Boundary Work, Not Castles

A Study of Where Language-Model Value Is Created, Why the Work There Is Invisible, and the Case for Externalised Composition

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

Abstract

A language model transforms text into text. Nothing of value has happened until that text is wired into systems that act, and the wiring has a name older than the models: it is *articulation work* in Strauss's sense — the labour of fitting parts of a division of labour together — and it has the property articulation work always has. It is invisible, it is low-status, and it is therefore under-resourced at exactly the point where the value is created.

This paper contrasts two architectures for doing that work. One internalises composition inside a product: the agent, the framework — the castle. The other externalises composition to mechanisms that already exist: the pipe, the file, the exit code — the plumbing. The difference is not taste. In Perrow's terms the castle is interactively complex and tightly coupled where the pipeline is linear and loosely coupled; in observability terms the castle's intermediates are *rendered* where the pipeline's are *returned*, which fixes which levels of delegation an operator can occupy. We give a mechanism for why castles are built anyway — trained abstraction, the second-system effect, a demo gradient, and the economics of invisible work — and we state the conditions under which the castle is nevertheless the right structure, because a dichotomy without a failure mode is a slogan. A staffing consequence follows: the people fitted to boundary work already exist, hold operational job titles, and are not, in the main, the people currently assigned to it.

A case study shows the externalised architecture is constructible in one small tool; competing interests are declared. The claims are stated so that they can be tested, and Section 9 says how they would lose.

Keywords: human–AI interaction; articulation work; invisible work; software architecture; Unix composition; agentic systems; integration; division of labour

1 Introduction

A language model is a text-to-text transformation. The text may be code, commands, analysis, or apology; it is still text, and text does not run, deploy, bill, alert, or ship. Between the model's output and anything a business would recognise as value sits a stretch of work: taking the text and wiring it into systems that do things. That stretch is where this paper lives.

One sentence carries the rest of it:

> The value of a language model is created at the boundary where its text meets systems that act, and the work of that boundary is articulation work — invisible by nature, and therefore mis-assigned by default.

The claim has three parts, and each is doing work. *At the boundary*: the model itself is increasingly a commodity reached through an API, so the differentiating engineering is not in the model but around it — a position the machine-learning systems literature reached about the previous generation of models a decade ago [23], and which the shift toward compound systems restates for this one [30]. *Articulation work*: the sociology of work has a fifty-year-old name and literature for exactly this labour — the supra-work of fitting tasks, tools, and people together

so that a division of labour actually functions [25, 8] — and that literature's central finding is that such work is systematically deleted from formal representations of what an organisation does [24, 26]. *Mis-assigned by default*: work that does not appear on a roadmap is not resourced by one, so organisations staff the visible thing (the model, the product) and starve the joint, or — the pattern this paper is mostly about — they buy the joint back in the shape of a product, which is how the castle gets built.

In order of importance, the paper contributes:

1. **A relocation of the problem** (Section 3). The gap between model output and value is not a capability gap but articulation work, with a lineage that explains its central pathology: it is invisible, and invisible work is neither budgeted, credited, nor deliberately staffed.
2. **A structural distinction** (Section 4). Two architectures for boundary work, distinguished by one property — where composition lives — from which nearly everything else follows: testability, failure locality, coupling in Perrow's sense [17], and which levels of delegation the operator can occupy [11].
3. **A mechanism** (Section 5). Why castles get built anyway: abstraction as the developer's trained response, the second-system effect [2], a demo gradient that favours rendered intermediates, and the economics of invisibility, in which articulation work is resold as a product precisely because it cannot be seen as labour.
4. **The other side** (Section 6). The conditions under which the castle is the correct structure, and the pipeline's own characteristic pathologies, including the security one. A dichotomy that cannot lose is not an argument.
5. **A staffing consequence and a constructibility case** (Sections 7–8). The people already fitted to boundary work hold operational job titles; a small tool shows the externalised architecture is buildable; Section 9 states the studies that would test all of the above.

We are explicit at the outset about what this paper is not. It contains no user study, no deployment telemetry, and no controlled comparison of the two architectures. An earlier version of this argument circulated as a company memorandum, hortatory in register and promotional in purpose; this paper is the version written to be wrong in discoverable ways. The claims are separated from the product, the cases where the argument fails are stated, and the predictions in Section 9 are ones a study could kill.

