Compile, Don't Memorize: A Context Compilation Architecture (CCA) for In-Context Learning

Anonymous ACL submission

Abstract

Large language models (LLMs) increasingly handle in-context learning (ICL) tasks where a long, novel context defines the rules, knowledge, and output schema for a series of questions. On benchmarks that grade against every detail of the context, even strong open-weights models pass only 12–16% of tasks: a single overlooked rule fails the whole response. We argue this brittleness is structural: the dominant "read-and-reason" paradigm asks the model to extract, plan, generate, and self-verify in one forward pass. We therefore ask whether explicit context compilation can fix it, how it compares to existing long-context strategies (gist retrieval, multi-agent self-play), and where the resulting harness benefit holds across task structure and model scale. We propose the **Context Compilation Architecture (CCA)**: compile the given context into a structured IR of rules, schemas, and exact terms; generate executable verifiers from the IR; run a checked reasoning loop that corrects any violation those verifiers flag. On CL-bench (1,899 tasks across 4 open base models), CCA outperforms vanilla prompting and two long-context baselines (ReadAgent-P, Ctx2Skill) on every base model, lifting Kimi K2.5 from **15.4% to 21.4%** with gains concentrated on rule-dense sub-categories.

1 Introduction

Modern LLMs are routinely deployed in scenarios where a single conversation contains tens to hundreds of thousands of characters of novel context (domain documentation, multi-agent workflows, custom DSLs, complex business rules), and the model is asked to comply with this context across many downstream questions. Benchmarks designed for this setting, such as CL-bench (Dou et al., 2026), grade responses against detailed *rubrics*: each of 5–20 criteria per question must be met, and a single missed constraint marks the task as failed. Empirically, even strong open base models pass only 12–16% of such tasks under vanilla prompting (Table 1).

We hypothesize that the issue is not raw reasoning capability but a saturation of the read-and-reason pipeline: any *single* constraint is usually within reach, yet the model is asked to perform extraction, planning, generation, and self-verification simultaneously in a single forward pass, and small details fall through the cracks. As context length grows and rubrics multiply, the joint probability of catching every required element collapses.

We draw inspiration from **movable type printing**. Before Bi Sheng's 11th-century invention, reproducing a new page required carving an entire fixed wooden plate that was expensive, inflexible, and single-use. Movable type changed this by introducing a two-stage workflow: a *compositor* first reads the new manuscript and selects individual type pieces into a custom plate; only then does the press mechanically reproduce the page. We argue current ICL methods resemble fixed-plate printing: the model carries pre-trained "plates" (memorized procedures) and tries to adapt them by force to whatever context arrives. Instead, we propose to *compile* the context: treat the raw input as a manuscript, extract the typed pieces (rules, entities, exact terms, output schema), and assemble them into an executable plate that mechanically verifies and produces the answer.

Concretely, we introduce the **Context Compilation Architecture (CCA)**: a four-stage pipeline that (i) compiles raw context into a structured intermediate representation (IR), (ii) emits Python verifiers from the IR, (iii) drafts a response using the IR as an explicit checklist, and (iv) when those verifiers detect concrete violations, regenerates with violation feedback. This converts the "did I remember everything?" question into "did my code-verifier pass?", a mechanical check that does not rely on the model's self-assessment.

CCA is, in modern terms, a concrete instance of *harness engineering* (Mysore, 2026; Böckeler, 2026; Masood, 2026): the emerging discipline that treats the LLM as a frozen reasoning utility and derives production reliability from the surrounding pipeline.

Research questions. We organize the paper around three questions any context-compilation harness must answer. **RQ1 (Effectiveness):** does explicitly compiling context into a structured IR with executable verifiers improve pass rate on rubric-graded ICL relative to vanilla prompting, and which CCA components are responsible for the improvement? **RQ2 (Comparison with existing strategies):** how does explicit context compilation compare to gist-memory retrieval (ReadAgent-P) and self-play skill libraries (Ctx2Skill) under rubric-based grading, and which design properties of each method are well- or poorly-matched to this evaluation regime? **RQ3 (Where and for whom):** where does the compilation harness help most, and how does the benefit scale with the underlying model's capacity and with the task's structural density?

