Specification of the Distinction-First Pipeline: Continuation, Accretion, and Externalized Memory in a Local-Model Essay-Writing System

Flyxion
Independent Researcher

July 12, 2026

Abstract

This essay specifies a nine-stage pipeline for producing chapter-length LaTeX prose with a locally hosted language model, and gives the rationale and full mechanics of every stage. The system is organized around a single claim: its purpose is continuation, not generation. Every stage exists to treat something already established — a distinction, a definition, an intent, an earlier chapter's conclusion — as a constraint on what a later stage may produce, rather than to produce content from an unconstrained space. This follows the same commitments that organize the author's broader theoretical program: history before state, repair before correctness, admissibility before prediction. It also follows from nine more specific, independently observed habits in how the author actually writes: essays begin from a distinction judged worth preserving rather than from a thesis; writing functions as a discovery process rather than a report of something already known; memory is externalized into structure rather than held in mind; a corpus grows by accretion, one paper becoming a chapter becoming a monograph becoming a sequence; there is a recurring search for what survives a transformation rather than a fixed claim proved for its own sake; recurring criticism is promoted into permanent mechanism rather than treated as a local fix; concrete examples function as witnesses of an already-established abstract structure rather than as its motivation; long works grow by discovering a missing dependency and writing it as the next chapter, rather than by planning a fixed table of contents in advance; and underneath all of this sits a persistent concern with how a large structure maintains its identity while changing, as opposed to simply producing more of it. Nine stages implement these commitments: a distinction-extraction stage that replaces thesis-first outlining; an exploratory drafting stage in which the main claim crystallizes only at the end; an admissibility gate that tests for a locatable distinction rather than a locatable thesis; a dependency-discovery stage that queues missing concepts as new chapters instead of patching over them; a resonance-ranked, pin-overridden, ledger-augmented corpus cross-referencing stage; a critique-revision loop that distinguishes genuine convergence from mere stalling and checks explicitly for a named invariant; a witness-finding stage that constructs concrete instances of the draft's abstract structure only after that structure is finished; and a fidelity audit against the original distinction. A persistent ledger and a persistent critique log externalize memory across runs, and a companion driver script accretes a small graph of chapters by recursively processing whatever dependencies a given chapter discovers it needs. Four of the nine stages were derived by taking

vocabulary from the Phoenix Protocol, a wave-interference memory architecture under separate audit, and stripping it of its physical claims, keeping only whatever functioned as a legitimate control pattern for an iterative language-model loop. One genuine implementation bug, found through testing, is documented as an incident report rather than silently fixed, since the failure mode it represents — an identity mechanism that collapses under composition — is exactly the kind of thing this system exists to catch in prose, and very nearly failed to catch in its own source.

1 Motivation

A large interconnected research program accumulates a vocabulary faster than any single author can hold consistently in mind across hundreds of pages written over months. Terms such as admissibility, repair operator, witness, and continuation acquire precise meanings in early essays and then risk being redefined, loosened, or silently repurposed in later ones, not out of carelessness but simply because no single essay-writing session has the whole corpus in view. A tool that assists with drafting can either ignore this problem, in which case it accelerates the production of prose while doing nothing about the risk of terminological drift, or it can be built to actively check new material against the existing corpus, against the author's own past conclusions, and against the author's own stated intent, as part of its ordinary operation. This pipeline takes the second approach throughout.

A second and more specific motivation concerns the actual shape of how these essays get written, which was reconstructed by reading across the author's essays, monographs, code repositories, and the way the author has discussed writing over many conversations, rather than by reasoning abstractly about what a good drafting tool should do. That reading surfaced nine recurring habits, true of the author's writing long before any of it was language-model-assisted, that a generic draft-critique-revise loop does not capture on its own.

Essays do not begin from a thesis. They begin from a distinction judged worth preserving — a repair operation, a difference between storage and trace, a concern about reconstruction versus recovery — that looks, on its own, too small to support a paper. The thesis is what the essay finds by taking that distinction seriously at length, not what it starts from. Writing itself functions as a discovery process rather than a report of something already known: the essays read as "I suspect X, let's see what survives formalization" rather than "I know X and will explain it," which is why they typically accumulate definitions, propositions, and examples before arriving at anything that would traditionally be called the main claim. Memory is repeatedly externalized into structure rather than held in the author's head: theorem environments, chapter hierarchies, named primitives, visible draft histories, and large collections of essays rather than one continuously growing document all serve this function. There is a strong tendency toward corpus construction rather than essay production: a single paper becomes a chapter, a chapter becomes a monograph, a monograph becomes part of a sequence, and terminological drift matters enormously for a structure like that in a way it would not for an isolated essay. There is a recurring search for invariants — what survives reconstruction, projection, correction, compression, memory loss, a change of coordinates — in place of theorems proved for their own sake. Criticism is repeatedly promoted into architecture rather than treated as a local correction: a chapter

that drifts from its intent does not get fixed once, it produces a fidelity check that runs on every future chapter; a revision loop that stalls does not get manually interrupted once, it produces a general stall detector. Examples function as witnesses of an already-established abstract structure rather than as the motivating case that produced it: the abstract machinery typically comes first, and significant effort goes into finding or constructing examples that demonstrate it afterward. Long works grow by accretion rather than advance planning: one chapter reveals a missing dependency, the dependency becomes the next chapter, which reveals another missing dependency behind it, so that a three-hundred-page monograph looks less like a tree planned from the root down and more like a graph that gradually reveals its own hidden nodes. And underneath all of this sits a persistent distinction between generation and continuation — between producing new objects and understanding how a large structure maintains its identity while changing over time — that shows up in the author's theoretical work, software workflows, critique loops, treatment of memory, and now in this pipeline's own architecture.