2 Related Work

2.1 Articulation work and invisible work

Strauss introduced *articulation work* as the work that makes divided labour cohere: meshing tasks, aligning schedules, repairing the plans that the formal description of the work assumed would not need repair [25]. Gerson and Star showed that this work is unavoidable in real information systems because no formal representation of an office's work is ever complete; somebody always closes the gap between the representation and the situation, by hand [8]. Star and Strauss then examined what organisations do with such work: they render it invisible, and the invisibility is not an accident but an ecology, with predictable consequences for who is credited, who is paid, and what gets designed for [24]. Suchman drew the design conclusion — systems built from the formal representation alone will be built for work that does not exist — and argued for making the actual work visible as a precondition of building for it [26]. In practitioner writing the same observation circulates as "glue work": essential, career-invisible, and disproportionately assigned

to whoever will absorb it [20].

This paper's use of the literature is direct rather than metaphorical. Wiring model output into running systems — validating it, gating it, granting it capabilities, deciding where its context lives and where its failures surface — is articulation work by Strauss's definition, and it inherits the ecology: it is absent from the architecture diagram, absent from the demo, and absent from the org chart, which is why Section 5 can treat its invisibility as an economic force rather than an oversight.

2.2 The other boundary-work

The term *boundary-work* is already taken. Gieryn uses it for the rhetorical labour by which groups demarcate their activity from adjacent activity — science from non-science, profession from craft — in pursuit of authority and resources [9]. We use the phrase literally in this paper: work performed at an interface. But the collision is kept deliberately, because Gieryn's sense explains part of our problem. The demarcation of "real engineering" from "scripting" is boundary-work in his sense, it is performed daily in software organisations, and it is one of the mechanisms by which the literal boundary work of Section 3 is made low-status and therefore left undone or done ad hoc. The reader will notice, correctly, that calling one architecture a castle is itself boundary-work in Gieryn's sense; Section 10 returns the charge.

2.3 Unix composition

The design tradition this paper leans on is stated in two sentences by McIlroy: write programs that do one thing well, and write programs to work together, expecting the output of every program to become the input of another [14, 21]. Pike and Kernighan's less-quoted companion argument matters as much here: the tradition decays precisely when tools grow options and modes that internalise what composition used to do, and the decay looks like convenience [18]. That is a 1984 description of a 2026 failure pattern. The relevance of the tradition to language models is structural rather than nostalgic: an LLM is a text-to-text transformation, which is the exact shape the pipe was built to carry.

2.4 Pathologies of premature structure

Software engineering has spent decades naming the failure this paper calls the castle. Brooks's second-system effect describes the designer's tendency to load the follow-up system with every generalisation withheld from the first [2], and his later distinction between essential and accidental complexity gives the test the castle fails: most of a framework is accident [3]. Fowler catalogues *speculative generality* — structure added for futures that never arrive — as a smell to be removed, not admired [5]; the XP tradition compresses the remedy to four letters [1]. Gall's law states the constructive direction: complex systems that work are found to have evolved from simple systems that worked, and complex systems built from scratch do not work [7]. Gabriel's "worse is better" explains why the small ugly thing spreads while the complete thing waits [6], and Foote and Yoder document where big designs actually end up [4]. None of this is about AI. All of it transfers, because the castle instinct is a property of the builders, not of the material.

2.5 Glue in machine-learning systems

Sculley et al. made the canonical observation for the previous model generation: in a real ML system the learning code is a small box surrounded by a vastly larger mass of configuration, data plumbing, serving, and monitoring, and the debt accumulates in the mass, not the box

[23]. Language models sharpen the picture. The box has moved behind a vendor's API, so for most deployments the mass is now the entire locally-built system, and Zaharia et al. observe that state-of-the-art results themselves increasingly come from compound systems — pipelines of models, retrievers, and checks — rather than from single models [30]. Both lines of work say where the engineering is. Neither says who should do it or what shape it should take, which are this paper's questions.

