LLM-as-AgentCompiler: Towards an extremely easy paradigm for industry-level agent deployment

Anonymous authors
Paper under double-blind review

Abstract

ReAct agents are increasingly deployed to translate LLM capabilities into real applications, but their deployment is limited by high cost and instability, as they are built for multi-turn user interaction rather than reliable, low-cost automation. Recent work compiles ReAct agents into cheaper, more stable workflows, but existing methods are often complex or domain-specific. We propose *LLM-as-AgentCompiler*, a simple, general three-role agentic framework that uses coding agents to compile ReAct agents into cheaper, more stable workflows for high-throughput deployment. Evaluated on two benchmarks (six task domains) with no human supervision, it achieves roughly 2×–33× token reductions with no significant accuracy loss on five of six domains, plus one identified counterexample (retail under a weak residual model, which we analyze). We show that even a simple, pure-LLM framework with no expert priors can automatically (1) identify the deterministic decisions and computations in the memories, tools, and traces of ReAct agents across diverse domains, and (2) compile them into more static workflows with sparse, relaxed LLM nodes. We view LLM-as-AgentCompiler as an initial proof of concept toward this paradigm; establishing its generality on more open-ended domains remains future work.

Keywords: industry-scale agent deployment; agent compilation; deterministic core; inference cost reduction; model robustness.

1 Introduction

Large language models are increasingly deployed as *ReAct* agents—reasoning and acting step by step through tools—to bring their capabilities into real applications. This reactive design adapts to unfamiliar tasks without a procedure specified in advance, but it was built for multi-turn interaction with a user, not for reliable, low-cost automation. At industrial scale two properties become first-order. *Cost*: the model is called at every step and those calls repeat across large volumes of similar cases, so inference spend grows with traffic even when the work is routine. *Instability*: because every step depends on the model, behavior shifts when the provider updates, quantizes, or swaps the backend. Together these decide whether an agent running over fixed business material is affordable and dependable enough to deploy.

Yet not every step of such an agent needs a model. Once the relevant facts are gathered, a policy lookup, a fee computation, or the arguments of a confirmed action are already determined by the underlying rules. We call an agent's settled computation its *Deterministic Core*: the spans whose output is a deterministic function of the gathered identity, reads, and confirmed intent. The Core is general—it spans both computed *values* (a fee, an eligibility verdict) and determined *behavior* (which tool to call, with which arguments)—and what remains outside it is genuine judgment:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

interpreting free language, disambiguating, generating. It is also hard to imagine a specialized real-world task with *no* such logic: any workflow built on fixed policies, tables, or rules carries a Deterministic Core, so the question is how large it is in a given domain, not whether it exists—and compiling it is always worthwhile. Re-invoking the model on the Core is pure overhead; compiling it into code removes that overhead while leaving judgment to the LLM.

Recent work compiles agent behavior into cheaper workflows, but the methods are often complex or tied to a single domain. We propose *LLM-as-AgentCompiler*, a deliberately simple and general framework that uses LLMs to compile a ReAct agent's Deterministic Core into a mostly static workflow with sparse, relaxed LLM nodes. It is a pure-LLM pipeline of three roles—a *decomposer* that proposes the model/code boundary, a *builder* that realizes it as executable code plus LLM-port contracts and self-tests, and a *verifier* that challenges both with policy-grounded, gold-free checks—using no expert priors and no hand-written structure—no pre-specified workflow graph, schema, or neuro-symbolic scaffolding—and no human supervision (per-domain compilation traces in Appendix E). Reference answers stay outside this loop.

The boundary is drawn by a *functional* criterion: for a candidate span, is the output determined by the gathered facts under the rules? This is a criterion for what to *attempt* to compile, and it departs from prior practice, which compiles only what was *observed* to recur or what a smaller model can handle—so it can reach settled computation that was never seen to repeat, while leaving genuinely open decisions to an LLM port and marking an incomplete interface unresolved rather than guessing. Grounding determinism in the rules rather than in observed recurrence needs only a few traces to locate candidates, and it yields an interpretable, policy-cited unit rather than a statistical regularity; in principle it can even apply to a policy or business document directly, a broader reach we note as a direction rather than a demonstrated result. Whether a compiled span is in fact correct is then checked empirically, against held-out gold.

We evaluate LLM-as-AgentCompiler on two agentic benchmarks spanning six task domains. The compiled agent matches its prompt-only counterpart's held-out, gold-scored accuracy—and, where the model's own arithmetic was the weak link, exceeds it—while cutting per-call tokens by up to $\sim$30$\times$, and because the Core no longer calls a model it degrades far less when the backend is downgraded. Our central contribution is a demonstration: a *simple, LLM-based framework*, carrying no expert priors and no hand-written per-domain structure, can compile ReAct agents into workflows fit for industrial deployment across a range of tasks. Concretely: (1) *LLM-as-AgentCompiler*, a pure-LLM three-role framework that partitions an agent's computation, below any tool or action boundary, into compiled code and retained LLM ports—guided by a functional criterion and requiring no expert priors, no hand-written structure, and no human supervision; and (2) evidence across two benchmarks, six task domains, and two model tiers that the compiled agent is cheaper and markedly more model-robust at no significant loss of held-out task score. We view this as an initial proof of concept toward a general and simple paradigm for industry-scale agent deployment.

2 Related Work

Agents and agent optimization. Large language models are increasingly deployed as agents that reason and act over multiple steps, calling tools and adapting to unfamiliar tasks without a hand-written procedure [Yao et al., 2023, Wang et al., 2024b, Patil et al., 2024, Qin et al., 2024, Tang et al., 2023, Cai et al., 2024]. This flexibility has made reactive agents the default interface to complex tasks, but it also means the model is invoked at every step, including on work it has effectively done before. A substantial line of research therefore tries to make agents more efficient and

reliable by learning from their own experience: inducing reusable routines from past trajectories and recalling them on new tasks [Wang et al., 2024c, Zhou et al., 2026, Ouyang et al., 2026], abstracting demonstrations into transferable skills [Sarch et al., 2024, Wang et al., 2024a, 2025, Zheng et al., 2025, Zhao et al., 2024, Zheng et al., 2024, Mi et al., 2026], and maintaining procedural or long-term memory across tasks [Packer et al., 2024, Chhikara et al., 2025, Xu et al., 2025, Fang et al., 2026, Han et al., 2025, Zhou et al., 2025a]. These methods demonstrably improve success rates, sample efficiency, and reuse. Their common trait, however, is that the learned structure is retrieved back into the model's context to guide it rather than to replace it: the LLM still perceives or executes each step, and the induced unit is a multi-step skill rather than a sub-part of a single decision.

Agent compilation into code. Automatically compiling settled agent behavior into low- or zero-LLM code—so that it runs as code instead of repeated model calls—has become a widely pursued direction [Jaber and Jaber, 2026, El Yadouni and Li, 2026, Chong et al., 2026, Trooskens et al., 2026, Wang et al., 2026b, Malik, 2026, Chundru, 2026, Wang et al., 2026a, Iyamu, 2026, Dennis et al., 2026, Xu et al., 2026, Sakizli, 2026, Qiu et al., 2025, Li et al., 2026, Harikumar, 2026, Shao et al., 2026], and the underlying idea—that not every step needs a model—is the one we build on. This work is recent and largely concurrent: most entries are unpublished preprints reporting call-count or latency reductions under varied settings, and the area is still consolidating its evaluation protocols. Because a fair head-to-head would require re-implementing moving, unrefereed systems and reconciling incompatible cost and accuracy conventions, we do not attempt one here and instead position our approach by its design choice. Where these systems take their unit from a pre-existing structure and a correspondingly *observational* criterion (a behavior seen to recur) or a *capability-based* one (whether a smaller model handles a step), we draw the boundary *inside a single generation* by a *functional* criterion—what the rules determine—and check the result against gold; comparing the two criteria empirically is left to future work.