This pipeline is an attempt to build a system that matches that description mechanically, stage by stage, rather than one that merely drafts prose and critiques it.

2 Three Levels of Description

Before proceeding, it is worth separating three distinct levels at which the claims in this essay operate, since later sections move between them without always announcing the shift, and conflating them would misstate what has actually been established.

The first level is the concrete shell implementation: a specific script, a specific configuration file format, specific prompt templates piped to a specific local model over standard input. Claims at this level are falsifiable by reading the source or running it, and are exactly as general as the code, no more.

The second level is a general architecture for language-model-assisted writing: the pattern of separating drafting from critique, gating cheaply before revising expensively, tracking a scalar signal to distinguish convergence from stalling, discovering missing dependencies rather than patching around them, and auditing a finished artifact against original intent rather than only against intermediate critique. This level is a claim about a reusable design, not about this particular script; the shell implementation is one instance of it.

The third level is methodological: a claim about theoretical research programs generally, and about this author's writing process specifically, namely that a corpus accumulates vocabulary, argumentative commitments, and structural dependencies that later writing ought to be checked against, discover, and declare its relationship to, rather than being free to override, ignore, or invent silently. This level is the weakest to defend and the one this essay is least entitled to assert broadly, since it is argued here from a single case rather than established in general. Where the essay speaks in this register it should be read as a hypothesis about this pipeline and about programs and authors structured the way this one is, not as an established claim about writing in general.

3 Overview of the Nine-Stage Pipeline

Stage	Purpose
1. Distinguish	Extract the seed distinction (A vs B) and the invariant question it raises from raw, possibly messy notes.
2. Outline	Build a section skeleton from the distinction, with the final section explicitly marked as an unresolved "Crystallized Claim."
3. Draft	Write the chapter as an exploration: definitions and propositions accumulate; the main claim is written last, in the Crystallized Claim section, and may differ from anything anticipated in the outline.
4. Ignite	Admissibility gate testing for a locatable distinction, not a locatable thesis, plus definitions-before-use and populated sections.
5. Dependencies	Scan the draft for concepts used but not defined and not already in the vocabulary; queue each as a new seed for a future chapter rather than patching over the gap.
6. Rise	Resonance-ranked corpus cross-reference, with a pinning override for under-repeated foundational terms, plus a persistent ledger of every prior chapter's distinction.
7. Persist	Critique-revision loop tracking a severity score across rounds, distinguishing convergence, dissolution (stalling), and exhaustion (oscillation); critique checks explicitly for a named invariant and logs its category tags to a cross-run log.
8. Witness	Find or construct concrete examples instantiating each formal object in the finished draft, after the abstract structure is settled, not before.
9. Audit	Compare the finished draft against the original seed distinction for drift, then update the ledger and finalize the LaTeX document.

A companion driver, discussed in Section 7, runs stage 5's output back through the whole pipeline recursively, which is what turns a single chapter-writing tool into something that accretes a small graph of chapters from one seed.

4 The Distinction, Not the Thesis

The first stage exists to correct a specific mismatch between how a generic outlining stage works and how these essays actually start. A generic outlining stage asks a model to sketch an argument's structure, which presupposes that an argument — something with a thesis — already exists to be sketched. That is not how these essays begin. They begin from something smaller and more specific: a distinction that on its own looks too small to support a paper. The thesis is what the essay finds by taking that distinction seriously at length, not what it starts from, and an outlining stage that demands a thesis up front is asking for something that, for this kind of writing, does not yet exist at the point outlining happens.

The Distinguish stage is deliberately narrow. Given raw notes, it does exactly one thing: name the distinction underneath them, as an A-versus-B contrast, and name the invariant question that distinction raises — what survives some transformation, reconstruction, projection, or correction, in a way that makes the distinction matter. It is explicitly instructed not to summarize the notes and not to produce anything resembling an outline; those are separate stages with separate objectives, and folding them together would let the model

reach for a premature thesis while nominally only summarizing. The output is two lines, machine-parseable, consumed by every subsequent stage:

Listing 1: Distinguish stage output format

```
DISTINCTION: repair vs erasure
INVARIANT_QUESTION: What survives reconstruction from partial witnesses?
```

Everything downstream is built from this pair rather than from the raw notes directly. The outline is built from the distinction with its final section explicitly marked `TBD`. The draft is instructed to explore the distinction rather than argue toward a conclusion. The admissibility gate checks for the distinction's presence, not a thesis's. The audit, at the end, checks fidelity to the distinction, not to a thesis that was never supposed to exist yet at the point the seed was written.

