PolicyC: Request-Conditioned Compilation of Large Policy Prompts

B. Garcia

July 2026

Abstract

Large language-model system prompts increasingly behave like policy programs: they combine safety constraints, privacy rules, tool requirements, artifact procedures, confirmation protocols, and response conventions. Supplying the entire policy on every request increases context and cost, while naive compression can remove a rare but decisive obligation. We study whether a large prompt P and request x can be compiled into a much smaller active slice P_x such that a model using P_x preserves the critical obligations satisfied by the same model using P.

We present PolicyC, a research prototype built around a validated graph of 43 structured policy nodes across six domains. A deterministic TypeScript compiler performs request and artifact-context matching, conservative retention, dependency closure, and compact prompt emission. A separate Python `asyncio` runtime executes paired full-policy and compiler-slice trials with bounded concurrency, hard call/token/cost ceilings, atomic raw-response persistence, resumability, hash-linked provenance, a rebuildable SQLite catalog, deterministic diagnostics, and strategy-blind semantic review.

We report three frozen held-out studies using a pinned GPT-5 mini snapshot: 160 cases, 960 model executions, and $2.657 in recorded cost. Compiled prompts reduced mean actual input by 89.69–98.23% and uncached-equivalent cost by 55.36–67.80%. However, conditional critical-obligation preservation ranged from 79.75% to 86.49%, below the preregistered 95% target. The latest 60-case study achieved 93.75% input reduction but only 79.75% preservation (Wilson 95% interval: 72.93–85.21%). Of 33 full-pass/compiler-fail pairs, 21 were attributed to lossy emitter wording, seven to selector errors, two to a context-interface asymmetry, and three primarily to model variance. The largest single defect converted conditional confirmation rules into an unconditional `ask_confirmation` directive after confirmation had already been supplied.

The experiments therefore support a mixed conclusion. Request-specific policy compilation reliably yields large efficiency gains in this synthetic system, but the current selector and text emitter do not preserve critical behavior well enough to replace the full prompt. The negative result localizes the central research problem: preserving predicates, satisfied preconditions, negation, and instruction precedence rather than merely retaining policy text.

1 Introduction

Production system prompts are growing from short behavioral instructions into heterogeneous policy bundles. A single prompt may state when live research is mandatory, which sources are authoritative, how destructive actions must be confirmed, which personal attributes must not be inferred, what tools may be called, how files must be inspected, and which response format is appropriate. Most user requests activate only a fraction of these rules, yet conventional inference supplies the full prompt for every request.

This creates an appealing systems question: can a request-conditioned compiler activate only the policies relevant to the current request? A successful compiler could reduce input tokens, cost, and latency while making the active contract easier to inspect. The failure mode, however, is asymmetric. Dropping redundant prose is harmless; dropping one confirmation, privacy, or tool-integrity rule can be severe. Average semantic similarity is therefore an inadequate target.

Generic prompt-compression methods remove low-information passages or tokens while preserving downstream task accuracy [1, 2, 3, 4]. Policy compilation has a different objective. It must preserve obligations whose importance is conditional on a specific request, tool surface, artifact, or risk. Related alignment work studies explicit principles and privileged instruction levels [5, 6], but does not directly test request-conditioned removal of privileged policy text.

We formalize the problem as follows. Let P be a policy program, x a user request plus optional structured context, and S a selector. The compiler emits

$$P_x = \text{closure}(S(P, x)), \tag{1}$$

where closure adds every transitive policy dependency. Let $C(P, x, y)$ indicate that answer y, produced under policy condition P, satisfies all independently specified critical obligations for x. The primary target is conditional preservation:

$$\theta = \Pr[C(P_x, x, Y_{P_x}) = 1 \mid C(P, x, Y_P) = 1]\,. \tag{2}$$

Conditioning on full-policy success is essential. If both conditions fail, the pair does not show that compilation preserved a successful obligation-bearing behavior. The primary efficiency target is

$$R_{\text{input}} = 1 - \frac{\mathbb{E}[T_{\text{input}}(P_x, x)]}{\mathbb{E}[T_{\text{input}}(P, x)]}. \tag{3}$$

This paper makes five contributions:

1. a typed representation of policy nodes, triggers, dependencies, obligations, and prohibitions;
2. a deterministic request-conditioned selector and dependency-closed prompt emitter;
3. a safety-bounded paired-experiment runtime with complete paid-call accounting and resumability;
4. three frozen held-out studies that separate efficiency from behavioral preservation; and
5. a failure taxonomy showing that conditional semantics and instruction precedence, rather than graph closure alone, dominate the remaining errors.