Using an LLM to optimize or compile the agent. A common thread across both directions above is that the system doing the improving is itself LLM-driven—a meta-level harness that reshapes a target agent. Early work in this vein uses a stronger model to search over prompts and demonstrations [Khattab et al., 2024] or to propagate natural-language critiques through a pipeline as textual "gradients" [Yuksekgonul et al., 2024], and more recent systems optimize agentic workflows with meta-tools or reflective search [Abuzakuk et al., 2026], making prompt and topology tuning systematic rather than manual. The compilation systems discussed earlier likewise rely on an LLM to perform the compilation itself, for instance by following a skill that mines and rewrites traces [El Yadouni and Li, 2026]. We share this reflexive stance that a capable model can build the artifact, but prior work is constrained by the *objective* it hands that model: rewording prompts or reordering nodes, or compiling only what is observed to recur or what a small model cannot do. Missing is an objective that directs the model to find the part of the computation that is a deterministic function of the inputs and turn exactly that into code.

This work. We introduce *LLM-as-AgentCompiler*, which uses a stronger model as the compiler with a specific objective: identify an agent's *Deterministic Core*—the spans (a computed value or a determined action) whose output is a deterministic *function* of the gathered identity, reads, and confirmed intent under fixed rules—and compile them into zero-LLM code, retaining LLM ports only for the judgment the rules do not fix. The boundary is drawn inside a single generation, below any tool or action boundary, by a functional criterion rather than by witnessed repetition or model capability, so the compiler can target settled computation that was never observed to recur; we

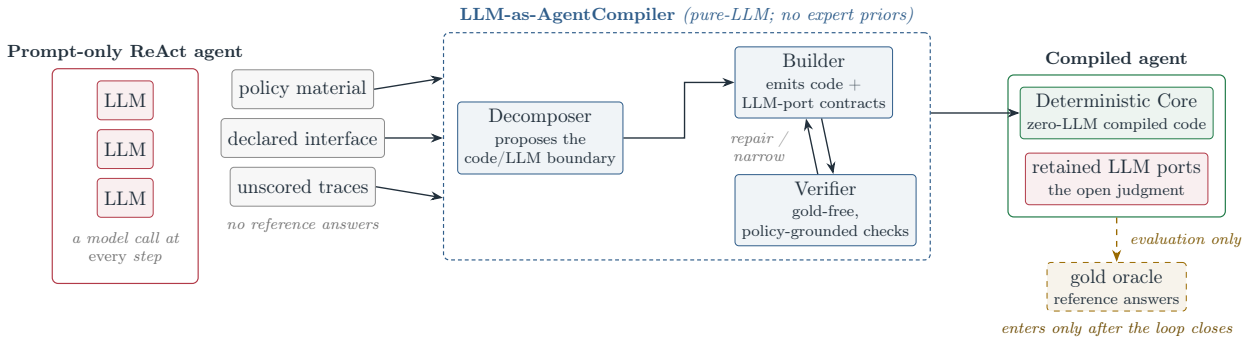

Figure 1: **LLM-as-AgentCompiler.** A prompt-only ReAct agent calls the model at every step (left). The compiler is a pure-LLM pipeline of three roles—a *decomposer* that proposes the code/LLM boundary, a *builder* that emits deterministic code plus LLM-port contracts, and a *verifier* with gold-free, policy-grounded checks—taking only policy material, the declared interface, and unscored traces, with no expert priors and no hand-written structure. It yields a *compiled agent* whose *Deterministic Core* runs as zero-LLM code and whose sparse ports keep the judgment; reference answers enter only at evaluation, outside the loop.

then check the result on held-out instances against gold, with an explicit cost account [Weinberger and Hozez, 2026, Erol et al., 2026, Kapoor et al., 2024] and under model downgrade measured by re-running rather than replaying [Gonuguntla, 2026]. Grounding the unit in the rules also makes it sample-efficient and interpretable, each compiled unit carrying its policy citation; in principle the same criterion could apply to a policy document directly, a reach we leave to future work. The framework is deliberately simple—three LLM roles, no expert priors, no per-domain logic—which we take as a feature: it targets the two properties that make deployment hard, inference cost and dependence on the model, by removing settled computation from the loop while leaving genuine judgment to the LLM.

3 Method

3.1 The Deterministic Core of an agent

Consider any agent that acts over fixed material—policies, tables, rules, and the tools of a domain—and gathers, as it runs, a state $z = (i, r, h)$: an identity and task context i, reads r from the environment or supplied tables, and intent h that the interaction has already confirmed. At each point the agent produces some output y: a value, a verdict, an argument list, or the choice of the next action. We partition these outputs by a single question—*is y a deterministic function of the state already gathered, under the fixed rules?*

$$y_s = f_s(z).$$

The spans for which such an f_s exists form the agent's *Deterministic Core*. The criterion is *functional*, not observational: agreement on a finite set of witnessed outputs is evidence for a candidate f_s, never its definition.

The core is broad, and it appears in agents that look nothing alike. It takes (at least) two forms. In its *value* form the settled output is a computed quantity or verdict—a fee assembled from rate tables, an eligibility decision read off account state, a classification fixed by a closed rule. In its *behavioral* form the settled output is a determined step or its arguments—once the intent is

confirmed and the state is known, *which* tool to call, with *what* arguments, and in *what* order are no longer open questions. What remains outside the core is genuine judgment: interpreting free language, disambiguating, relational inference, and generation—the outputs that are not functions of the gathered state and that a model must still produce.

3.2 Why compiling the core helps

If a span genuinely satisfies $y_s = f_s(z)$ and f_s is implemented faithfully, then the compiled span returns exactly what the model would have—for *that* span, accuracy is kept rather than traded for cost. This is a conditional, not a guarantee: whether a given span meets both requirements is empirical, so we do not read iso-accuracy off the criterion but measure it against held-out gold. Two deployment properties motivate isolating the core. First, cost: a compiled span is a table lookup or arithmetic, not a model call, so the per-step inference a reactive harness repeats across similar cases is removed where it was never doing judgment. Second, stability: f_s does not consult a model, so its output is invariant when the backend is updated, quantized, or downgraded—which is why a partially compiled agent degrades less than a prompt-only one. The core is therefore worth isolating precisely, and *no more than is true*: a span wrongly moved into the core trades a model call for a wrong answer.

3.3 Decomposition: a logical, functional partition

LLM-as-AgentCompiler draws the boundary by reasoning about the rules and the information flow, not by counting repetitions. For each candidate span it asks whether the required output is fixed by the gathered identity, reads, and confirmed intent under the supplied policy; the reasoning is functional and logical (what the rules determine) rather than statistical (what was seen to recur) or capability-based (what a smaller model happens to handle). Traces are used heuristically—to locate candidate computations and expose their dependencies—but repeated agreement on observed inputs never by itself certifies determinism over new ones. The partition is drawn wherever it falls, including *below* a tool or action boundary and inside a single generation: identifying an entity from natural language may stay with the model while mapping that entity through a policy table is compiled. Where the interface does not close the function—a needed table is absent, an intent is unconfirmed—the span is marked `UNCLASSIFIED` and left to the model rather than completed with a benchmark-specific guess; abstention is part of the method, not a failure of it.