2.6 Agents and their record

The agent literature supplies the castle's strongest current form: systems that interleave model reasoning with tool invocation inside one artefact [29], up to repository-scale software work with little human presence [28]. The evaluation record is instructive. Kapoor et al. argue that agent benchmarks systematically overstate real-world utility by ignoring cost, reproducibility, and overfitting to the benchmark [12]. On a benchmark built from consequential simulated-workplace tasks, the strongest agent tested completed roughly a quarter of tasks autonomously at release [27]; realistic web-task benchmarks report similar gaps [31]. Outside the lab, a RAND interview study of failed AI projects found the leading causes were misunderstandings of the problem and failures of data and infrastructure integration — boundary causes, not capability causes [19] — and a widely circulated 2025 industry study reported that the large majority of enterprise generative-AI pilots showed no measurable profit-and-loss effect [15]. The industry figure's methodology has been debated and we cite it as evidence of the perceived scale of the problem rather than of its precise size. The pattern across these sources is consistent with this paper's thesis and is not explained by model capability alone.

2.7 The companion paper

This paper is the builder's-side companion to a taxonomy of operator delegation [11], which locates a boundary between operators who are updated by the work and operators who are updated by instruments that measure the work. The connection runs in both directions. The instruments of that paper — the check, the gate, the allowlist, the file — are the artefacts of this paper's boundary work; and this paper's two architectures determine whether that paper's ladder can be climbed and descended at all, because a castle that renders its intermediates instead of returning them fixes its operator on the far side of the boundary of kind whether they chose it or not.

3 The Boundary

Consider a deployed language model doing something useful: reviewing commits, triaging logs, drafting responses, filling a queue. Strip away the model call itself and inventory what remains. The remainder is boundary work, and it decomposes into five decisions that somebody makes, whether or not anybody notices making them.

Transport. Getting the right text in front of the model, and the model's text in front of the next thing that needs it — a compiler, a ticket system, a human, another model. Transport looks trivial and is where most of the hours go, a finding the ML-systems literature made before this model generation existed [23].

Validation. Deciding whether an output may take effect, and building the thing that decides: the test, the check, the gate, the exit code. Validation is the boundary's load-bearing wall, because a language model's outputs are fluent whether or not they are right, so nothing downstream can be trusted to notice on its own.

Capability. Deciding what the model may touch while it works: which commands, which files, which systems, which nothing. The capability decision is a security decision, and Section 6 argues it is the one place the castle's instinct — build a wall — is exactly correct.

Placement of state. Deciding where context, memory, and history live, and consequently who else can read, version, diff, and delete them. State placed in a file is legible to every tool the organisation already has; state placed inside a product is legible to that product.

Routing of failure. Deciding where an error surfaces, whom it interrupts, and what a person finds when they arrive. Failure routing is the difference between a stage that says `exit 1` and a narrative that says the run did not complete as expected.

Two remarks. First, these five are the same objects as the instruments and invariants of the companion taxonomy [11], seen from the builder's side rather than the operator's: whoever does the boundary work is deciding, in advance, what the operator will later be able to know. Second, every one of the five is articulation work by definition — it exists because the formal picture ("the model does the task") is incomplete, and somebody closes the gap [8]. The literature predicts what happens next: the five decisions get made, because they must, but they get made invisibly, by whoever is nearest, without budget or design review, and the organisation's official account of its AI system will contain a model and a use case and none of the above [24, 26]. The rest of this paper is about the two shapes the closed gap can take.

4 Two Architectures

There are two places composition can live, and the choice determines nearly everything else about the system.

Composition internalised: the castle. Control flow, state, and inter-step communication live inside one artefact and are reached through its features. The agent is the pure case: a loop that plans, remembers, invokes tools, and decides when it is finished, all within one process whose insides are displayed — when they are displayed — as a rendered narrative. The framework is the general case: an abstraction layer through which every model interaction must pass, whose value proposition is precisely that composition is somebody else's code.

Composition externalised: the pipeline. Control flow lives in the shell or the scheduler, state lives in files in the working directory, and inter-step communication is a text stream. The model call is one small program among others; everything that connects it to the world is an ordinary command line that the operator wrote and can read.

Figure 1 draws both, and Table 1 states the consequences side by side. Three of them deserve prose.