2 Related Work

Prompt compression. Selective Context filters prompt content using self-information [1]. LLMLingua introduces budget controllers and token-level compression for efficient inference [2]; LongLLMLingua adapts compression to long-context retrieval and position bias [3]; LLMLingua-2 frames task-agnostic compression as token classification distilled from a larger model [4]. These methods primarily evaluate answer quality, retrieval, or semantic fidelity. PolicyCinstead treats policies as typed program elements and evaluates independent obligations because infrequent policy text may be critical even when it appears semantically peripheral.

Principle-guided behavior and instruction priority. Constitutional AI demonstrates how an explicit written constitution can guide critique and preference learning [5]. The Instruction Hierarchy formalizes priority among system, developer, user, and tool-originated instructions and improves resistance to lower-trust conflicts [6]. PolicyCdoes not train a model or introduce a new hierarchy. It asks which privileged instructions can be omitted for one request without changing critical behavior.

Partial evaluation and program slicing. The compiler analogy is closer to partial evaluation than to generic summarization. A partial evaluator specializes a program using known inputs while preserving its semantics [7]. Program slicing retains statements that may affect a criterion. Policy prompts are not conventional programs, and language models are stochastic interpreters, but the analogy motivates explicit dependencies, conservative closure, and paired semantic tests. The experiments show where the analogy breaks: flattening a conditional policy into unconditional natural language can change behavior even when the correct node was selected.

Behavioral evaluation. Structural recall and behavioral preservation are distinct. A selector can retrieve every annotated node while the emitted wording weakens, strengthens, or misconditions those rules. PolicyCtherefore never treats its own selected nodes as behavioral ground truth. Each case carries independently declared obligations, and primary results come from strategy-blind answer comparison rather than selector metrics alone.

3 System Design

3.1 Policy representation

The current synthetic enterprise-agent prompt is represented as 43 manually structured nodes in six YAML packs: core behavior, current information and web use, files and artifacts, email and calendar, images, and writing. The graph grew from 39 nodes in compiler 0.5 to 43 in compiler 0.7 as held-out failures were converted into development requirements.

Each node contains a stable identifier, kind, severity, priority, triggers, optional dependencies, model-visible runtime instructions, obligations, prohibitions, and validator references. Node kinds distinguish universal rules, request-gated rules, and structural definitions. Zod runtime validation rejects unknown fields and invalid enum values. Graph validation rejects duplicate identifiers or edges, missing references, self-dependencies, cycles, invalid always-active nodes, unknown validators, and unreachable structural nodes.

The representation is intentionally explicit. The current system compiles a structured policy graph; it does not yet parse an arbitrary natural-language system prompt into policies automatically.

3.2 Selection and dependency closure

Given request x, artifact metadata, and the tool surface, the TypeScript compiler:

1. detects request intents using deterministic lexical rules;
2. activates universal policies;
3. matches triggers against request text, artifact type, operation, features, domain hints, risk hints, and tools;
4. conservatively retains high-impact safety, privacy, and tool-integrity rules;
5. traverses the transitive `requires` graph; and
6. orders selected nodes deterministically.

The closure operation is exact with respect to declared graph edges. This establishes dependency completeness, not behavioral completeness: an undeclared dependency or a lossy emitted instruction can still fail.

3.3 Prompt emission

The compiler serializes five experimental strategies: `full_policy`, `compiler_slice`, `kernel_only`, `direct_matches`, and `conservative_expanded`. The held-out studies compare only `full_policy` and `compiler_slice`.

Compiler 0.5 introduced a compact universal kernel to avoid repeatedly emitting honesty, privacy, hidden-reasoning, fabrication, trace-exposure, tool-simulation, and background-work policies. Compiler 0.6 removed a leaked internal tool-status directive and added policies derived from confirmed v0.5 regressions. Compiler 0.7 further hardened tool availability and selected regressions from v0.6. Despite those changes, v0.7 revealed that the emitter still converts some conditional policies into unconditional actions.