3.4 The pipeline: components, their inputs, and their roles

LLM-as-AgentCompiler realizes the partition with three separated roles, each with a distinct information set, followed by an isolated evaluation. Keeping the roles apart is what lets the boundary be proposed, challenged, and revised without any one component both writing the artifact and judging it.

- **Decomposer** — *sees* the policy material, the declared interface (for the interactive tau tasks, the benchmark's fixed tool set), and unscored agent traces; *never* sees reference answers. It proposes, per span, `compile` / `keep-LLM` / `UNCLASSIFIED`, with a natural-language rationale and a falsifiable expectation for each compiled span. It writes no code and computes no answers.
- **Builder** — *sees* the decomposer's proposal (and, in later rounds, verifier suggestions). It emits the artifact: deterministic code for the compiled spans, an explicit contract for each retained LLM port, and self-tests. The artifact is a separately inspectable object.

- **Verifier** — *sees* the artifact, the policy, and the interface, and runs *gold-free*, policy-grounded checks on fidelity, on the live port interfaces, and on the proposed partition. It can suggest a split, a repair, or an abstention, but it does not edit the artifact it judges—and if it does, the framework detects the over-reach and reverts it; it may only *narrow* the compiled set, so an error moves work back to the model rather than inventing new compiled behavior.
- **Evaluation** — reference answers enter here and only here, after the synthesis loop has closed and the best checkpoint is selected. This keeps gold able to *falsify* the artifact without ever *supervising* its construction.

The order and separation are deliberate. Because the quality of the initial partition strongly conditions everything built on it [Zeng et al., 2025, Abuzakuk et al., 2026, Khattab et al., 2024], the decomposer runs *first and only once*, reasoning about the logic of the task alone—policy, interface, traces—before any code exists to bias it; the builder commits that partition to an inspectable artifact, and the verifier, seeing the artifact rather than the intent, catches a mis-drawn boundary or an unfaithful compilation and pushes corrections back. This division earns its keep: across the per-domain traces (Appendix E) the verifier repeatedly overturns a plausible-but-wrong decision—for instance, rejecting an early "the first checked bag is free" convention in airline by enumerating the fee cases against it, or narrowing an over-broad compiled span back to the model—a correction a single agent that both proposes and judges has no independent vantage to make. This is why we keep the roles apart rather than fold them into one iterated agent. The loop runs decomposition once, then iterates build against verification:

```
Input : policy material P, unscored traces T, interface I
Output: compiled artifact A with explicit LLM ports
B = Decompose(P, T, I)   # per span: compile / keep-LLM / UNCLASSIFIED  (run once)
A = Build(B)             # deterministic code + LLM-port contracts + self-tests
repeat:
    S = Verify(A, P, I)  # policy-grounded, gold-free: defects + boundary edits
    A = Build(B, S)      # apply only boundary-narrowing or repair suggestions
until no admissible change or round budget reached
return best checkpoint A # scored by the gold oracle OUTSIDE this loop
```

Listing 1: The compilation loop. Synthesis (decompose, build, verify) is gold-free; reference answers are read only by the external oracle, after the loop.

At runtime the retained ports and the compiled code compose directly: an LLM port turns raw input into the facts z, deterministic code computes $f_s(z)$ for every compiled span, and any remaining port supplies the judgment the rules do not fix—a single agent whose settled computation runs as code and whose open decisions stay with the model.

4 Experimental Setup

4.1 Evaluation arms and conditions

We organize evidence by domain, interface condition, and evaluation stage rather than pooling all runs. We use two public agent benchmarks. *RuleArena* [Zhou et al., 2025b] poses single-generation tasks—airline baggage fees, tax, and NBA roster rules—that require reading a rule manual and computing a fee or eligibility decision; its airline and tax domains are our primary calculator candidates, and tax additionally supplies a model-downgrade ladder. The *tau* benchmark family [Yao et al., 2024, Barres et al., 2025] poses interactive, multi-turn tool-use tasks; we use its

telecom, airline, and retail domains and report them separately because the protocol runs repeated trials per task.

For each artifact we distinguish two runtime arms. The *pure* arm gives the task to the prompt-only agent; the *compiled* arm runs the compiled parser/ports and deterministic code end to end.

4.2 Splits, pairing, and metrics

The RuleArena held-out split is deterministic and gold-free: with the all-levels setting the first ten records of each difficulty are development and the rest are held out (ten removed per level, not the first 30 concatenated), so airline and tax yield 270 held-out rows from 300 and NBA yields 186 from 216. Comparisons use aligned row identifiers and keep separate all-row and fair-set readings when truncation forces an exclusion. Accuracy is the fraction of stored score flags marked correct, always reported with its n and split. For tau, seeds of one task are not independent tasks, so per-trial success and pass^3 are kept distinct.

The principal cost quantity is inference work at a fixed model and task set, measured with local tokenizer counts.

4.3 Reproducibility

Both benchmarks are public and their held-out splits are deterministic, and we release the compiled artifacts, the LLM-port prompts, and the scoring scripts, so every reported number can be reproduced by re-running the released pipeline against the benchmark. Accuracy is computed by the benchmark's gold oracle, which the artifact synthesis never sees; token counts use a local tokenizer.

5 Results

We report six task domains across two benchmarks: RuleArena (airline, tax, NBA; 270/270/186 held-out rows) and the interactive tau tasks (airline, telecom, retail; 20/40/29 independent units, each over K seeds). Accuracy is held-out and gold-scored; token counts use a local tokenizer. Because the claim is *iso-accuracy at reduced cost*, the accuracy side tests *non-inferiority*—that compilation does not lose task score—rather than winning, while the cost side tests reduction.

5.1 Iso-accuracy at reduced cost

The compiled agent matches the prompt-only agent's held-out accuracy across both benchmarks while removing most of the per-call work (Table 1).

Single-turn RuleArena. Airline is the clearest case: at 270 held-out rows the compiled arm scores 0.7556 against 0.7370 pure (no compiled row loses; McNemar $p = 0.0625$) at a token ratio of 0.254—a 3.9× reduction, mostly policy text that no longer rides in the prompt—with the parser lossless (all 300 rows). On **tax** the arms are a *strict dead-heat*: 0.8778 each, $\Delta = 0$, McNemar $p = 1.0$, paired-bootstrap 95% CI $[-0.026, +0.026]$ inside the 5% band—the cleanest "equivalent" evidence in the paper, at roughly half the tokens. The exact calculator can even do *better* when the model's arithmetic is the weak link: an earlier end-to-end round reached 0.90 vs. 0.8074 (+0.093, 25:0, $p \approx$ 6e-8), the gap being the model misreading brackets in a long document. On **NBA** the compiled agent is non-inferior and numerically better, 0.7258 vs. 0.6075 (+0.118, 43:21, $p = 0.008$) at 0.124× marginal cost.