4.1 Coupling

Perrow's analysis of high-risk systems sorts them along two axes: interactive complexity, the degree to which components can affect one another in unplanned ways, and coupling, the degree to which a disturbance propagates before anyone can intervene. Systems that are both interactively complex and tightly coupled have accidents as a normal property, not an exceptional one [17]. The castle is that quadrant by construction: its components share a process, a memory, and a control loop, and a mis-step propagates at the speed of the loop. The pipeline is the opposite quadrant: stages interact only through a stream, files are exactly the buffers that loose coupling requires, and a disturbance stops at the first stage that exits non-zero. A language model is a stochastic

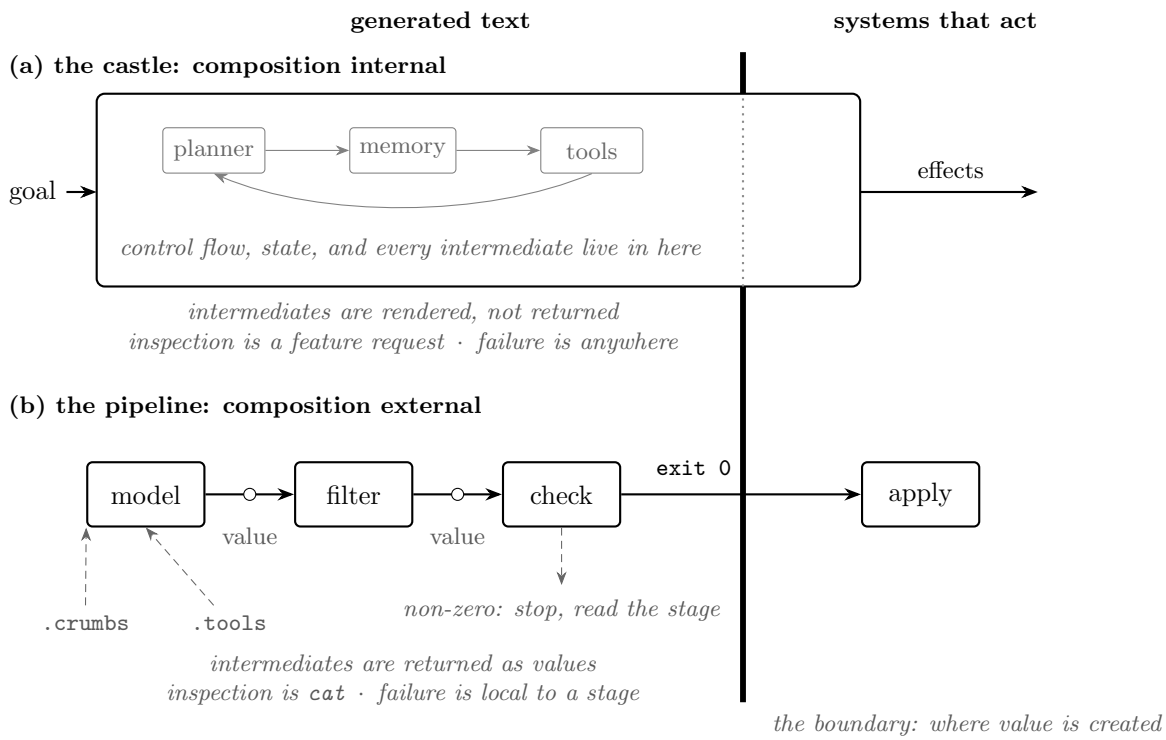

Figure 1: Two ways across the same boundary. Above, composition is internal: what crosses the line is an effect, what the operator sees of the middle is whatever the product chooses to render, and the boundary itself is still present inside the castle — merely out of sight. Below, composition is external: every joint is a value that can be captured, tested, or branched on, state is files beside the work, and the crossing is gated by an exit code. Which drawing a system is in is the thing worth knowing about it; the number of boxes is not.

component; placing one in the middle of a system raises the system's interactive complexity by itself. It is an odd moment to tighten the coupling as well, and that is what the castle does.

4.2 Observability

The companion taxonomy locates a boundary of kind in the operator's position: below it the operator is updated by the work, above it by instruments that measure the work, and a defect no instrument covers is invisible in principle rather than in practice [11]. The two architectures distribute operators across that boundary involuntarily. A castle's intermediates are renderings — prose about what the agent did — which cannot be captured, diffed, or branched on; its operator is an audience whether or not the task warranted one. A pipeline's intermediates are values, so its operator can stand anywhere: read every stage, gate on a check, or walk away and let cron run it, and — the property that matters when something looks wrong — come back down without restating the task somewhere that has never heard of it. Architecture, in other words, is delegation policy. Choosing the castle chooses the operator's side of the boundary for them, and says nothing about having done so.