3.4 Cross-language boundary

TypeScript is the single authority for policy loading, validation, matching, graph closure, candidate construction, prompt emission, hashing, and artifact serialization. Python owns provider calls, concurrency, rate limiting, timeouts, retry classification, atomic persistence, evaluation, reports, and run-catalog storage. Versioned JSON Schemas define the boundary, and Python verifies candidate and compiled-prompt hashes before execution.

Trial identifiers bind the run, case, sample, strategy, model, provider, and candidate. This prevents the behavioral runtime from silently reconstructing a different compiler.

3.5 Safe paid execution

The live runtime is designed to fail closed. A dry run validates every payload and produces expected and worst-case spend. Paid execution requires an exact run-specific confirmation string. Hard ceilings cover provider attempts, input tokens, output tokens, tool calls, and dollars. Missing usage is never silently treated as zero.

Raw provider responses are atomically persisted before parsing. Completed trials resume without another provider call when provenance matches. Ambiguous network outcomes consume a call and reserved exposure and are not retried unless explicitly enabled. A local SQLite catalog stores run status, source commit, dataset and manifest hashes, model, trial counts, usage, costs, failures, and authoritative artifact paths. Immutable run directories remain the scientific source of truth and can rebuild the catalog.

The scheduler uses a bounded worker queue, an `asyncio` semaphore, and a sliding-window limiter. In the first 300-call held-out run, concurrency six completed the experiment in 8.35 minutes versus 49.12 minutes of summed provider-call latency, a 5.88× effective throughput ratio. This is an observed concurrency benefit, not a controlled sequential benchmark.

4 Evaluation Protocol

4.1 Case construction

Every case specifies the user request, artifact context, applicable obligations, critical obligation IDs, prohibitions, refusal expectation, tool expectation, and a semantic rubric. These labels are authored independently of the candidate prompt. Each case is executed under both strategies for the same

Table 1: Frozen held-out studies. "Pairs" counts complete paired outputs before semantic ungradables.

Compiler	Cases	Model calls	Complete pairs	Web searches	Cost	Output cap
0.5	50	300	129/150	0	$0.6881	2,048
0.6	50	300	139/150	41	$1.0357	2,048
0.7	60	360	180/180	20	$0.9333	3,072
Total	160	960	—	61	$2.6570	—

sample index and pinned model. Dispatch order is deterministically counterbalanced.

The studies use synthetic requests and a synthetic policy system. No private production data is included. Provider storage is disabled. The API key is loaded locally and excluded from run artifacts.

4.2 Outcome definitions

For a paired sample, four determinate critical outcomes are possible:

- both pass;
- full policy passes and compiler slice fails (FullOnly);
- compiler slice passes and full policy fails; or
- both fail.

The empirical conditional-preservation estimator is

$$\widehat{\theta} = \frac{n_{\text{both pass}}}{n_{\text{both pass}} + n_{\text{full only}}}. \tag{4}$$

We report trial-level Wilson 95% intervals and exact two-sided McNemar tests for discordant pairs. Because three generations from one case are clustered, these intervals and tests are descriptive and do not create three independent policy situations.

4.3 Deterministic and semantic evaluation

The deterministic evaluator checks case-local obligations plus a universal floor: no fake background work, hidden-reasoning disclosure, unsupported precision, raw tool traces, or simulated tool use. It also scores refusal and tool behavior. These checks are reproducible but lexically brittle.

Primary behavioral results use strategy-blind semantic packets. Packets contain opaque answer IDs, the request, obligations, allowed or required tools, and observed strategy-neutral tool evidence. They omit strategy, prompt size, latency, tokens, and cost. Grade hashes are locked before answer IDs are joined to a private strategy map. Reviewers were isolated Codex reviewer agents, not human annotators; this limits external validity and is stated explicitly.

4.4 Frozen studies

Table 1 summarizes the three held-out executions. Each compiler was frozen before its corresponding case set was opened to candidate compilation. Once results were inspected, that dataset became development evidence for later versions and was never reused as fresh held-out evidence.