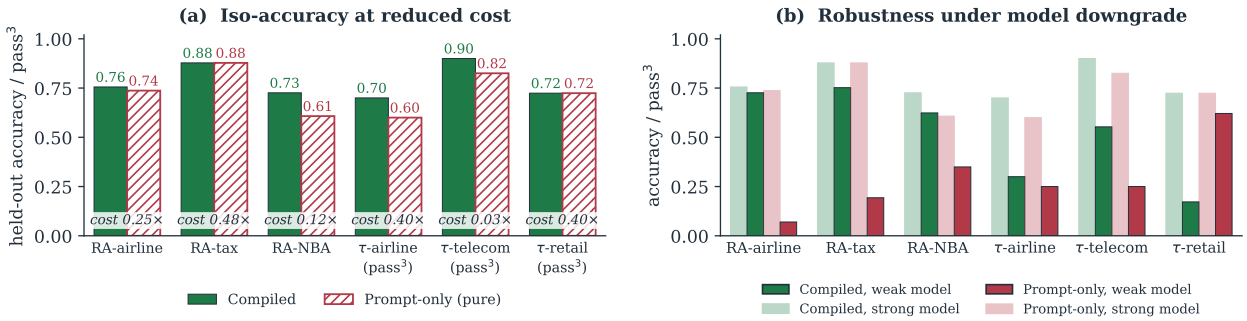

Figure 2: **Results summary.** **(a)** On all six domains the compiled agent (green) matches or exceeds the prompt-only agent's held-out score (red, hatched) at the token/cost ratio printed under each group. **(b)** Under weaker residual models the compiled arm (dark) stays above the equivalent prompt-only arm in five of six domains; faded bars are the strong-model (Opus) references. Retail is the honest exception: its retained gatherer needs a strong model, so under Haiku the compiled arm falls below pure. Panel-(a) values and cost ratios are stated in the text; τ pass^3 detail is in Table 2.

Table 1: Iso-accuracy at reduced cost, Opus 4.8. RuleArena reports held-out accuracy; tau reports pass^1 (pass^3 in Table 2). McNemar p is the paired test on the discordant cases; the direct token ratio is compiled/pure.

Domain	n	Compiled	Pure	Δ	McNemar p	Token ratio
RA-airline	270	0.7556	0.7370	+0.0185	0.0625	0.254
RA-tax	270	0.8778	0.8778	0.0000	1.000	0.479
RA-NBA	186	0.7258	0.6075	+0.1183	0.008	0.124
τ-airline	20	0.7667	0.7667	0.0000	1.000	0.403
τ-telecom	40	0.9583	0.9333	+0.0250	0.250	0.030
τ-retail	29	0.862	0.839	+0.023	1.000	0.395

Interactive tau tasks (across a tool boundary). The tau family is multi-turn and *has* tool calls, so it tests the harder claim: that the boundary can still be drawn by functional role rather than along the tool-call sequence. The compiled agent keeps an LLM gatherer for the customer's free-language intent and a deterministic engine that owns permission, argument assembly, and the action sequence. On pass^1 all three domains match or edge the baseline (Table 1); the pass^3 gaps (airline 0.70 vs. 0.60, telecom 0.90 vs. 0.825) sit inside the ± 0.10 noise floor at these unit counts and read as direction-consistent, not significant. Token reductions are large (0.030× telecom, 0.40× airline, 0.395× retail); these are recurring deployment costs, whereas compiling a domain is a separate *one-time* cost—a few builder–verifier strong-model rounds, on the order of tens to a few hundred RMB, amortized over all later runs and negligible at the throughput this targets. The compiled telecom engine is in Appendix D, the full pass^3 and per-model breakdown in Appendix B.

Statistical analysis. We judge non-inferiority against the standing 5% safety band: a domain passes when the paired-bootstrap 95% CI of Δ = compiled − pure has its lower bound above -0.05 (B=10^4). Every domain passes, and the two cleanest RuleArena cases pin the extremes—airline clears zero at $[+0.004, +0.037]$ (a strict improvement), tax sits centered at $[-0.026, +0.026]$. The McNemar column of Table 1 corroborates (dead-heat on tax and τ-airline, a significant paired advantage on NBA, no compiled losses on airline): compilation is "equivalent and safe" not because

it wins but because its CI cannot reach a meaningful loss. The claim is well powered on RuleArena ($n = 186$–270) and weaker on tau ($n = 20$–40 units), where we report pass^3 differences as direction-consistent rather than significant.

5.2 Cost reduction from compilation, not a weaker model

A natural question is whether the savings come from a cheaper residual model rather than from removing computation. We separate the two with a difference-in-differences (DiD) test: re-run *both* arms under a weaker residual model [Gonuguntla, 2026] and ask whether the compiled arm loses *less* than the pure arm relative to Opus. Since the compiled code calls no model and its score is byte-identical across the swap, any DiD is carried entirely by the retained LLM port. It is strongly positive: on tax DiD $= +0.167$ under GLM and $+0.559$ under Haiku 4.5 ($p = 10^{-4}$, pure collapsing to 0.193 while compiled holds 0.752); airline (0.726 vs. 0.070) and NBA ($\Delta_{\text{deg}} = +0.156$, $p < 10^{-6}$) agree, as does telecom (pass^3 0.55 vs. 0.25); the full breakdown is in Appendix B. Robustness is domain-dependent, and **retail** marks the limit: under Haiku its compiled arm collapses to pass^3 0.172 while pure holds 0.621 ($p < 10^{-3}$), the only domain degrading *more*—by design, since retail retains a heavy LLM gatherer that no deterministic guard can recover. A compiled Core is thus carried out of the model's reach, but a retained port lives or dies with the residual model.

6 Limitations and Scope

Our claims are bounded chiefly by the breadth of evidence rather than by the mechanism. The functional criterion is an operational rule, not a proof that a generated program is correct for every input, and broadening to more domains and agent styles will take further validation. We also compare only against a prompt-only baseline: a head-to-head against prior compilation methods is outstanding, as most are concurrent, unrefereed, and without released code—we leave it to future work once those systems stabilize and open-source.

The size of the payoff depends on how large the Deterministic Core is, which varies by domain. Our two benchmarks deliberately exercise *different* capabilities—RuleArena demands reading a long rule document and exact multi-step computation, the tau tasks multi-turn intent-gathering and permissioned tool control across a dialogue—so success on both is evidence of breadth, not of a single favorable setting. What they share is not a task type but a substantial Deterministic Core, which we take to be the common case for specialized work; a genuinely open-ended domain of mostly free generation would have a small Core and small savings, but such domains are the exception among the workflows we target.

7 Conclusion

Reactive agents and workflows need not be opposing designs: an intermediate step identifies the portions of an execution already determined by gathered state, compiles those into code, and keeps model calls where judgment remains open. LLM-as-AgentCompiler realizes this as an auditable decomposer–builder–verifier pipeline with gold-isolated evaluation; across six domains in two benchmarks its compiled substructure matches a prompt-only baseline on held-out settings and, calling no model, degrades far less under downgrade. We delegate compilation to coding agents, so we expect its reach to grow with model capability. This also suggests a broader paradigm: fewer hand-designed solution priors, more agent-led revision through verification feedback, making compilation an iterative search rather than a fixed expert prescription.

References

Sami Abuzakuk, Anne-Marie Kermarrec, Rishi Sharma, Rasmus Moorits Veski, and Martijn de Vos. Optimizing agentic workflows using meta-tools, 2026. URL https://arxiv.org/abs/2601.22037.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2024. URL https://arxiv.org/abs/2305.17126.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

Zan Kai Chong, Hiroyuki Ohsaki, and Bryan Ng. Compiling deterministic structure into slm harnesses. *arXiv preprint arXiv:2604.17450*, 2026.

Jagadeesh Chundru. Agentic compilation: Mitigating the llm rerun crisis for minimized-inference-cost web automation. *arXiv preprint arXiv:2604.09718*, 2026.

Simon Dennis, Rivaan Patil, Kevin Shabahang, and Hao Guo. Compiling agentic workflows into llm weights: Near-frontier quality at two orders of magnitude less cost, 2026. URL https://arxiv.org/abs/2605.22502.

Salma El Yadouni and Guanyi Li. Tracecompiler: Skill-guided mining and compilation of llm agent traces into mostly deterministic workflows. *arXiv preprint arXiv:2608.02680*, 2026.