Contributions. (RQ1) We introduce CCA, a context-driven compilation pipeline that auto-generates per-context verification tooling and improves rubric-graded ICL pass rate over vanilla prompting on every base model tested (§3, §4.5); we localise the load-bearing component via a controlled ablation (§5.1). (RQ2) We characterise how two strong long-context strategies (gist-memory retrieval and multi-agent self-play) interact with rubric-based grading and identify design properties well- or poorly-matched to this regime (§5.2). (RQ3) We isolate the task-structural and model-capacity factors that bound the harness benefit, through per-domain and per-sub-category breakdowns (§5.3).

2 Related Work

Long-context in-context learning. ReadAgent-P paginates a context, stores per-page gists, and at query time looks up and re-fetches relevant pages (Lee et al., 2024); this suits long-document QA where information is concentrated, but its gist step omits verbatim facts that rubric grading requires. Ctx2Skill casts context learning as skill augmentation via a Challenger/Reasoner/Judge/Proposer/Generator self-play loop producing a natural-language skill library retrieved at inference time (Si et al., 2026), with AutoSkill a related single-stage variant (Yang et al., 2026); the resulting library encodes procedural guidance rather than per-criterion obligations. Broader work combines retrieval, routing, and compression: RAG and variants vary retrieval granularity, order, or routing (Lewis et al., 2020; Jiang et al., 2024; Yu et al., 2024; Li et al., 2024b); LLMLingua-2 and ICAE compress prompts or encode memory slots (Pan et al., 2024; Ge et al., 2024); Chain-of-Agents distributes inputs across cooperating readers (Zhang et al., 2024). Empirical studies further show that even within nominal windows long-context models struggle with mid-context evidence (Liu et al., 2024) and that effective context lengths often fall short of claimed windows (Hsieh et al., 2024), motivating explicit structure extraction over positional access. CCA differs by compiling each context once into a typed IR that *preserves* exact terms and into executable verifier code that mechanically checks each constraint.

Code-as-reasoning and verification. Building on chain-of-thought prompting (Wei et al., 2022), PAL, PoT, and Chain-of-Code translate a problem into executable programs, delegating computation to an interpreter (Gao et al., 2023; Chen et al., 2023; Li et al., 2024a). LLM-as-judge made model-based evaluation a standard self-checking interface (Zheng et al., 2023), with extensions to iterative feedback, verbal-feedback retry, tool-interactive critiquing, reasoning checks, step-level verifier benchmarks, and synthesised Python verifier sets (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024; Miao et al., 2024; Jacovi et al., 2024; Pezeshkpour and Hruschka, 2026). In all of these, code is an answer-producing computation generated per question; in CCA, code is a per-context verifier generated once and reused to check whether a natural-language answer satisfies the context's typed rules, not to produce the answer.

Harness engineering and agent infrastructure. Practitioner discourse frames harness engineering as the execution environment, tool interfaces, contextual guides, lifecycle, observability, verification, and governance surrounding a frozen model (the ETCLOVG taxonomy) (Mysore, 2026; Böckeler, 2026; Masood, 2026; Seong et al., 2026; Li et al., 2026), arguing that the gap between production-ready agents and the $\sim$88% that never ship lies in this layer rather than in raw model capability (Masood, 2026). Adjacent infrastructure work supplies

reusable agent abstractions (AutoGen, AgentScope, MetaGPT, SWE-agent) (Wu et al., 2024; Gao et al., 2024; Hong et al., 2024; Yang et al., 2024). Closest in spirit, DSPy (Khattab et al., 2023) compiles declarative LM pipelines into self-improving programs by optimizing prompts and weights across a fixed program structure; CCA differs in unit of compilation, compiling each given context once into a typed IR and per-context Python verifiers rather than a fixed pipeline of LM calls. To our knowledge, CCA is the first method to treat in-context learning itself as a *compilation* problem (producing typed structure and executable verifier code from each given context), and to position the resulting pipeline within the harness-engineering discipline.

3 Context Compilation Architecture (CCA)

3.1 Overview

CCA processes a single *context group* (a context document plus one or more downstream tasks sharing that context, 1–12 tasks per context in CL-bench) in four stages (Figure 1). Stages 1–2 (Compiler, CodeGen) run **once per context**; stages 3–4 (Reasoner-1, Reasoner-2) run **once per task**.