All studies use `gpt-5-mini-2025-08-07`, two strategies, three samples per case, concurrency six, and no retries. Compiler 0.5 had no provider tools. Compilers 0.6 and 0.7 introduced web and synthetic

Table 2: Compiled-condition changes relative to the full prompt. Positive reductions are favorable; positive latency deltas are slower.

Compiler	Input reduction	Billed-cost reduction	Uncached-cost reduction	Latency delta
0.5	98.23%	23.01%	67.80%	−19.95%
0.6	89.69%	24.50%	55.36%	+0.39%
0.7	93.75%	12.14%	59.46%	+14.63%

Table 3: Primary strategy-blind semantic results. Compiler 0.6 includes three ungradable pairs, so strategy pass-rate denominators differ.

Compiler	Full pass	Compiled pass	Both	Full only	Compiler only	Both fail
0.5	107/129	101/129	92	15	9	13
0.6	74/138	73/136	64	10	9	53
0.7	163/180	141/180	130	33	11	6

function-tool cases, making their input and cost accounting harder but more representative of an agent policy surface.

5 Results

5.1 Efficiency

Table 2 shows the most stable result in the project. Every compiler reduced mean actual input by at least 89.69% and uncached-equivalent cost by at least 55.36%. Actual billed savings were smaller because the repeated full prompt benefited from prompt caching, while the compiled condition varied by request. Tool fees also matter: compiler 0.7 made 11 paid web searches versus nine under the full condition. Excluding search fees, its generation-only cost fell 19.75%, but the preregistered total billed-cost reduction was only 12.14%.

Latency did not improve consistently. Compiler 0.5 was 19.95% faster, compiler 0.6 was essentially unchanged, and compiler 0.7 was 14.63% slower. Output length, tool activity, cache state, and model reasoning all affect latency; input reduction alone does not guarantee wall-clock improvement.

5.2 Critical-obligation behavior

Tables 3 and 4 show that none of the held-out studies supports near-equivalence. Compiler 0.6 produced nearly identical marginal pass rates for full and compiled prompts, yet conditional preservation was only 86.49%. Equal averages can hide swapped successes: nine pairs passed only under the compiler while ten different pairs passed only under the full prompt.

Compiler 0.7 is a stronger negative result. The full prompt passed 90.56% of answers, whereas the compiled slice passed 78.33%. Conditional preservation fell to 79.75%, and 16 distinct cases contained at least one full-pass/compiler-fail result. The packet-level McNemar result is directionally significant, though a case-level sign test over 21 non-tied cases was $p = 0.0784$. Leave-one-case-out preservation ranged from 79.38% to 81.25%, indicating that no single case explains the result.

Table 4: Conditional preservation and uncertainty. Intervals are trial-level Wilson 95% intervals and remain descriptive because samples are clustered by case.

Compiler	Preservation	Wilson 95%	Distinct full-only cases	McNemar p
0.5	92/107 = 85.98%	78.15–91.32%	9	0.3075
0.6	64/74 = 86.49%	76.88–92.49%	6	1.0000
0.7	130/163 = 79.75%	72.93–85.21%	16	0.0013

Table 5: Compiler 0.7 held-out-v3 gates.

Gate	Threshold	Observed	Decision
Conditional preservation	$\geq 95\%$	79.75%	Fail
Wilson lower bound	$\geq 90\%$	72.93%	Fail
Distinct full-only cases	≤ 3	16	Fail
Mean input reduction	$\geq 90\%$	93.75%	Pass
Total billed-cost reduction	$\geq 15\%$	12.14%	Fail
Complete paired coverage	$\geq 90\%$	100%	Pass

5.3 Preregistered compiler 0.7 decision

Compiler 0.7 had explicit success gates fixed before execution. Table 5 records the decision without post hoc relaxation.

The compiler passed two of six gates. The study was operationally complete but behaviorally negative.

6 Failure Analysis

6.1 Compiler 0.5: machine directive leakage

Compiler 0.5 produced 15 full-only semantic regressions across nine case IDs. Eleven shared one emitter defect: a machine-oriented token resembling `report_unavailable_tool:web` appeared in model-visible text. The model copied or acted on this directive even when it was inappropriate. The remaining four regressions involved production-publish confirmation, destructive file overwrite, recurring-calendar scope, and legal-meaning preservation.