Mehmet Hamza Erol, Batu El, Mirac Suzgun, Mert Yuksekgonul, and James Zou. Cost-of-pass: An economic framework for evaluating language models. In *International Conference on Learning Representations (ICLR)*, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory, 2026. URL https://arxiv.org/abs/2508.06433.

Ashritha Gonuguntla. The replay gap: Static evaluation of model switching in llm agents scores the wrong world, 2026. URL https://arxiv.org/abs/2608.08239.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent llm systems for workflow automation, 2025. URL https://arxiv.org/abs/2510.04851.

Pranav Harikumar. Plancompiler: A deterministic compilation architecture for structured multi-step llm pipelines. *arXiv preprint arXiv:2604.13092*, 2026.

Nossa Iyamu. Activity frames: Deterministic screen-activity compilation for agent memory and replay. *arXiv preprint arXiv:2608.05784*, 2026.

Jaber Jaber and Osama Jaber. Auto: The agi compiler. *arXiv preprint arXiv:2607.04542*, 2026.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.03714.

Junyan Li, Zhang-Wei Hong, Maohao Shen, Yang Zhang, and Chuang Gan. Flowcompile: An optimizing compiler for structured llm workflows. *arXiv preprint arXiv:2605.13647*, 2026.

Arun Malik. Progressive crystallization: Turning agent exploration into deterministic, lower-cost workflows in production. *arXiv preprint arXiv:2607.07052*, 2026.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents, 2026. URL https://arxiv.org/abs/2602.01869.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory, 2026. URL https://arxiv.org/abs/2509.25140.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations (ICLR)*, 2024.

Libin Qiu, Yuhang Ye, Zhirong Gao, Xide Zou, Junfu Chen, Ziming Gui, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Kun Zhao. Blueprint first, model second: A framework for deterministic llm workflow. *arXiv preprint arXiv:2508.02721*, 2025.

Furkan Sakizli. Tscg: Deterministic tool-schema compilation for agentic llm deployments. *arXiv preprint arXiv:2605.04107*, 2026.

Gabriel Sarch, Lawrence Jang, Michael J. Tarr, William W. Cohen, Kenneth Marino, and Katerina Fragkiadaki. VLM agents generate their own memories: Distilling experience into embodied programs of thought. In *Advances in Neural Information Processing Systems 37: NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/8ac50fd0a4eeeb1f077b17bb7c5353c3-Abstract-Conference.html.

Jie-Jing Shao, Haiyan Yin, Yueming Lyu, Xingrui Yu, Lan-Zhe Guo, Ivor Tsang, James Kwok, and Yu-Feng Li. Lifting traces to logic: Programmatic skill induction with neuro-symbolic learning for long-horizon agentic tasks, 2026. URL https://arxiv.org/abs/2605.01293.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Geert Trooskens, Aaron Karlsberg, Anmol Sharma, Lamara De Brouwer, Max Van Puyvelde, Matthew Young, John Thickstun, Gil Alterovitz, and Walter A. De Brouwer. Compiled ai: Deterministic code generation for llm-based workflow automation. *arXiv preprint arXiv:2604.05150*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024a. URL https://openreview.net/forum?id=ehfRiF0R3a.

Jincheng Wang, Min Zheng, and Tao Wei. Covenant: Natural-language workflow compilation for aligned agent execution. *arXiv preprint arXiv:2607.25400*, 2026a.

Xiaohua Wang, Kai Yu, XuXiao Liang, Liang Wang, and Chao Han. Good to go: The loop skill engine that hits 99% success and slashes token usage by 99% via one-shot recording and deterministic replay. *arXiv preprint arXiv:2605.14237*, 2026b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *International Conference on Machine Learning (ICML)*, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c. URL https://arxiv.org/abs/2409.07429.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks, 2025. URL https://arxiv.org/abs/2504.06821.

Sarel Weinberger and Amir Hozez. Token reduction is not cost reduction. *arXiv preprint arXiv:2607.12161*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents, 2025. URL https://arxiv.org/abs/2502.12110.

Xiangzhe Xu, Hanxi Guo, Guangyu Shen, Siyuan Cheng, and Xiangyu Zhang. Natural-language workflows are not software yet: Artifact-driven compilation for reliable agent execution, 2026. URL https://arxiv.org/abs/2608.21341.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024. URL https://arxiv.org/abs/2406.12045.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Guancheng Zeng, Xueyi Chen, Jiawang Hu, Shaohua Qi, Yaxuan Mao, Zhantao Wang, Yifan Nie, Shuang Li, Qiuyang Feng, Pengxu Qiu, Yujia Wang, Wenqiang Han, Linyan Huang, Gang Li, Jingjing Mo, and Haowen Hu. Routine: A structural planning framework for llm agent system in enterprise, 2025. URL https://arxiv.org/abs/2507.14447.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: LLM agents are experiential learners. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, February 20-27, 2024, Vancouver, Canada*, pages 19632–19642. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29936.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=Pc8AU1aF5e.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning llm agents without fine-tuning llms, 2025a. URL https://arxiv.org/abs/2508.16153.

Ruiwen Zhou, Wenyue Hua, Liangming Pan, Sitao Cheng, Xiaobao Wu, En Yu, and William Yang Wang. Rulearena: A benchmark for rule-guided reasoning with llms in real-world scenarios. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 550–572, 2025b. doi: 10.18653/v1/2025.acl-long.27. URL https://aclanthology.org/2025.acl-long.27/.

Yuqing Zhou, Zhuoer Wang, Jie Yuan, Hong Wang, Samson Koelle, Ziwei Zhu, and Wei Niu. Wiseflow: Workflow-induced structured experience for self-evolving conversational service agents, 2026. URL https://arxiv.org/abs/2601.08158.

A Reproducibility and Disclosure

For review we provide an *anonymous code repository*[1] that contains, for all six domains, the *compiled artifacts*—the deterministic code and the retained LLM-port prompts—together with the deterministic held-out splits, so that the reported partition and its outputs can be inspected directly. The full *framework*, the decomposer–builder–verifier pipeline code, will be open-sourced upon acceptance, along with the scoring scripts, so that every number can then be reproduced by re-running the released pipeline against the public benchmark and scoring with its gold oracle. For every run we record the model requested and the model served, the tokenizer used for local token counts, and the treatment of truncated or empty replies.

A.1 Gold isolation

Reference answers enter only the external oracle, after the synthesis loop has closed. The decomposer, builder, and verifier never receive gold, and the synthesis workspace contains no reference

[1] https://anonymous.4open.science/r/LLM-as-AgentCompiler-109B

answers; we include an input-access table for each role and a check that the workspace holds no gold totals.

A.2 AI use disclosure

This work uses large language models in two distinct ways, which we separate explicitly. First, as the *object of study*: the compiled artifacts and the decomposer–builder–verifier pipeline that produces them are generated by LLMs with no hand-written per-domain structure, and this generation is itself the contribution described in Section 3 and traced in Appendix E rather than an authoring convenience. Second, as an *authoring aid*: the authors used AI-based coding and writing assistance while implementing the framework code and while drafting and editing this manuscript. In both roles the authors take full responsibility for the content; every reported claim, number, and citation was checked by the authors against the released runs and sources.

A.3 Artifact availability

The anonymous repository above is scrubbed of credentials, private endpoints, and identifying information; the released compiled artifacts and held-out splits are those used for the reported numbers. The framework code and scoring scripts will follow the same scrubbing before their public release upon acceptance.

B Per-Domain Metric Detail

This appendix expands the summary numbers of Section 5. Table 2 gives the full $\text{pass}^1/\text{pass}^3/\text{pass@3}$ breakdown for the interactive tau tasks; Table 3 gives the per-model RuleArena accuracies behind the downgrade analysis; Table 4 the tau downgrade; and Table 5 the full token-cost ratios. RuleArena figures are over 270/270/186 held-out rows; tau over 20/40/29 independent units.