5 Exploratory Drafting and the Crystallized Claim

The draft prompt is built to match the discovery-over-reporting pattern directly rather than to produce polished prose as quickly as possible. It instructs the model to accumulate definitions and propositions that follow from taking the seed distinction seriously, explicitly forbids opening with a pre-announced conclusion such as "this chapter argues that," and requires a final subsection titled *Crystallized Claim* that states, in one paragraph and written last, whatever the preceding exploration actually arrived at. This final section is permitted, and expected, to be narrower than or different from anything the outline anticipated, since the outline itself was built without a thesis to outline toward in the first place.

Design Principle 1. A drafting prompt that asks a model to write toward a conclusion already assumes the conclusion is known. A prompt whose only fixed point is a distinction, with the conclusion deferred to a clearly marked final section, mechanically reproduces writing as discovery rather than writing as report, at the cost of the draft having no guaranteed argumentative shape until the very last section is reached.

This has a consequence for how every later stage treats the draft: nothing before the Crystallized Claim section may be assumed to state the chapter's actual position. This is why the admissibility gate in Section 6 does not require a thesis in the opening, and why the audit in Section 13 compares the finished draft to the seed distinction rather than to any thesis stated partway through.

6 The Admissibility Gate

Before any revision effort is spent on a draft, the pipeline runs a minimal gate against it. This gate is deliberately cheap, blunt, and narrow, and its purpose is economy rather than quality control in the usual sense. The critique-revision loop described in Section 10 is comparatively expensive, running several rounds of adversarial review against a document that may run to several thousand words. Spending that effort on a draft that never had a distinction to explore in the first place wastes it; no amount of critique-and-revise cycling

produces a chapter from an outline that was never coherent to begin with, since the loop is designed to sharpen an existing exploration, not to originate one. The gate is therefore placed before the expensive stage and given the authority to reject rather than merely warn.

What the gate checks is detectability, not quality, and this distinction is worth stating precisely since it is easy to misread the checks below as an evaluation of the draft's merit, which they are not. The gate asks whether the system can locate a distinction-like object, a term defined before use, and a populated section ending in a Crystallized Claim — not whether the distinction is interesting, the definition is precise, or the argument in the populated section is sound. A draft with a banal or ultimately unpromising distinction passes this gate exactly as readily as a draft with an excellent one, provided the distinction is stated plainly enough to be located. This is intentional: judging the quality of an exploration is exactly the kind of task the critique-revision loop exists to do at length, with multiple rounds and an adversarial posture, and folding a quality judgment into a single cheap gate call would either make the gate expensive or make it unreliable.

Concretely, the gate checks four things. First, whether there is an identifiable distinction being explored — some contrast or pair the draft is actually working with — visible somewhere in the first section, even where no conclusion has yet been drawn from it, since the draft is explicitly permitted not to draw one there. Second, whether every technical term used before it is formally introduced is in fact defined somewhere in the draft, even informally. Third, whether the draft has more than one section with actual argumentative content rather than empty headers. Fourth, whether the draft in fact ends with a Crystallized Claim section, since a draft that never reaches one has failed to do the one thing the exploratory structure was for, no matter how good its middle sections are. A draft that fails any of these is regenerated, up to a small configurable number of retries, before the pipeline gives up and asks for a better seed rather than proceeding.

Listing 2: The admissibility check, stated as a predicate

```
def admissible(draft) -> bool:
   return (
      has_locatable_distinction(draft.first_section) # not: has thesis
      and all(is_defined_before_use(t) for t in draft.terms)
      and count(sections_with_content(draft)) > 1
      and has_crystallized_claim_section(draft)
   )
# Not tested: whether the distinction is interesting, or the argument sound.
```

7 Dependency Discovery and Chapter Accretion

This is the largest single mechanism in the pipeline, and it targets the pattern of chapter accretion directly: long works in the author's corpus are not planned as a fixed table of contents in advance, but grow by writing one chapter, discovering that chapter depends on a concept that has not itself been established, writing that concept as another chapter, and repeating, so that a large monograph looks less like a tree planned from the root down and more like a graph that reveals its own missing nodes as it is written.

The Dependency stage implements the discovery half of this directly. After a draft passes the admissibility gate, a separate model call reads it specifically for unbuilt foundations rather than for prose quality: technical terms used in a load-bearing way, not already in the established vocabulary list, and not themselves defined anywhere in the draft. Each such concept is a missing dependency — something that would need its own chapter before this one is fully grounded, even though the current draft may otherwise be fine as exploratory writing. Each one is reported with a one-sentence guess at what a chapter establishing it would need to cover, and each is written out as a new seed file rather than being flagged as a problem to fix in place:

Listing 3: A queued dependency seed file

```
CONTINUES: repair vs erasure
gyrofoil measure -- needs a chapter establishing its formal
definition and role in repair.
```

The `CONTINUES:` line matters and is discussed further in Section 15: the dependency is written down as continuing the chapter that spawned it, not as a free-floating new topic, preserving the provenance of why it was queued at all.