	Castle (internal)	**Pipeline** (external)
Control flow	product code	a shell expression
State	a store inside the product	files in the working directory
Intermediates	rendered as narrative	returned as values
Unit of test	the whole	each stage alone
Failure	global; interactively complex and tightly coupled [17]	local; linear and loosely coupled, files as buffers
Adding an instrument	a feature request	an edit to a command line
Swapping a component	a re-integration	replacing one stage
Operator levels reachable [11]	fixed by the product's shape	all six, and the descent is recoverable
Maintainer	the original authors	anyone who has the shell

Table 1: Where composition lives determines the rest. Every row is a consequence of the one distinction, which is why the architectures are worth naming rather than treating as points on a continuum of tooling taste.

4.3 Maintenance

The castle's costs arrive on a schedule. Day one it demos; month three it has configuration; year one it has a changelog, deprecations, and a dependency graph, and the organisation discovers that the boundary work it bought has been converted into maintenance of the thing that hides the boundary work. Lehman's laws of software evolution guarantee the trajectory: a system in use either changes continually or becomes progressively less useful, and its complexity grows unless work is done to hold it down [13]. The pipeline's costs arrive differently — Section 6 is honest about them — but its unit of decay is one stage, and one stage can be read, replaced, or deleted by someone who was not present when it was written. The most widely adopted agent frameworks now run to hundreds of thousands of lines across their repositories; the pre-commit review gate in Section 8 is one line, and the comparison is unfair in exactly the way that matters.

5 Why Castles Get Built

If the pipeline's properties are as described, the castle needs an explanation, because capable people build castles constantly. We offer four mechanisms; they compound.

5.1 Abstraction is the trained response

A software developer's education is substantially training in one move: see a pattern, extract it; see repetition, generalise it. The move is correct for software that will be maintained for years, and it is the professional reflex that distinguishes the developer from the scripter — a demarcation actively policed, in Gieryn's sense [9]. Handed a new capability, the developer applies the reflex: wrap it, interface it, prepare it for futures that have not arrived. Fowler names the output [5]; Brooks named the amplified form when the builder has one successful system behind them and every deferred generalisation in front [2]. Nothing about the reflex is stupid. It is optimisation for a context — owned code, long horizons, stable requirements — that the boundary does not have. Boundary work is other people's components, short horizons, and requirements that change when the log format does. Gall's law states what the context rewards instead: the simple thing that works today, grown [7].

5.2 The second system is an agent

The agent is the castle's attractor because it promises to internalise the last thing left: composition itself. No pipeline to write, no stages to wire — describe the goal and the loop figures it out [29, 28]. The promise is not empty; the evaluation record says it is early [12, 27, 31], and the failure reports from deployment say the binding constraint was rarely the model [19, 15]. What the promise reliably produces today is the structure: a process holding state, control flow, and capabilities that no one outside it can inspect, which is to say the tightly coupled quadrant of Section 4.1 with a stochastic core. When it misbehaves, the question "which stage broke" has no referent.

5.3 The demo gradient

Castles demo well because a demo is a rendering, and rendering is the one thing a castle does do to its intermediates: the narrative of the agent thinking is a product surface, built to be watched. Plumbing demos badly because when it works there is nothing to watch — the commit was simply blocked, the log line simply explained. Organisations allocate by what they can see, which is the invisible-work finding again wearing a budget [24]: the architecture whose work is visible as spectacle out-competes the architecture whose work is visible as absence of incident, independent of which one is creating value.

5.4 The economics of invisibility

Put the mechanisms together and the market completes the circuit. Articulation work is real, unavoidable, and invisible; invisible work is unbudgeted; unbudgeted-but-necessary work gets acquired instead of assigned; and so the boundary work returns as a purchasable object — the framework, the platform, the agent product. A framework is articulation work resold as a product. Buying one does not remove the work; it relocates the work to where it can no longer be seen, adds the vendor's roadmap to it, and leaves the five decisions of Section 3 made by default, by someone else, for a system they have never met. The organisation's articulation work is now maintaining its articulation-work product, and the memo-writer's observation that most enterprise AI projects stall [15, 19] stops being mysterious: the projects were staffed for a capability problem and were, all along, an articulation problem.