Table 2: Interactive tau tasks, held-out, Opus 4.8; pass^3 over K seeds per unit, compiled vs. pure arm. pass^3 gaps sit inside the ± 0.10 noise floor at these unit counts, so we treat them as direction-consistent rather than significant.

		pass^1		pass^3		pass@3	
Domain	n	compiled	pure	compiled	pure	compiled	pure
Airline	20	0.7667	0.7667	0.700	0.600	0.85	0.85
Telecom	40	0.9583	0.9333	0.900	0.825	1.00	1.00
Retail	29	0.862	0.839	0.724	0.724	0.966	0.931

‡ The NBA pure arm under GLM is not reported: the thinking-mode model with the full rule prompt is unstable through the gateway (serial, ∼15 min/row, ∼47% discarded), so no reliable paired baseline exists.

C The Compiled Airline Artifact

This appendix reproduces the artifact LLM-as-AgentCompiler emitted for the RuleArena airline baggage-fee task, so the reader can inspect what the system produces rather than a description of it. The compiled agent is two objects that meet at one hand-off: a single retained **LLM port** that extracts facts and never computes, and a **zero-LLM calculator** that owns every fee, threshold, table, and sum.

Table 3: RuleArena accuracy per residual model (both arms re-run per model), compiled vs. pure arm. GLM rows are provisional (†: served-model equivalence unverified—the gateway substitutes the served checkpoint). For tax we list both the headline round (a strict dead-heat) and an earlier end-to-end round; DiD is the difference-in-differences of Section 5.

Domain	Model	compiled	pure	note
Airline	Opus 4.8	0.7556	0.7370	headline
	GLM†	0.7519	0.7074	
	Haiku 4.5	0.7259	0.0704	compiled $\gg$ pure
Tax	Opus 4.8 (final)	0.8778	0.8778	headline, dead-heat
	Opus 4.8 (earlier)	0.90	0.8074	earlier end-to-end round
	GLM†	0.7148	0.5481	DiD = +0.167
	Haiku 4.5	0.7519	0.1926	DiD = +0.559
NBA	Opus 4.8	0.7258	0.6075	headline
	GLM	0.7043	—	pure unmeasurable‡
	Haiku 4.5	0.6237	0.3495	DiD = +0.156

Table 4: Interactive tau tasks under the Haiku 4.5 downgrade (pass^3; retail pass^3), both arms re-run. Telecom and airline stay compiled $\geq$ pure; retail is the counterexample where the retained gatherer needs a strong model.

Domain	compiled	pure	vs. Opus
Airline	0.300	0.250	both fall; compiled ahead
Telecom	0.550	0.250	both fall; compiled ahead
Retail	0.172	0.621	*compiled below pure*

Table 5: Token-cost ratios (compiled/pure), Opus 4.8, local tokenizer. "direct" counts all tokens; "out×5" weights output tokens 5× to reflect typical output-token pricing. Every domain is well below parity (3×–33× reduction).

Domain	direct ratio	out×5 ratio
RA-airline	0.254	0.244
RA-tax	0.479	0.508
RA-NBA	0.124	0.236
τ-airline	0.403	0.531
τ-telecom	0.030	0.149
τ-retail	0.395	0.413

```
facts = parse(description)          # PORT A: the only LLM in the system
total = compute_total_cost(         # calculator.py, zero LLM
    facts["base_price"], facts["customer_class"],
    facts["routine"], facts["direction"], facts["bag_list"])
```

Listing 2: The entire runtime coupling: the LLM is called once to fill five fields; the total is then a deterministic function of them.

The calculator validates `customer_class`, `routine`, `direction`, and `item_kind` against fixed enumerations and raises on anything outside them, so a malformed extraction fails loudly instead of returning a plausible wrong number.

C.1 The retained LLM port: parser prompt

The parser receives the case description and the two enum vocabularies; it does *not* receive the policy or fee tables, which is what removes ∼550 lines of policy from every model call. The system prompt, abridged:

```
You extract structured facts from a description of one air traveller's bags.
You do NOT calculate fees, totals, or prices. Another program does all
arithmetic from the facts you return. Return exactly one JSON object:

  {"base_price": <number>,
   "customer_class": "<one of CLASS>",
   "routine": "<one of REGION>",          # the non-U.S. endpoint's region
   "direction": 0 or 1,                    # 0 = from U.S., 1 = to U.S.
   "bag_list": [{"name","size":[L,W,H],"weight",
                 "item_kind"?, "cabin_gauge"?}, ...]}

CLASS  : Basic Economy, Main Cabin, Main Plus, Premium Economy, Business, First
REGION : U.S., Canada, Mexico, Cuba, ... , China, Japan, Australia (21 values)
KIND   : diaper_bag, breast_pump, breast_milk_cooler, child_safety_seat,
         stroller, medical_or_mobility_device   (or omit the key)

- base_price: the ticket price as written. Nothing added or removed.
- routine  : resolve both cities to countries, return the non-U.S. endpoint's
             REGION; both endpoints in the U.S. -> "U.S." and direction 0.
- bag_list : one entry per item, IN THE ORDER LISTED. Do not reorder or merge.
             size is [L,W,H] as written -- do not sort, add, or convert.
- item_kind: set only from the item's NOUN, not its size or weight. You report
             WHAT THE ITEM IS, not what it costs; the program decides the
             consequence. If the noun is not one of the six, omit the key.
- cabin_gauge = "garment_bag" only for a soft-sided garment bag; do not measure
             it or decide whether it fits.

Never state a fee, tier, threshold, or total. Never add or compare the numbers
you extract -- not even L + W + H. Never decide which item rides in the cabin or
is checked bag #1. Never call an item exempt/free/waived. Output JSON alone.
```

Listing 3: PORT A system prompt (abridged from `parser_spec.md`).

The two judgements the port *does* make cannot be compiled: an item's *noun* decides which of the six named categories it belongs to (if any) and whether the garment-bag gauge applies — "neither is a function of $[L, W, H]$ and a weight," whereas everything downstream of the noun is arithmetic or policy reading and lives in the code.

C.2 The compiled calculator

`calculator.py` is 1220 lines of pure Python — no I/O, no imports, no randomness — so the same inputs give the same integer, always. Every constant is transcribed from the policy with its source line cited in the code, and every point where the policy is silent is a named `SWITCH_*` constant (the `U1`–`U11` undecided points), so a reviewer changes each in one place. Table 6 lists the functions; the deterministic core is reproduced below (docstrings abridged).

```python
def base_fee(ordinal, customer_class, routine, direction):
    # ordinals ≥ 4 all read the "Fourth Bag +" table (policy.txt:190)
    if ordinal < 1:
        raise ValueError("checked-bag ordinal must be ≥ 1")
    row = _BASE_FEE_TABLES[(min(ordinal, 4), direction)][routine]
```

Function	What it owns
`compute_total_cost`	Public contract: `base_price` + every checked-bag fee, as an `int`.
`compute_breakdown`	Main body: classify cabin vs. checked items, assign ordinals, price each bag, sum, and emit an `anomalies` list that never changes the total.
`base_fee`	Base fee by table lookup on (min(ord, 4), dir, routine, class).
`oversize_fee`	Dimension-sum tier ladder (left-open, right-closed); class-independent.
`overweight_fee`	Absolute-weight tier ladder; `free_limit` only sets whether the bag is overweight at all.
`weight_free_limit`	The 50 vs. 70 lb allowance from class/routine/complimentary status.
`select_free_cabin_items`	Which items ride free in the cabin; the free *count* is a total function of the dimensions ([B-1]).
`carry_on_fits`, `personal_item_fits`	Per-axis, orientation-free gauge tests; the personal gauge nests inside the carry-on gauge.
`_consideration_order`, `_order_checked`, `_fact_key`	Assign slots and ordinals per `SWITCH_ORDINAL_ASSIGNMENT`, reading only each item's own facts, so permuting `bag_list` cannot move the total.
`_bag_fields`	Validate one parser bag; *reject* the withdrawn `"exempt"` key.
`_is_complimentary`, `_is_not_accepted`	The `U1` complimentary rule and the "Not accepted" tier.
`_validate_enum`, `BagNotAcceptedError`	Reject out-of-enum values; raise (no total) for a refused itinerary.