This result established that selecting the right high-level policy is insufficient if the prompt representation exposes internal control syntax. Removing the directive was a high-leverage compiler 0.6 remediation, but the counterfactual improvement was not reported as a new held-out result.

6.2 Compiler 0.6: tool and scope failures

Compiler 0.6 produced ten full-only semantic regressions across six cases. Eight were attributed to compiler or compiled-prompt defects: calendar responses omitted needed clarification or confirmation; image-generation responses treated an available tool as unavailable; external forwarding responses underspecified the recipient or privacy consequence; and one hidden-reasoning response omitted the required visible rationale. Two current-information cases were more consistent with stochastic evidence-quality failures.

The key lesson was that marginal equality is not preservation. Full and compiled pass rates were both about 53.6%, yet the paired result still contained ten lost full-prompt successes.

Table 6: Primary attribution for compiler 0.7 full-only regressions.

Cause	Pairs	Cases	Dominant examples
Emitter wording / semantic loss	21	8	Redundant confirmation, present-tense background-work fiction, forced Draft/Notes wrapper
Selector omission / misselection	7	4	Prompt-only image request treated as generation, missed image edit, forbidden spreadsheet call, "now" treated as current information
Context-interface asymmetry	2	1	Domain hint surfaced only in the compact condition and omitted from the blind packet
Primarily stochastic	3	3	Missing citation, invented legal detail, invented date

6.3 Compiler 0.7: conditional semantics

All 33 v0.7 full-only pairs were reviewed after the blind grade lock. Primary attribution is shown in Table 6.

The largest cluster contained 15 pairs across six already-confirmed Gmail or Calendar actions. The selected policies were relevant, but the emitter transformed "require confirmation before action" into the unconditional line `Required actions: ask_confirmation`. The model then requested a second confirmation instead of performing the authorized tool call. This is a semantic compilation bug: policy selection and dependency closure were structurally correct, but the emitted program ignored the satisfied precondition.

Three prompt-only image requests triggered image generation despite explicit "do not generate" language. One spreadsheet arithmetic request called a tool despite "do not use it." Three rewrite requests added a generic Draft/Notes wrapper despite "return only the rewrite." These cases show that negation and user-specified format precedence must be represented explicitly rather than left to keyword matching.

Two v0.7 judgments also exposed an evaluation-pipeline issue. The compiler surfaced a `work calendar` domain hint to the model, while the full condition and blind packet did not expose the same context. The locked grades remain primary. Treating those two compiled answers as passes in a sensitivity analysis raises preservation only to 80.98% and reduces distinct full-only cases from 16 to 15. Every behavioral gate still fails.

7 Operational Incidents and Integrity

Real provider execution exposed issues that offline tests did not.

Output truncation. Compiler 0.5 had 26 incomplete responses at the 2,048-token cap, leaving 129 of 150 complete pairs. Compiler 0.6 had 16 incomplete responses, leaving 139 pairs. Incomplete results remained failures and were never silently promoted on resume. Compiler 0.7 increased the cap to 3,072 and completed all 180 pairs.

Provider schema validation. Compiler 0.6 encountered six pre-model HTTP schema rejections for no-argument synthetic function tools. Compiler 0.7 encountered 36 pre-model rejections because `strict: true` was emitted for schemas with optional properties. The adapter was repaired before comparative results were inspected; rejected payloads reported no usage and were quarantined rather

than overwritten. Final accounting discloses 306 client attempts for v0.6 and 396 for v0.7 while retaining the authorized model-execution counts of 300 and 360.

Tool-cost accounting. The provider can return several search activity records for one billed search request. The runtime was amended to use provider-reported billed request counts instead of counting activity records. Tool costs are separated from input/output costs and included in the total billed-cost gate.

Resume correctness. An early resume path could reinterpret an incomplete persisted response as complete. The fix preserves the original terminal outcome and proves that a completed trial does not trigger another paid call. Stale derived evaluations were quarantined. Raw responses, trials, manifests, budgets, reports, and incident records remain hash-linked.