6 When the Castle Is Right

A dichotomy that cannot lose is a slogan, so this section states where the argument fails, in both directions.

The wall. The castle's one good idea is the wall, and the wall is correct wherever the input is hostile. A model that holds capabilities and reads untrusted text is an injection target: the pipe that makes composition easy makes delivering an attacker's instructions easy by exactly the same mechanism [10]. Isolation, capability confinement, and a hard perimeter are castle architecture, and they are right. The refinement this paper argues for is that the wall belongs around *capability*, not around *composition*: an explicit allowlist is a wall; an opaque control loop is a keep. Keep the wall; skip the keep.

Many hands, one artefact. When thousands of concurrent users who do not have a shell must reach the same behaviour, the product shape is not optional. Externalised composition presumes an operator who can compose, and that presumption fails for most of the population most tools are built for — the honest limitation the companion paper concedes of its own case

study [11]. A castle is how composition is delivered to people who cannot be asked to do it.

State that must be transactional. Files in a working directory are the right store for one operator's context and the wrong store for ledgers, tenancy, and anything requiring atomic multi-party update. Where the state is the product, the product should hold the state.

Orchestration beyond the pipe's shape. The pipe carries a line of text left to right. Parallel fan-out, joins, retries with budgets, and latency-critical DAGs can be expressed in shell, but past a modest size the expression is worse than a program, and pretending otherwise produces the pipeline's own ruin: the three-thousand-line script that has become a castle without any of a castle's discipline. Gall's law has a second clause — the evolved complex system was once a simple system that *worked* — and evolution does not stop on its own [7]. The pipeline's discipline is keeping stages small enough to read, and it is a discipline, not a default.

The pipeline's native pathologies. Quoting, word-splitting, silent truncation, and error propagation that must be asked for (`set -o pipefail`) are real costs, paid in exactly the incidents this paper says the architecture avoids. The claim of Section 4 is not that the pipeline does not fail; it is that the pipeline fails locally, and that its failures are legible to the tools and people the organisation already has.

The compressed rule: internalise where the perimeter, the tenancy, or the shape of the control flow demands it; externalise everything else; and never internalise *by default*, because the default is where the five decisions of Section 3 get made without anyone making them.

7 Who Works the Boundary

If the value is created at the boundary, the staffing question is who is fitted to work there, and the answer is uncomfortable for current org charts: the fitted population already exists, already has job titles — operations, site reliability, platform, systems administration — and is mostly not where organisations put their AI effort.

The claim is about trained dispositions, not talent. Operational work selects, daily, for composition under time pressure with components one does not own: pipe this into that, gate it on a check, schedule it, and be able to explain at 3 a.m. which stage broke. Development work selects for abstraction over components one does own. Both dispositions are correct for their contexts; the boundary is the operational context wearing a new component. Handed a text-to-text tool, the operational disposition asks what can be piped into it — the question the boundary rewards — while the trained response of Section 5.1 asks what should be built around it.

Two literatures make the claim less anecdotal than it sounds. The end-user-programming tradition established both that the population of people who compose computations without holding the title "developer" is enormous — an order of magnitude larger than the professional population by the standard estimate [22] — and that their compositions succeed by staying close to the task rather than generalising away from it [16]. And the invisible-work literature predicts the organisational failure mode with some precision: because boundary work is invisible and low-status [24, 20], organisations staff AI with whoever is legible as "the AI person" — model knowledge, castle output — and the composition-native population is left to wire things up unofficially, uncredited, and without design authority over the five decisions they are in fact making. Section 9 turns the staffing claim into a prediction, because it should be one: it may be wrong, and it is testable.

8 Case Study: One Tool at the Boundary

Three of this paper's claims cost nothing to make. Externalise composition — into what? Keep the wall without the keep — built how? Let the operator descend when the result looks wrong — descend to what? The claims need an existence proof, and until there is one the argument is a preference. What follows is evidence about what is *constructible*, not about what works better; the competing interest is declared in Section 10, and the discipline is the same as the companion paper's: no performance, productivity, or preference claim is made on the tool's behalf [11].