Table 6: The compiled calculator, by function. The LLM port supplies five facts; every row is a deterministic function of them.

```
    return row[_CLASS_COL[customer_class]]

def oversize_fee(dim_sum, routine):
    # tiers "Over X to Y" => left-open, right-closed; class-independent
    t1, t2 = OVERSIZE_FEES[routine]
    if dim_sum ≤ DIM_FREE_LIMIT:  return 0
    if dim_sum ≤ DIM_TIER1_LIMIT: return t1
    if dim_sum ≤ DIM_TIER2_LIMIT: return t2
    return t2 if SWITCH_BEYOND_TOP_TIER == "clamp" else 0   # U3: >115 in

def overweight_fee(weight, routine, free_limit):
    # boundaries on ABSOLUTE weight; free_limit only sets overweight-or-not
    if weight ≤ free_limit: return 0
    # ... tier ladder over WEIGHT_TIER_A/B/C[routine] ...
```

Listing 4: The three per-bag fee functions — each a pure lookup over a transcribed policy table. The per-bag surcharge is `max(oversize_fee, overweight_fee)`.

The flagship function is the cabin allowance. Its docstring carries the enumeration the verifier produced ([B-1]): the free *count* is a total function of the dimensions, and only the *label* of which item claims a slot is undecidable — a distinction the code makes structurally, never by searching.

```
def select_free_cabin_items(items):
    # One carry-on slot AND one personal-item slot (policy.txt:5). A personal-
    # gauge item fits either slot (gauges nest, P subset C); a carry-on-gauge
    # item fits only the carry-on slot. So the free count is
    #     min(|C|, 1 + (1 if |P| ≥ 1 else 0)),
    # and giving the carry-on slot to a C-but-not-P item always attains it.
    fits_c, fits_p = [], set()
```

```
    for i, item in enumerate(items):
        size, gauge = item[0], item[1]
        if carry_on_fits(size, gauge):
            fits_c.append(i)
            if personal_item_fits(size):
                fits_p.add(i)
    if not fits_c:
        return {}
    order = _consideration_order(fits_c, items)
    carry_on_pos = next((i for i in order if i not in fits_p), order[0])
    free = {carry_on_pos: "carry_on"}
    for i in order:
        if i != carry_on_pos and i in fits_p:
            free[i] = "personal_item"; break
    return free
```

Listing 5: `select_free_cabin_items`: the gauges nest, so the free count is `min(|C|, 1 + (1 if |P|>=1 else 0))` and the assignment never searches.

The nesting is a tested fact, not an assertion: `personal_item_fits(s)` implies `carry_on_fits(s)` for every `s` because $18{\times}14{\times}8 \le 22{\times}14{\times}9$ on every axis, and `experiment.py` enumerates the integer triples up to 25 in to keep it so. At runtime `compute_breakdown` re-checks the attained count against this maximum and reports any shortfall as an `anomaly` that never silently changes the total.

D The Compiled Telecom Artifact

The telecom task differs from the airline one in a way that sharpens the claim: it *has* tool calls (an action boundary exists), yet the compiled agent does not partition along the tool-call sequence. It partitions by *functional role*. The compiled agent is a large zero-LLM engine (`rdr_core.py`) with exactly *two* narrow retained ports, and the engine — not the model — decides what happens next and drives the whole tool chain (identity lookup, reads, fixes). The engine's invariant is stated in its architecture note: it is "the DETERMINISTIC core. Zero LLM, zero file I/O, stdlib only, pure (same state in → same decision out). Every rule carries its policy citation."

```
# Two ports, both narrow.  Neither is allowed to decide what happens next;
# that is rdr_core.py's job and it needs no model.
#   P1  DECODE_PROMPT    free-text retelling of a device tool's output -> fields
#   P6  REASSURE_PROMPT  the already-decided instruction -> one reassuring reply
decision = decide(st)   # rdr_core.py: the ONE deterministic entry point / turn
```

Listing 6: The partition. The LLM is confined to two ports that emit facts and wording; a single pure function `decide(st)` determines every action, and the LLM never calls a tool.

Because the manual's device diagnostics and fixes are carried out by the *user* (they leave the agent as text, not as agent tool calls), during a device chain there is often no agent tool available at all; the engine's job is to determine the next instruction and guide the user. The cut is therefore by role, not by the tool boundary: the model supplies the one thing no code can obtain — the customer's free-text retelling of what a device screen shows — and the code owns every table lookup, threshold, verdict, and sequencing decision.

D.1 The retained LLM port: the decode prompt

Port **P1** turns one customer message into closed JSON fields. It is kept out of the deterministic part's job *structurally*: its schema has no field for "next action", "verdict", or "what to say", so it

cannot express a decision. The system prompt, abridged:

```
You are a strict information extractor inside a telecom support system.
You do NOT talk to the customer, you do NOT decide what to do next, and you
do NOT diagnose anything. You convert ONE customer message into JSON facts.

Return ONE JSON object, nothing else. Every key is OPTIONAL: include a key ONLY
if this message actually reports it. Never guess. If the customer did not say
it, omit the key -- an omitted key means "not read yet", which is different
from any value.

  "device": { ... closed-vocabulary readings, e.g.
      "signal_strength": none | poor | fair | good | excellent,
      "sim_status":      active | missing | locked_pin | locked_puk,
      "airplane_mode":   on | off,   "apn_name": correct | incorrect, ... }
  "app_permissions": { ... }         # what an app is allowed to do
  "app_not_found":   true            # the named app is not installed
  "out_of_scope":    true            # ONLY when the request is something a
      # telecom agent cannot do AT ALL (not merely hard, slow, or needing a
      # colleague). If a telecom agent handles it, OMIT the key.

JSON only. No prose, no markdown fence, no explanation.
```

Listing 7: Port P1 `DECODE_PROMPT` (abridged from `rdr_prompts.py`). Facts only; every key optional; no field can carry a decision.

The second port, `REASSURE_PROMPT`, is wording-only: "the system has ALREADY decided what the customer must do; you may not change it, add to it, or leave it out." A compiled check (`_validate_reassure`) rejects any reply that names a tool the decision did not choose, leaks JSON, or adds a question — so even the wording port cannot smuggle in an action.