Blind-review provenance. For v0.7, three isolated reviewer agents graded 360 anonymous answers in three batches. The merged blind sheet contained exactly 180 packet IDs and 360 answer IDs with no missing critical verdicts. Its SHA-256 was committed and pushed before the private strategy map was opened. The semantic result can therefore be reconstructed without trusting an editable summary.

8 Discussion

8.1 What the evidence supports

The efficiency claim is robust within this synthetic setup. Three frozen studies, different case sets, and increasingly tool-rich prompts all produced roughly one to two orders of magnitude less input and more than 55% lower uncached-equivalent cost. This shows that most policy text is inactive for most requests and that deterministic slicing can exploit that sparsity.

The preservation claim is resolved negatively for the current architecture. Conditional preservation remained between 79.75% and 86.49%; no Wilson interval reached the 95% target. Compiler 0.7 regressed despite passing its targeted development suite. This is exactly why spent held-out failures cannot substitute for a new evaluation set.

8.2 The semantic gap

The main challenge is no longer graph traversal. Dependency closure can prove that declared prerequisite nodes were retained, but it cannot prove that the natural-language emitter preserved their semantics. Four constructs dominate the observed gap:

1. **Predicates:** a rule may apply only before an action or only when a precondition is absent;
2. **State:** confirmation, inspection, or authorization may already be satisfied in the request;
3. **Negation:** "do not use" and "prompt only" must suppress positive tool triggers; and
4. **Precedence:** an explicit output request must override a generic format template unless a higher-priority safety rule conflicts.

Flattening these constructs into an unordered list of "required actions" changes the policy program.

8.3 Why aggregate pass rates are insufficient

Compiler 0.6 provides the clearest example. Full and compiled marginal pass rates were almost identical, which could be presented as parity. Pairing revealed that the two conditions succeeded on different outputs. For policy replacement, a compiler-only pass does not cancel a full-only failure; the latter is a lost baseline success. Conditional preservation and exact regression provenance are therefore more informative than a difference in means.

8.4 Economics and caching

Prompt caching weakens the direct relationship between token compression and the user's bill. A repeated full prompt receives substantial cached-input discounts, whereas request-specific slices vary. Tool fees and longer compiled outputs can further offset savings. We therefore report three distinct quantities: actual billed cost, generation-only billed cost, and uncached-equivalent cost. The first describes the observed bill; the last better isolates the compiler's context effect.

9 Threats to Validity

Structured input. The original question concerns a large prompt P, but the compiler begins from manually structured YAML nodes. Automatic extraction of policies, dependencies, priorities, and predicates from arbitrary prose remains unsolved.

Single model and synthetic environment. All live studies use one pinned GPT-5 mini snapshot and a synthetic enterprise policy system. Results may differ across models, providers, temperatures, or real organizational prompts.

Changing held-out datasets. Each compiler was evaluated on a newly authored held-out set. This protects holdout discipline but prevents a controlled causal comparison between compiler versions. Cross-version trends reflect both compiler changes and case difficulty.

Clustered samples. Three generations per case measure model variability but do not create three independent situations. Trial-level Wilson intervals and McNemar tests are descriptive. Case-level sensitivity is reported where available.

Reviewer validity. Strategy blindness reduces one source of bias, but the semantic graders were Codex agents rather than human experts. The project does not claim human inter-annotator reliability. Legal, financial, and tool-use cases remain synthetic rubric tests, not professional judgments.

Evaluator and context interfaces. Deterministic validators have both false positives and false negatives. Blind packets can also omit context needed to judge a model-visible statement. The v0.7 domain-hint asymmetry demonstrates the need for a single symmetric context contract shared by both strategies and graders.

Execution amendments. Provider-adapter and accounting fixes occurred during v0.6 and v0.7 execution. They were narrowly scoped, documented before unblinding, and did not modify cases,

Table 7: Primary held-out provenance. Commit values identify the frozen source used to construct each run.

Compiler	Run ID	Frozen source commit
0.5	`run_f5f8e50b9931f65f`	`53fc3c01f77b64910d7fa5777eda502c383f5bc1`
0.6	`run_f3c3cdb0e2d1e368`	`0e2db600496aa3489da582f88a63f4da03b998ca`
0.7	`run_c019d419734c6c5e`	`d81862808f66b5fb80b5c3bcb2f06764a8a23f03`

compiled prompts, obligations, or model parameters. Nevertheless, a future confirmatory run should begin from the repaired adapter.