A companion script implements the accretion half. It runs the pipeline on an initial seed, and whenever that run queues a dependency, recursively runs the pipeline on that dependency as well, which may itself queue further dependencies, until either the queue is empty or a configured depth cap is reached. A processed-set guard prevents infinite cycles if two chapters end up depending on each other, or the same gap gets queued twice under different phrasing.

Listing 4: The accretion driver, simplified

```
def run_one(seed, depth):
   key = identity(seed)
   if key in processed: return # cycle guard
   processed.add(key)
   if depth > MAX_DEPTH:
      return # leave queued, don't process
   pipeline_run(seed) # may queue new dependencies
   for dep in queued_by(seed):
      run_one(dep, depth + 1)
```

7.1 An Incident: Identity Collapsing Under Composition

Incident 1. During testing of the accretion driver, a real bug surfaced that is worth documenting rather than silently patching, since the failure mode is a small instance of exactly the kind of thing this whole system exists to catch — an identity that fails to survive a transformation.

The pipeline's slug function, used to turn a chapter's title into a filesystem-safe identifier, truncated its output to forty characters. This was harmless for a single chapter, where the truncated slug only ever needed to disambiguate a directory name already made unique by an attached timestamp. It was not harmless for dependency filenames, which are built

by prefixing a parent chapter's slug onto a child's: `parent-slug--child-term.seed`. Once a parent's own slug already reached the forty-character cap, appending anything to it and re-truncating discarded the appended part entirely, so a child's filename collapsed onto its parent's. In practice this meant a second-generation dependency was written to the exact path its parent occupied, silently overwriting it, and the accretion driver — which tracks which files it still needs to move once processed — attempted to move a file that a nested recursive call had, unbeknownst to the outer call, already moved a moment earlier, producing a file-not-found error several stages removed from the actual cause.

The fix was not a patch at the point of failure but a second, deliberately unbounded identity function used everywhere identity needs to survive composition, while the truncated version was kept for the one purpose it was always safe for:

Listing 5: Two identity functions with different survival guarantees

```
def slug(s): # truncated -- safe only where a timestamp
   return dashify(s)[:40] # already disambiguates, e.g. OUTDIR names

def slug_full(s): # untruncated -- required wherever identity
   return dashify(s) # must survive being prefixed onto again
```

The general lesson, stated at the level of Section 2's third register: an identity mechanism that is safe in isolation is not automatically safe under composition, and a system that recursively builds identities out of other identities — exactly what chapter accretion does — needs to ask, for every identity function it uses, whether that function is closed under the composition the system actually performs, not just correct for a single, non-recursive use. This is the same question the pipeline asks of prose — what survives a transformation — applied for once to its own source code rather than to a chapter draft.

8 Vocabulary as a Controlled Resource

The pipeline treats the project's established terminology as a controlled vocabulary rather than as free text. A configuration file lists every term that has an established meaning elsewhere in the corpus — named frameworks, together with core primitives such as admissibility, distinction, witness, and continuation. This list is injected verbatim into every prompt the pipeline constructs, with an instruction to use these terms consistently and not to silently redefine or repurpose them. The list is authored by hand and grows only when a term becomes load-bearing somewhere in the corpus; the pipeline does not infer or invent vocabulary on its own.

Design Principle 2. A system that assists with writing inside an established theoretical program should be given the program's vocabulary as an explicit constraint, not left to infer it from context on each invocation.

This is a modest mechanism, but it changes what the critique stage in Section 10 is capable of catching. Without an explicit vocabulary, a critique pass can only evaluate a draft against itself: internal contradictions, dropped threads, unsupported claims. With the vocabulary supplied, the critique pass can additionally check a draft against everything else

the vocabulary has meant before, which is a different and in some ways more important kind of error to catch, since a single essay can be internally flawless while still contradicting the definition of the same term two essays earlier.

Three claims are being made together here that are worth separating. At the implementation level, the claim is only that a configuration file lists terms and that this list is injected into prompts; this is verifiable by reading the code. At the design level, the claim is a recommendation — that any system assisting with writing inside an established body of work should be given that vocabulary explicitly rather than left to infer it — defensible on the general grounds that inference from context is unreliable and an explicit list is cheap. At the epistemic level, the claim is that meaning is actually preserved across a corpus by this mechanism, which this essay has not established and asserts only as a working hypothesis motivating the design.

9 Externalized Memory: Resonance, Pinning, and the Ledger

If a directory of existing essays is supplied, the pipeline performs a cross-referencing pass before every critique. For every vocabulary term that appears in both the configuration file and the current draft, the system searches the existing corpus for other occurrences of that term and counts them. Terms are then ranked by this count and only the highest-ranked handful are actually pulled into the critique prompt, together with a few lines of surrounding context from wherever they occur in the corpus. This ranking step exists because an unranked cross-reference pass does not scale: a corpus of any size produces far more matching context than a language model's working context can usefully absorb, and naively including all of it drowns the signal the critique stage actually needs.