3.2 Pipeline as a Composition

CCA's pipeline mirrors its two-phase structure in Figure 1: a ***per-context*** phase compiles a context c once into reusable artifacts, and a ***per-task*** phase reuses those artifacts to draft and conditionally correct one answer per downstream task. Given a raw context c, its task set $\mathcal{T}_c$ (each $t \in \mathcal{T}_c$ carrying a question q_t), and the instruction prompts $\mathcal{I}$ shared across the four LLM stages, the joint probability of all final outputs factorises as:

$$P(\{y_t\}_{t\in\mathcal{T}_c}|c,\mathcal{I}) = \underbrace{P(I_c|c,\mathcal{I}) \cdot P(\mathcal{M}_c|I_c) \cdot P(s_c|\mathcal{M}_c, c)}_{\textit{Per-context}\ (\text{F1, F3, F5})} \cdot \underbrace{\prod_{t\in\mathcal{T}_c} P(d_t|I_c, s_c, \tau(c), q_t) \cdot P(y_t|d_t, \mathcal{V}_c, \theta)}_{\textit{Per-task}\ (\text{F2, F4, F6, F7})} \quad (1)$$

where I_c is the typed JSON IR emitted by the Compiler ($\mathcal{C}$); $\mathcal{M}_c \subseteq \{\texttt{RC}, \texttt{FV}, \texttt{DA}\}$ is the verifier module set emitted by CodeGen ($\mathcal{G}$), with `RC`, `FV`, `DA` abbreviating `rule_checker`, `format_validator`, `data_analyzer`; $s_c = \texttt{DA}(c.\texttt{data})$ if $\texttt{DA} \in \mathcal{M}_c$ and $\varnothing$ otherwise (the cached summary is additionally gated by an IR-strategy condition before injection into Reasoner-1; see §3.5); $\tau(\cdot)$ is the head/tail context-truncation operator; $\mathcal{V}_c = \mathcal{M}_c \cap \{\texttt{RC}, \texttt{FV}\}$ is the draft-time verifier subset; and d_t is the draft emitted by Reasoner-1 ($\mathcal{R}_1$). The second per-task factor $P(y_t|d_t, \mathcal{V}_c, \theta)$ bundles draft-time verifier execution and conditional correction by Reasoner-2 ($\mathcal{R}_2$): it computes the violation list $v_t = \bigcup_{m\in\mathcal{V}_c} m(d_t)$, sets $y_t = \mathcal{R}_2(d_t, v_t)$ when $|v_t| \geq \theta$ (we use $\theta = 2$) and $\mathcal{R}_2$ returns non-empty text, and otherwise sets $y_t = d_t$. By convention every $P(\cdot)$ implicitly conditions on the corresponding LLM stage's instruction prompt in $\mathcal{I}$; we write $\mathcal{I}$ explicitly only on the first term to declutter the rest.

Equation 1 makes three structural properties of CCA explicit. *(i)* The per-context artifacts $(I_c, \mathcal{M}_c, s_c)$ are amortized over every task that shares c; only the per-task product re-fires as t varies. *(ii)* The IR I_c enters the Reasoner-1 prompt directly as a checklist (F2) and the cached pre-execution result s_c enters as a separate input block (F5). *(iii)* The draft-time verifiers $\mathcal{V}_c$ operate on d_t (not on c); `DA` does not appear in the violation list because it has already been consumed in producing s_c.

The labels F1–F7 attached to Eq. 1 are the unit of analysis in the ablation of §5.1: F1 Compiler, F2 IR-as-checklist injection, F3 CodeGen dispatcher, F4 draft-time verifier execution, F5 per-context `data_analyzer`, F6 Reasoner-2 correction loop, F7 head/tail truncation. A per-F-code definition table with section back-references is in Appendix A.7.

3.3 Stage 1: Compiler

The Compiler reads the raw context (concatenated system + user messages) and emits a JSON IR whose fields (`role`, `rules`, `knowledge`, `workflow`, `output_spec`, `available_tools`, `data_profile`, `compilation_meta`) are summarised in Appendix A.2. The Compiler prompt instructs the model to preserve fictional or counterfactual content verbatim (CL-bench contains contexts where domain facts are deliberately altered) and to extract every must/should/always/never statement as a rule. Two IR fields warrant emphasis: `knowledge.exact_terms` captures verbatim strings (agent role names, status codes, identifiers) that downstream rubrics typically require literally; and the `codeable` flag on each rule decides whether it will be checked mechanically by CodeGen (§3.4) or left to the Reasoner's semantic judgment.

