. The author develops `toast` and is affiliated with LinuxToaster.com, which distributes it commercially. Section 7 is therefore not independent, and is confined to claims about what is constructible. The classification in Section 3 and the argument in Section 4 stand or fall without it; a reader who discards the case study entirely should still be able to accept or reject the rest on the evidence given.

10 Conclusion

Graph frameworks say execution follows the graph. The reversal is that the graph follows execution, and it sounds smaller than it is.

Sorting a framework's primitives into three piles is what makes the difference concrete. One pile has had an owner since the 1970s and does not need a second implementation above it. One pile is a genuine gap the kernel never filled, and durable execution engines are the right place to fill it — underneath, as substrate, which is where Hydra said mechanism belongs. The last pile is not infrastructure at all. Every node in it records a moment when a model could not be trusted with something. Each of those moments passes; the register of them does not empty, because rising capability is spent on wider scope. The class churns, and a churning thing written in the same file as a socket is what Hydra warned against.

The mechanical graph went below the line half a century ago. The trust-compensating graph has no line to go below, because there is nothing stable for a substrate to own.

Underneath all three piles is one move, and it is the one to keep: stop classifying nodes by what they do and classify the information inside them by where it comes from. Which reduces to one question. Could this have been reconstructed from a record of a run that already happened? Running produces the operating system's half for free. It produces a journal but not the boundaries the journal records. It produces nothing at all about a decision rule for inputs that never arrived. So the recommendation is not that agent systems should have no written graph, and never was. It is that they should have two, split where that question splits them: write what cannot be recovered, and let the rest be produced by running — as it already is, by an operating system that has been keeping that graph since before any of these frameworks existed. A call graph, a flame graph and a process tree are worth reading precisely because nobody wrote them. An agent's execution deserves the same treatment: consulted about as often as the process table, which is to say when something has gone wrong and you want to know what happened.

References

[1] M. E. Conway. How do committees invent? *Datamation*, 14:28–31, April 1968.

[2] DBOS, Inc. Durable execution for building crashproof AI agents. Technical blog, 2025. https://www.dbos.dev/blog/durable-execution-crashproof-ai-agents

[3] E. A. Feigenbaum. The art of artificial intelligence: Themes and case studies of knowledge engineering. In *Proc. 5th International Joint Conference on Artificial Intelligence*, volume 2, pages 1014–1029, 1977.

[4] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proc. SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982. *SIGPLAN Notices* 17(6). doi:10.1145/872726.806987

[5] B. Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016. doi:10.1145/2909476

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. doi:10.1016/0167-6423(87)90035-9

[7] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors. *Building Expert Systems*. Addison-Wesley, Reading, MA, 1983. ISBN 0-201-10686-8.

[8] 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 Conference on Computer Systems*, pages 59–72, 2007. doi:10.1145/1272996.1273005

[9] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004. doi:10.1145/1013208.1013209

[10] LangChain, Inc. LangGraph documentation: state, nodes, edges, checkpointers, interrupts and commands. Accessed 2026. https://langchain-ai.github.io/langgraph/

[11] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. 5th ACM Symposium on Operating Systems Principles*, pages 132–140, 1975. doi:10.1145/800213.806531

[12] D. Harms-Merbitz. Six levels of delegation: A per-task taxonomy of human–AI autonomy and the boundary where the operator's understanding stops being refreshed. Companion preprint, 2026.

[13] M. D. McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, July–August 1978.

[14] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[15] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[16] Temporal Technologies. The definitive guide to durable execution. Technical documentation, 2025. https://temporal.io/blog/what-is-durable-execution

[17] K. Wolstencroft et al. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013. doi:10.1093/nar/gkt328

[18] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652, 2024. arXiv:2405.15793.

[19] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. arXiv:2210.03629.