Ranking by occurrence count is a simple proxy for how load-bearing a term is, and it is a proxy that costs nothing beyond a directory search, rather than requiring any semantic understanding of the corpus itself. It is also known to be wrong in one specific and important direction: some foundational concepts are assumed throughout a corpus rather than restated in it, and so appear rarely despite being the most load-bearing terms present, while a term can be repeated often because it is locally fashionable in a particular stretch of writing without being central to the program as a whole. Frequency and importance need not coincide, and a ranking step built on frequency alone would silently favor the fashionable term over the assumed one.

This failure mode is addressed directly rather than merely noted. The vocabulary configuration file supports a second category of entry, a *pinned* term, marked with a leading `!` in the source file and stripped of that marker before the term is shown to the model. Pinned terms bypass the occurrence-count ranking entirely: if a pinned term appears in the current draft and has at least one hit anywhere in the corpus, it is included in the critique context regardless of where it would have fallen in the ranked list. Ranking-by-frequency thus governs the unpinned majority of the vocabulary, where frequency is a tolerable proxy, while pinning lets the author assert importance directly for the specific terms most likely to violate the proxy.

Listing 6: Resonance ranking with pinning override

```
for term in vocabulary:
```

```
    if term not in draft: continue
    count = occurrences_of(term, corpus)
    if count > 0:
        scored[term] = count

selected = top_k(scored, k = RESONANCE_TOP_K) # rank-based

for term in pinned_vocabulary:
    if term in draft and term in scored and term not in selected:
        selected.add(term) # pin-based override

for term in selected:
    context[term] = surrounding_lines(term, corpus, limit = 10)
```

A second, independent source of context is consulted at the same stage: a persistent ledger file, external to any single run, that accumulates one row per completed chapter across the pipeline's entire history.

Listing 7: Ledger schema (tab-separated)

```
slug title distinction continuation outcome timestamp
```

This directly targets the externalize-memory-into-structure pattern: rather than relying on the corpus directory's raw prose to carry continuity between chapters, or relying on the author to remember what an earlier chapter established, the pipeline maintains its own structured record of what every prior run concluded, and every corpus-scan stage reads it unconditionally, prepending a summary of prior chapters' distinctions to the context handed to the critique stage:

Listing 8: Ledger excerpt as injected into critique context

```
### Ledger -- prior chapters on record (externalized memory)
- Repair vs Erasure -- distinction: repair vs erasure (ROOT, CONVERGED)
- Gyrofoil Measure -- distinction: ... (CONTINUES: repair vs erasure, ...)
```

The ledger is deliberately flat and append-only rather than queryable or structured beyond six columns. This is a level-two design choice in the sense of Section 2: the claim is that externalizing memory into any inspectable structure, however simple, is better than not externalizing it at all, not that this particular six-column schema is the correct one. A richer ledger — one that also recorded, say, every definition introduced per chapter rather than only the chapter's overall distinction — would strengthen the mechanism further; the current schema was kept minimal deliberately, to get a working version of the pattern in place before elaborating it.

10 Persistence, Convergence, Stalling, and the Invariant Check

The critique-revision loop is bounded by a maximum number of iterations, but is not designed to run for that many iterations unconditionally. Each critique pass reviews the current draft against seven checks: unsupported or overstated claims; structural problems such as weak transitions or redundant sections; hand-wavy reasoning presented as if it

were rigorous; missing definitions for terms used before they are defined; terminology drift against the vocabulary and corpus excerpts assembled in Section 9; theorem or proposition statements that do not earn their status, such as an unproven conjecture stated as a theorem; and, targeting the invariant-search pattern directly, whether the draft explicitly names something that survives the transformation, reconstruction, or correction it discusses, or only asserts claims without ever asking what is preserved. A draft can pass every other check and still fail this seventh one, if it never actually engages the question the Distinguish stage raised as its `INVARIANT_QUESTION` in the first place.

Each critique pass is required to end with three lines: an integer severity score from zero to ten, a qualitative verdict, and a comma-separated list of which of the seven checks were flagged that round. The pipeline tracks the severity score across rounds. Two conditions cause the loop to stop early. If the severity reaches zero, or the critique explicitly states that no substantive issues remain, the loop has converged and the current draft is accepted as final. If the severity instead fails to improve for a configurable number of consecutive rounds, the loop is judged to have stalled and stops as well, but with a different message: the last draft before stalling is reported as needing a human read, rather than being silently accepted as though the lack of further critique meant the draft had become adequate.

Definition 1. A critique-revision loop is said to *dissolve* at round n if the severity score has been non-decreasing for k consecutive rounds ending at n, where k is a configured patience parameter, and the loop has not separately converged. Dissolution is a distinct outcome from convergence and is reported as such.