10 Toward Compiler 0.8

The next compiler should represent policy semantics rather than only selected text. A minimal intermediate representation would separate:

- activation predicates;
- required preconditions;
- evidence that a precondition is already satisfied;
- permitted, required, and forbidden actions;
- negated surface cues;
- instruction priority and format overrides; and
- user-visible rationale requirements.

The emitter should derive an action only when its predicate is true in the current state. For example, a confirmation policy should compile to `execute` when the exact target, scope, consequences, and user authorization are already present; it should compile to `ask` only when required fields remain unresolved.

All 33 v0.7 full-only pairs should become development regressions, including the two context-interface cases. Offline tests should assert selected nodes, emitted predicates, tool routing, and exact required/forbidden actions. A small paid development smoke may validate runtime behavior, but cannot estimate preservation. After compiler 0.8 is frozen, a newly and independently authored held-out-v4 set is required for fresh evidence.

Longer-term work includes automatic policy parsing, hybrid learned selectors with calibrated conservative fallback, formal checks over the intermediate representation, richer multi-turn tool harnesses, cross-model replication, external human review, and optimization for the smallest obligation-preserving slice rather than a trigger-selected slice.

11 Reproducibility

The public implementation is available at https://github.com/bgar324/policyc. Table 7 records the primary run identities. Each run directory contains the manifest, candidate artifacts, raw attempts, trials, budget ledger, report, blind packets, and private answer map. The SQLite catalog is rebuildable from these files.

Compiler 0.7's exhaustive blind grade lock is commit `f91bbf5`; its final execution audit and public summary are commit `cebff5a`. The primary derived hashes are:

- run report: `d3edf008aec1ae88eebde9e3634c2827a80ec394ccacd5846030e9ec31b91aed`;
- unblinded semantic results: `86bc56cf23ae1ee7e293aed23407f098d695e86fba220b91abb6811c8e625318`.

12 Conclusion

PolicyCreframes system-prompt reduction as policy compilation rather than generic compression. Across three frozen studies, request-specific slices consistently removed most input context and lowered uncached-equivalent cost. That result is practically meaningful: policy prompts contain substantial request-irrelevant structure.

The same studies reject the current replacement hypothesis. A model using the compiled slice did not preserve enough of the full prompt's critical successes, and compiler 0.7 failed four of six preregistered gates. Most regressions were traceable to compiler behavior rather than irreducible model randomness. In particular, a flat natural-language emitter lost conditional confirmation state, explicit tool negation, and output-format precedence.

The research outcome is therefore neither a product claim nor a dead end. PolicyC has established a reproducible experimental framework, a strong efficiency result, and a concrete semantic compiler agenda. The next question is no longer whether policy prompts can be made smaller; it is whether their predicates and priorities can be specialized without changing the obligations they impose.

References

[1] Y. Li. Unlocking Context Constraints of LLMs: Enhancing Context Efficiency of LLMs with Self-Information-Based Content Filtering. *arXiv:2304.12102*, 2023. https://arxiv.org/abs/2304.12102.

[2] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. *arXiv:2310.05736*, 2023. https://arxiv.org/abs/2310.05736.

[3] H. Jiang, Q. Wu, X. Luo, D. Li, C.-Y. Lin, Y. Yang, and L. Qiu. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. *arXiv:2310.06839*, 2023. https://arxiv.org/abs/2310.06839.

[4] Z. Pan, Q. Wu, H. Jiang, M. Xia, X. Luo, J. Zhang, Q. Lin, V. Rühle, Y. Yang, C.-Y. Lin, H. V. Zhao, L. Qiu, and D. Zhang. LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. *arXiv:2403.12968*, 2024. https://arxiv.org/abs/2403.12968.

[5] Y. Bai et al. Constitutional AI: Harmlessness from AI Feedback. *arXiv:2212.08073*, 2022. https://arxiv.org/abs/2212.08073.

[6] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions. *arXiv:2404.13208*, 2024. https://arxiv.org/abs/2404.13208.

[7] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.