D.2 The deterministic engine

`rdr_core.py` is 5,582 lines of pure, stdlib-only Python; every rule carries its policy citation. Table 7 groups its machinery: fixed registries that enumerate the closed action space, permission/eligibility predicates that decide from state alone, and a single per-turn driver that walks the read→fix chain and assembles every tool call. Representative bodies follow (abridged).

```python
def _t_apn_bad(st, acc):
    # (a) manual:90 -- the READING says "incorrect", so reset the APN
    if dget(st, "apn_name") == "incorrect":
        return True
    if dget(st, "apn_name") is not None:
        return False                        # reading says OK
    if service_ok(st):
        return False                        # workflow:99 "persist"
    if dget(st, "airplane_mode") == "on":
        return False                        # Step 1.1 owns it
    sim = dget(st, "sim_status")
    if sim is not None and sim != "active":
        return False                        # Step 1.2 owns it
    if sim is None and not stalled_read(st, "sim_status"):
        return False                        # Step 1.2 not settled yet
    return stalled_read(st, "apn_name")     # workflow:99-107
```

Listing 8: A permit/deny trigger. Two policy clauses kept deliberately separate; an unknown reading (`None`) is not treated as any value — absence makes a sound.

Component	What it owns
`DEVICE_VOCAB`	The single enumeration of every closed device-reading field and its allowed values — the authority P1 writes into.
`STEPS, PROBE_STEP, CHAIN`	The operation registry: each diagnostic/fix step as `reads`, `trigger`, `gate`, `fix`, and a policy `cite`, composed into per-category chains.
`READ_SOURCE, TOOL_FIELDS`	Field → diagnostic-address map (one address per field).
`_t_apn_bad, _t_sim_missing, _t_airplane, ...`	Trigger predicates: "does this fault fire given state," each a closed comparison against `DEVICE_VOCAB` with a manual citation.
`_g_abroad, _g_data_prereq, ...`	Path-applicability gates: "is this step's path even in scope."
`adjudicate_suspension, adjudicate_usage`	Account-state verdicts (`not_suspended` / `cannot_lift_contract_expired` / ...; `within_limit` / `exceeded`).
`overdue_bills, contract_expired, quote_refuel, ...`	Compiled threshold and arithmetic lookups.
`plan_account_reads`	Identity/auth stage: a tagged union over which identity field is present; assembles the account tool call and its arguments.
`walk_chain`	The read→fix chain walker: the next action for a category is a pure function of (category, device record, ledger, account facts).
`decide`	The one deterministic entry point per turn (pure): terminal/transfer → out-of-scope → identity → confirmation gate → category loop.
`commit_instruction, ingest_tool_result, apply_decoded`	The single writers of device-ledger / account-world / decoded-state.
`render, transfer_summary`	Zero-token templated wording.

Table 7: The compiled telecom engine, by component. The two LLM ports supply facts and wording; every row is a deterministic function of the gathered state.

```python
def plan_account_reads(st):
    ...
    if acc["customer"] is None:
        phone = ident.get("phone_number"); cid = ident.get("customer_id")
        name, dob = ident.get("full_name"), ident.get("dob")
        if cid and not _ledger_hit(st, "acct:by_id"):
            return _acct(st, "get_customer_by_id", {"customer_id": cid},
                         "acct:by_id", "main_policy.md:95")
        if phone and not _ledger_hit(st, "acct:by_phone"):
            return _acct(st, "get_customer_by_phone", {"phone_number": phone},
                         "acct:by_phone", "main_policy.md:94")
        if name and dob and not _ledger_hit(st, "acct:by_name"):
            return _acct(st, "get_customer_by_name", {"full_name": name,
                         "dob": dob}, "acct:by_name", "main_policy.md:96-99")
        ...
```

Listing 9: The identity/auth stage assembles which account tool to call, and its arguments, from confirmed identity fields — in code, never by asking the model.

The chain walker `walk_chain(st, cat)` closes the loop: for the active category it reads what the next trigger needs, and once the trigger fires and the fix has not already been spent (a ledger check), it emits the tool instruction with its policy citation — returning `{"kind":"exhausted"}` only when the manual's steps are used up. Its own docstring states the criterion: "the next device/account action for this category is a pure function of (category, device record, ledger, account facts)."

E Per-Domain Compilation Traces

For each domain the trace below shows the decomposition and the substantive build/verify rounds produced by the LLM-as-AgentCompiler loop, each round naming only *what it changed*.

E.1 RuleArena — airline

```
Decomposition: one pass -> LLM parser (facts only) + zero-LLM fee calculator.
round 1: build the calculator -- base / oversize / overweight fees, cabin-gauge tests, per-bag
   sum.
round 2: verifier overturns the "first bag is never charged" convention; the free-cabin-item
   COUNT is moved into the calculator and derived from the dimensions.
round 3: remove the last fixed-position convention; reject the parser's boolean "exempt" key,
   keeping the observed noun class.
```

E.2 RuleArena — tax

```
Decomposition: one pass -> parser = fact extractor, calculator = full deterministic tax
   computation.
round 1: build compute_tax (Schedules 1/2/3/A/C/SE, education & child credits, tax-table +
   worksheet paths, thresholds as literals); parser input left as pass-through.
round 2: add a deny-list pre-port strip (drop policy tables, computed lines, non-fact bullets);
   fold two dead parser fields out of the emit set.
round 3: rebuild the strip as an allow-list (keep only fact lines), moving the compute-
   instruction narrative out of the model's input.
```

E.3 RuleArena — NBA

```
Decomposition: one pass; the decomposer defines the boundary itself -> LLM reads the roster facts
   , the engine applies the salary-cap rules.
round 1: fill sign_year on the signing contract, fixing a null-comparison crash.
round 2: correct the salary-increase cap when a historical team differs and the player joined by
   trade or waiver.
round 3: adopt the wide "Room" reading; return None for unpriced minimum-salary contracts; salary
    path fails loudly on a missing season.
round 4: add a draft-age field and the Over-38 salary-reallocation rule with its exemptions.
round 5: add the pick round to the fact space (a first-round-pick sale clause was tried and
   dropped).
round 6: add the mixed salary path (per-year rates fill, explicit values override).
round 7: route covered-term salary lookups to the effective term while the cap checks keep the
   stated term.
```

E.4 tau-bench — airline

```
Decomposition: one pass -> two LLM ports (a gatherer + a wording port); the engine owns gates,
   tables, arithmetic, and the tool-call assemblers.
round 1: from-scratch build; claims quarantined to one function, confirmations content-bound.
round 2: split flight-selection into compiled legality/ranking + a narrow re-validated selector
   port; add fare-delta, confirmation expiry, incremental bag fees, cabin & trip-type gates,
   route-inquiry and compensation paths.
```

```
round 3: make the bag-fee / endpoint lookups total (off-table membership no longer crashes); add
   a payment-quota gate on all change paths; remove the one-stop surface; wire deferred
   compensation.
```

E.5 tau-bench — retail

```
Decomposition: one pass -> two LLM ports + a deterministic engine.
round 1: emit the first full artifact (the two ports + the engine).
round 2: add the predicate/semantics port (numeric order, value references, tier chain).
round 3: gate resolver reads the (sentence, board) pair, not the sentence alone.
round 4: fill a destination from the single legal option.
round 5: batch a confirmation into one unit; handle same-order conflicts.
round 6: add give-up/stop, pick-one, and owned-scope handling.
round 7: choose the new-item id value (with a defer mechanism).
round 8: targeted verification of specific units + a regression guard.
round 9: fix a discarded revision that left its signature (an insisting customer is no longer
   scored silent).
round 10: scope the mind-change mechanism so it does not over-fire; require record-derivable
   fields to be computed rather than asked.
```

E.6 tau-bench — telecom

```
Decomposition: one pass -> a deterministic engine + two narrow ports (decode + reassure).
round 1: first build -- the engine and the two ports.
round 2: self-fix the located defects by widening dev coverage, with no target list.
round 3: attack the app-permission family of operations.
round 4: wire the artifact's own observables into the gold-free feedback channel.
round 5: bundle several rule and permission fixes into one arm.
round 6: re-run the frozen artifact with the served model held constant, removing a model
   confound.
```