The severity score deserves an explicit caveat. It is produced by asking the critiquing model to self-report an integer, and nothing in the pipeline calibrates that integer against any external standard. A drop from eight to six carries no guarantee of being comparable, in the amount of actual improvement it represents, to a drop from four to two; the same numeric gap could correspond to very different amounts of real progress depending on where in the scale it occurs, or on how the critiquing model happens to be distributing its scores that round. The score is a heuristic control variable, used only to distinguish a downward trend from a flat one, and the pipeline is deliberately built to ask only that coarse question of it. If the model omits the required severity line entirely, the score is not left undefined or treated as zero; it defaults to five, a value chosen specifically so that a missing score can neither register as convergence nor as a repeated plateau on its own, forcing the loop to keep going rather than exit on a signal it never actually received.

Listing 9: The persistence loop as a state machine

```
prev = infinity; stall = 0
for round in 1..MAX_ITERATIONS:
   critique = critique(draft)
   sev = parse_severity(critique) or 5 # fallback: never trust silence
   if sev == 0 or says_no_issues(critique):
      return CONVERGED, draft
   stall = stall + 1 if sev >= prev else 0
   prev = sev
   if stall >= DISSOLVE_PATIENCE:
      return DISSOLVED, draft # last draft before stall, flagged
```

```
    draft = revise(draft, critique)
return EXHAUSTED, draft # hit MAX_ITERATIONS without either
```

Three distinct exit states follow from this, not two. A loop that converges has a critique attesting the draft is done. A loop that dissolves has a critique attesting the draft is stuck, with the specific unresolved issues still on record in the last critique file. A loop that merely exhausts its iteration budget without doing either — possible if severity keeps oscillating rather than trending flat or down, so neither condition ever triggers — has neither attestation, and this is reported as a third kind of ambiguity, deliberately distinguished from convergence rather than folded into it. An unbounded loop that merely runs until it exhausts its iteration budget would conflate several very different situations under a single stopping condition: a draft that reaches the final round because it converged three rounds earlier and the loop simply had nothing further to do is in a different state than a draft that reaches the final round while still being told the same handful of unresolved problems it was told rounds earlier, and both are different again from a draft that oscillated the whole way through. Reporting all three as "finished" discards information the author needs; the stall detector and the exhaustion state both exist to preserve that information rather than let a fixed iteration count paper over it.

Every critique round's severity and category tags are also appended to a persistent, cross-run critique log, distinct from the per-run critique files already written to disk:

Listing 10: Critique log schema (tab-separated)

```
slug round severity categories timestamp
```

The purpose of this log, and what it enables beyond this single run, is taken up in Section 11.

11 Turning Criticism into Architecture

The pattern targeted here is specific: rather than treating a piece of criticism as a local fix, the author's typical response is to promote it into a mechanism that makes the entire failure mode visible going forward. A chapter drifting from its intent does not get corrected once; it produces a fidelity audit that runs on every chapter thereafter. A critique loop that plateaus without improving does not get manually interrupted once; it produces a general dissolution detector. Both of those promotions are already built into this pipeline as permanent stages, described in Sections 10 and 13.

The critique log introduced in Section 10 is a first-order implementation of the mechanism that would let this promotion happen again in the future, though it is not yet the promotion itself. Recording slug, severity, and category tags for every critique round, across every chapter, across the pipeline's entire history, is what would let a later analysis ask a question no single run can ask of itself: does the same category of issue — terminology drift, missing invariant, illegitimate theorem status — recur at high severity across many chapters, rather than being a one-off problem with one draft. If it does, that recurrence is itself evidence that the corresponding check deserves to be promoted from a critique-time flag into an earlier, structural gate, the same way fidelity auditing and dissolution detection were themselves promoted from ad hoc fixes into permanent stages of this pipeline.

This essay stops short of specifying that promotion mechanism itself; the log exists, but nothing yet reads it and proposes a new gate automatically. This is a deliberate boundary rather than an oversight. Deciding that a recurring critique warrants a new permanent check is itself a judgment about what the corpus's failure modes actually are — and per Section 16, judgments of that kind belong on the human side of this system's boundary, not the automated side. What the log provides is the raw material for that judgment to be made well informed, rather than from memory of a handful of recent chapters.

12 Witnessing the Abstract Structure

The asymmetry this stage targets is specific: most writing starts from examples and generalizes from them; this author's essays typically start from an abstract structure and only then spend effort finding or constructing examples that instantiate it, with the examples functioning as witnesses of an already-established structure rather than as the motivating cases that produced it.

The Witness stage is a separate model call, run only after the critique-revision loop has finished, on whatever draft that loop produced, regardless of which of its three outcomes occurred. It is given the finished draft and instructed to treat every definition, proposition, theorem, and lemma as something requiring a witness: either an existing passage in the draft that already instantiates it concretely, or, if none exists, one newly constructed concrete example that does.

Listing 11: Witness stage output format

```
WITNESS: Definition 1 -- a torn page reconstructed from ink-transfer
residue witnesses repair preserving structure erasure would destroy.
```

This is kept as a separate, late-running pass rather than folded into drafting for a specific reason: folding it into the draft stage would reproduce exactly the example-first pattern this stage exists to avoid, by giving the model the opportunity to reach for a convenient example before the abstract structure it is meant to witness has actually stabilized through the critique-revision loop. Running it last ensures every example is checked against a structure that has already survived adversarial critique, not one still in flux.