`toast` is a command-line program that reads standard input, writes standard output, sets an exit code, and reads dot-files from the working directory. It implements none of composition, scheduling, persistence, or concurrency; each is left to the shell, which is the architectural decision this paper is about. The five decisions of Section 3 then land as follows.

Transport is the pipe. A log is triaged by piping it in; a review is a pipeline stage; a newsletter is `curl` into the model into `mail`. No stage knows it is part of anything.

Validation is the exit code. The one-line pre-commit gate —

```
git diff --cached | Reviewer || exit 1
```

— is a complete integration: model review on every commit, blocking on failure, installed by editing a hook. The honest description of the line is that it is articulation work made one line long, which is the whole design goal.

Capability is `.tools`, an allowlist with one permitted command per line; absent the file, the model touches nothing. This is the wall without the keep: a hard, operator-written, greppable perimeter around what the stochastic component may do, with no opaque loop inside it.

State is files. Context lives in `.crumbs`, conversation in a local history file; both can be read, versioned, diffed, and deleted with the tools the operator already has, and both survive a move between levels of delegation because they are beside the work rather than inside a product.

Failure routing is the non-zero exit, which stops a loop, fails a hook, or pages a human, at the stage that failed, with the stage's output on disk beside it.

What the case study shows is that the externalised architecture is reachable with one small artefact, that the castle's legitimate idea survives as a file, and that the operator's full ladder — one-shot to unattended — is reachable from the same binary by shell expression alone, which the companion paper demonstrates level by level [11]. What it does not show is that operators are better off: that is Section 9's job, and the tool's existence is not evidence for it.

9 Falsifiable Predictions and an Empirical Programme

The framework implies studies, and the studies have outcomes that would kill it.

P1: Effort lands at the boundary. Instrument real LLM deployments and classify engineering effort (commits, tickets, hours) as model-side (prompts, model choice, evaluation of outputs) or boundary-side (transport, validation, capability, state, failure). Prediction: boundary-side effort dominates, and its share grows with deployment age. The framework is damaged if sustained deployments spend most effort model-side.

P2: Failures are articulation failures. Code post-mortems of stalled enterprise LLM projects by proximate cause: capability (the model could not do the task) versus articulation (the task was never wired, gated, or maintained). Prediction: articulation causes dominate, consistent

with the interview evidence to date [19]. The framework is damaged if stalls are predominantly capability-bound.

P3: The architectures separate on maintenance, not demos. Matched teams receive the same integration task battery, one set composition-first tooling, one set framework-first. Measure time-to-first-demo, time-to-diagnose injected mid-pipeline faults, time-to-swap a component, and time for a fresh maintainer to make a safe change. Prediction: castles win or tie the demo; pipelines win diagnosis, swap, and handover by large margins. A null result across the maintenance measures would remove Section 4's teeth.

P4: Operational background out-predicts ML background. Stratify participants by background (operations versus development versus data science) on identical boundary tasks. Prediction: operational background predicts success better than ML knowledge does, and the interaction with tooling from P3 is positive. This is Section 7 made falsifiable, and it is the prediction the author considers most likely to fail in interesting ways.

P5: Growth curves diverge. Longitudinally, framework-based deployments accrete code, configuration, and dependencies faster than pipeline-based deployments serving comparable load, in line with the maintenance argument of Section 4.3. Convergent curves would suggest the castle's costs are transitional rather than structural.

10 Limitations and Threats to Validity

No empirical evaluation. Every causal claim in Sections 4, 5, and 7 is a hypothesis. The coupling analysis borrows a framework built for physical plants [17]; the invisibility economics borrows one built for human workplaces [24]; neither transfer is established here.

The dichotomy is a compression. Real systems mix the architectures, and the honest unit of analysis is probably the individual joint — where does *this* composition live — rather than the system. The paper's binary is a rhetorical convenience with the usual costs of one.

Anecdote and survivorship. The vignettes of Section 5 are patterns the author has watched, which is a polite name for anecdote, and pipelines that rotted have fewer chroniclers than castles that shipped.