13 Auditing for Fidelity to the Distinction

The final stage compares the finished draft against the original seed distinction, independent of anything the critique-revision loop said, and asks specifically whether the finished draft's Crystallized Claim is still a serious exploration of that same distinction, or whether it has quietly substituted a different distinction, dropped the original contrast, or hedged it into vagueness over the course of revision. This check exists because the critique loop, by construction, only ever compares a draft to the critique of that draft; nothing earlier in the pipeline ever looks back at the seed once outlining has finished. Revision is supposed to sharpen an exploration, not replace it, and a loop that only measures forward progress against its own critiques has no way of noticing if it has, cycle by cycle, argued its way to a different point than the one the author started with. The audit stage is the only point in

the pipeline where the original distinction is consulted a second time, and it runs against whichever draft the persistence loop leaves behind regardless of which of the three outcomes described in Section 10 produced it — a converged draft can still have drifted, and a dissolved or exhausted draft is audited too, since a human reading it afterward benefits from knowing whether it drifted in addition to knowing it stalled.

What counts as drift here is narrower than it might first appear, and this matters because the draft was never asked to commit to a thesis in advance. Arriving at a claim no one, including the author, could have predicted from the seed distinction is expected and correct behavior for an exploratory draft, not evidence that anything went wrong. What would count as drift is more specific: quietly substituting a different distinction than the one the chapter started with, dropping the original contrast entirely rather than resolving or narrowing it, or hedging it into vagueness so that the final claim no longer engages the distinction at all. The audit prompt is written to make this distinction explicit to the model reviewing it, since a naive comparison between seed and final draft would otherwise flag every successful exploration as suspicious simply for not being predictable in advance.

14 Relation to the Phoenix Protocol

Four of the nine stages above — the admissibility gate, the ranked retrieval step, the persistence loop, and the fidelity audit — are named after, and structurally derived from, the lifecycle of the Phoenix Protocol: a wave-interference memory architecture, developed as part of a separate and unrelated project, in which a signal is admitted into memory subject to an accessibility threshold, persists across a fixed heartbeat rhythm while its amplitude decays, is retrieved through a resonance calculation between a query frequency and the signal's stored frequency, and is finally checked for reconstruction fidelity against its original form. That architecture is itself the subject of an extended separate audit, one which treats it as a speculative computational system to be reconstructed sympathetically, critiqued rigorously, and selectively reinterpreted in terms compatible with the author's own framework, rather than accepted or dismissed wholesale.

The borrowing here is of that same kind, applied to a much smaller object. Nothing in this pipeline depends on wave mechanics, on a fixed heartbeat frequency, or on amplitude as a physical or even metaphorical quantity. What survived the translation was the shape of four operations — gate, rank-and-retrieve, persist-with-a-stopping-rule, audit — because each one turned out to name a real and useful control pattern for an iterative language-model loop, independent of the physical vocabulary it originally arrived in. Renaming a hit-count-based retrieval ranking as resonance, or a severity plateau as dissolution, costs nothing and risks a great deal if the names are mistaken for claims about how the system works. They are not; they are labels kept for continuity with the architecture that suggested the shape of the solution, attached to mechanisms that could equally well have been named after their literal function.

Design Principle 3. A vocabulary may be borrowed from a speculative or unproven system without borrowing that system's claims, provided every borrowed term is cashed out, on arrival, as an operation whose behavior can be stated and tested without reference to the system it came from.

The remaining five stages — Distinguish, the exploratory draft's Crystallized Claim, Dependencies, Witness, and the critique log — carry no Phoenix Protocol vocabulary at all. They were derived directly from the nine writing patterns described in Section 2's motivation, with no intermediate translation step through any other system. The Phoenix-derived stages and these five are therefore independent in origin, and their coexistence in one pipeline is a claim only about what turned out to be jointly useful, not about any deeper relationship between the two sources.

15 The Continuation Declaration as Human Decision Surface

The pipeline drafts, checks internal and corpus-level consistency, checks against a persistent ledger of prior chapters, revises against critique, checks for a named invariant, detects its own stalling, discovers and queues missing dependencies, finds witnesses for its own abstract structure, and checks for drift from original intent. What it does not do, at any stage, is decide what a chapter is about, which of several candidate primitives deserves to be treated as foundational, whether a queued dependency is actually worth writing, whether a given chapter is a root contribution or a continuation of existing work, or how two independently developed concepts ought to relate once both exist. Those decisions constitute what might be called the human decision surface of the larger research program, and every one of them either happens before a seed ever reaches the Distinguish stage, or happens when a human reads a finished chapter, a queued dependency, or a ledger entry and judges it.

Chapter accretion, described in Section 7, is the part of this pipeline most likely to have quietly crossed that boundary, since a system that automatically discovers and queues new chapters is one plausible reading away from a system that decides, on its own, what the corpus needs next. The boundary is preserved by a specific, narrow mechanism rather than by general good intentions: a seed file's optional first line may declare `ROOT` or `CONTINUES: <terms>`, and this declaration is read, never inferred. The dependency stage populates the `CONTINUES:` line of every queued seed with the chapter that spawned it, so provenance is never lost, but it does not decide whether that queued concept is actually foundational, actually worth its own chapter, or actually correctly scoped — those questions remain open until a human either runs the queued seed through the pipeline as written, edits it first, or discards it without running it at all. A run with no continuation declaration at all proceeds — the pipeline does not block on it — but is logged as `UNSPECIFIED` rather than having the gap silently filled by a guess.

Design Principle 4. A system that discovers structure automatically (missing dependencies, recurring critique categories, resonant vocabulary) may surface that structure as a proposal, logged and inspectable, without thereby being entitled to decide the proposal is correct. The line between discovering and deciding is the line this system is required to keep on the human side of, at every stage where automation is added.

16 What the Pipeline Does Not Do

It is worth stating the resulting boundary plainly, in one place, since it is easy to lose track of across nine stages. The pipeline does not decide what a chapter is about, which of several

candidate primitives deserves to be treated as foundational, whether a queued dependency is actually worth writing, whether a chapter is a root or a continuation of existing work, how two independently developed concepts ought to relate once both exist, or whether a recurring critique category warrants promotion into a new permanent gate. It does not notice scoping errors. A critique-revision loop can be run indefinitely over a chapter built on a wrongly chosen foundational distinction and it will only ever produce a more polished chapter built on the wrong foundation; accretion can run for many recursive levels against a wrongly scoped root distinction and will only ever produce a larger structure built on that wrong scoping. Nothing in this specification is capable of noticing that the foundation or the scoping was wrong, because noticing that requires comparing the chapter, or the emerging graph of chapters, against the rest of the program's architecture — precisely the part this system was not built to hold. What it was built to hold is narrower and, within that scope, thorough: drafting, internal and cross-corpus consistency, convergence detection, missing dependency discovery, concrete instantiation of abstract structure, and fidelity to stated intent, all logged, none of it silently discarded, and all of it subordinate to decisions a human makes before, during, and after every run.

17 Conclusion: Continuation, Not Generation

Read back across the preceding sections, a single principle organizes what would otherwise be a list of unrelated engineering choices. The distinction-extraction stage of Section 4 exists so that a chapter is answerable to something the author already judged worth preserving, rather than to a thesis invented for the occasion. The vocabulary constraint of Section 8 exists so that new writing is checked against what earlier writing already meant. The ledger of Section 9 exists so that a corpus-scan reads the pipeline's own accumulated history rather than starting from a clean slate on every run. The dependency-discovery mechanism of Section 7 exists so that a missing foundation is queued to be actually built rather than patched over in the chapter that needed it. The invariant check of Section 10 exists so that no chapter is allowed to claim anything without having said, somewhere, what survives the transformation it concerns. The audit of Section 13 exists so that the finished chapter can be checked against the intent that started it, not only against its own intermediate critiques. The continuation declaration of Section 15 exists so that the pipeline's own growth is honest about whether it is starting something new or extending something already underway. In every case, something already committed to functions as a constraint on what may legitimately come next, and none of the nine stages generates content from an unconstrained space.

This is a level-three claim in the sense of Section 2, and it should be read as a hypothesis about this pipeline and about programs structured like it, not as an established fact about writing systems in general. But it is a hypothesis that follows directly, at the level of mechanism rather than metaphor, from commitments that organize the author's broader theoretical program: history before state, repair before correctness, admissibility before prediction. Those commitments say, at the level of a research program, that what has already happened constrains what is admissible next, that fixing an error takes priority over asserting a clean answer, and that whether a continuation is allowed at all takes priority

over predicting which continuation is likeliest. This pipeline is the same claim applied one level down, to the mechanics of producing a single chapter and, through accretion, a small graph of them: what has already been written constrains what a draft may claim, fixing a draft's problems takes priority over letting it stand, and whether a draft is even admissible for revision takes priority over predicting how good a revision of it might be. The pipeline does not implement history-before-state, repair-before-correctness, or admissibility-before-prediction as formal objects; it implements the much smaller, purely mechanical shadow those commitments cast on the specific problem of drafting prose, with a language model in the loop, inside a corpus that is still growing.

One genuine bug in the mechanism meant to let these nine stages compose was found by testing during the development of this pipeline, rather than assumed away, and is documented in Section 7.1 rather than quietly patched: an identity function safe for a single chapter collapsed once dependency chains composed it with itself repeatedly, and the fix required asking, of every identity the system constructs, whether it survives the specific transformation — being prefixed onto again, recursively — that this system actually performs. That the bug was found this way, in the pipeline's own source rather than in someone else's prose, is itself a small instance of the larger claim: a system built to ask what survives a transformation eventually has to ask that question of itself, and answering it honestly, in public, in the specification rather than only in a private fix, is what keeps continuation as a real commitment rather than a stated one. The pipeline continues a corpus; it does not decide what the corpus is about.