Origin and interest. This argument began as a promotional memorandum, and the author develops and sells the tool in Section 8. The case study is confined to constructibility claims for that reason, and a reader who discards it entirely should find that Sections 3 through 7 do not lean on it.

Reflexivity. Naming an architecture "the castle" is boundary-work in Gieryn's sense [9]: a demarcation performed in pursuit of authority for a way of working. The paper accepts the charge. The defence is not neutrality but explicitness — the demarcation is stated, mechanised, and staked to the predictions of Section 9, which is the most that can be done with one.

11 Conclusion

A language model produces text, and text does not act. Everything between the text and the acting is work — transport, validation, capability, state, failure — and it is work with a name, a literature, and a fifty-year record of being done invisibly by whoever is nearest. The field's current answer to that work is to internalise it: wrap it in a framework, dissolve it into an agent, buy it as a product, and discover it again later as maintenance, opacity, and stalled pilots. The alternative

is older than the problem: leave composition outside, in the shell, in files, in exit codes, where every joint is a value, every failure is local, and every intermediate can be read by the person who is responsible for it.

Unix's lesson was never the pipe. It was that the joints were left where a person could reach them. The model is the least of an AI system; the joints are most of it; and the people who already live at the joints have been holding infrastructure together for half a century. What this paper asks for is narrow: notice where the value is created, keep the seams reachable, staff them on purpose — and when a castle is truly required, build the wall and not the keep.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999.

[2] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary edition, Addison-Wesley, Reading, MA, 1995. See ch. 5, "The second-system effect."

[3] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[4] B. Foote and J. Yoder. Big ball of mud. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, pages 653–692. Addison-Wesley, 2000.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999. See "speculative generality," ch. 3.

[6] R. P. Gabriel. Lisp: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991. The "worse is better" argument.

[7] J. Gall. *Systemantics: How Systems Work and Especially How They Fail*. Quadrangle/The New York Times Book Co., New York, 1977.

[8] E. M. Gerson and S. L. Star. Analyzing due process in the workplace. *ACM Transactions on Office Information Systems*, 4(3):257–270, 1986.

[9] T. F. Gieryn. Boundary-work and the demarcation of science from non-science: Strains and interests in professional ideologies of scientists. *American Sociological Review*, 48(6):781–795, 1983.

[10] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 79–90, 2023.

[11] 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. LinuxToaster.com, 2026. `https://linuxtoaster.com/papers/six_levels.pdf`

[12] S. Kapoor, B. Stroebl, Z. S. Siegel, N. Nadgir, and A. Narayanan. AI agents that matter. arXiv:2407.01502, 2024.

[13] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[14] 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.

[15] Project NANDA. *The GenAI Divide: State of AI in Business 2025*. MIT, July 2025. Industry report; methodology debated, cited here for the perceived scale of pilot failure rather than its precise magnitude.

[16] B. A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, 1993.

[17] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Basic Books, New York, 1984.

[18] R. Pike and B. W. Kernighan. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, 1984.

[19] J. Ryseff, B. F. De Bruhl, and S. J. Newberry. *The Root Causes of Failure for Artificial Intelligence Projects and How They Can Succeed: Avoiding the Anti-Patterns of AI*. RAND Corporation, RR-A2680-1, Santa Monica, CA, 2024.

[20] T. Reilly. Being glue. Talk, 2019. `https://www.noidea.dog/glue`

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

[22] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 207–214, 2005.

[23] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28*, pages 2503–2511, 2015.

[24] S. L. Star and A. Strauss. Layers of silence, arenas of voice: The ecology of visible and invisible work. *Computer Supported Cooperative Work*, 8(1–2):9–30, 1999.

[25] A. Strauss. Work and the division of labor. *The Sociological Quarterly*, 26(1):1–19, 1985.

[26] L. Suchman. Making work visible. *Communications of the ACM*, 38(9):56–64, 1995.

[27] F. F. Xu, Y. Song, B. Li, et al. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. arXiv:2412.14161, 2024.

[28] 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 37*, pages 50528–50652, 2024.

[29] 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.

[30] M. Zaharia, O. Khattab, L. Chen, et al. The shift from models to compound AI systems. Berkeley Artificial Intelligence Research Blog, February 2024.

[31] S. Zhou, F. F. Xu, H. Zhu, et al. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024. arXiv:2307.13854.