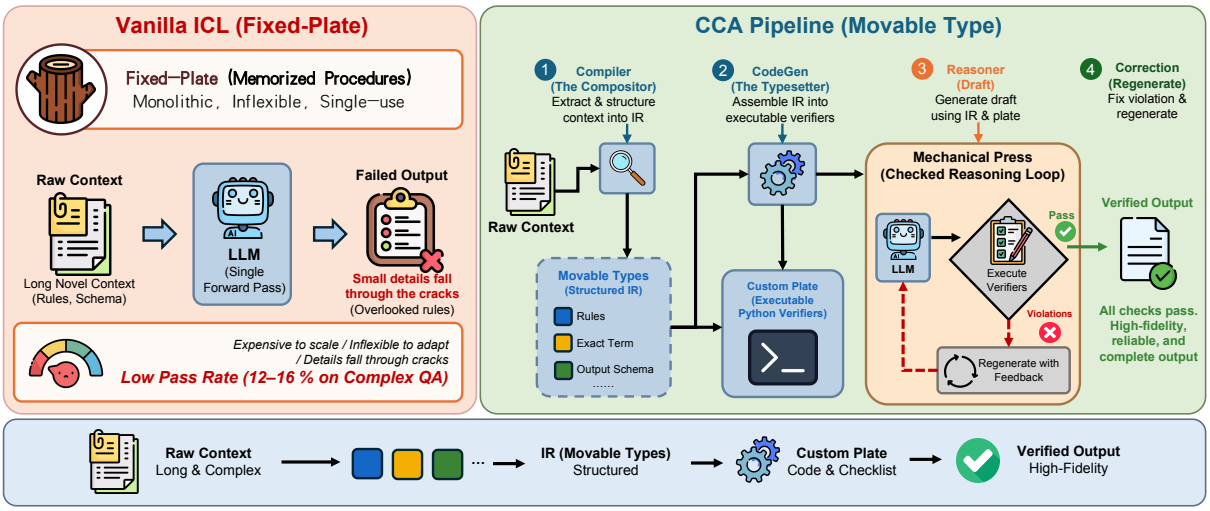

Figure 1: **CCA pipeline vs vanilla ICL. Left** (red, "Fixed-Plate"): vanilla ICL pushes the long context through a single LLM forward pass; small details fall through the cracks and pass rate stays at 12–16% on rubric-graded benchmarks (Table 1). **Right** (green, "Movable Type"): CCA decomposes the context into reusable typed pieces in four stages—(1) *Compiler* (*the compositor*) extracts a JSON IR of rules, exact terms, workflow, and output schema (F1); (2) *CodeGen* (*the typesetter*) assembles up to three Python verifiers from the IR (F3, F5); (3) *Reasoner* drafts an answer using the IR as an explicit checklist (F2); (4) *Correction* regenerates with violation feedback when verifiers flag ≥ 2 violations (F4, F6). Per-context artifacts (stages 1–2) are amortized over every question sharing the context; per-task stages (3–4) run once per question. Head/tail context truncation (F7) is described in §3. The pipeline converts the diffuse self-judgment "did I remember everything?" into the mechanical check "did my code-verifier pass?".

3.4 Stage 2: CodeGen

The dispatcher inspects three IR fields (the count of codeable rules, `data_profile.format`, and the number of `formatting_rules` in `output_spec`) and emits up to three Python modules. Two are draft-time verifiers that run *post-Reasoner-1* on the draft response and emit a violation list (each tagged with rule_id and evidence span): `rule_checker(response_text)`, emitted when the IR contains $\geq$1 rule with `codeable=True`; and `format_validator(response_text)`, emitted when the IR contains $\geq$3 `formatting_rules`. The third, `data_analyzer(raw_data)`, is qualitatively different: it is emitted when `data_profile.format` $\in$ {`tsv`, `csv`, `inline_table`}, runs at the *per-context* phase on the raw data block (not on any draft), and its cached summary is injected as a "CODE EXECUTION RESULTS" block in the Reasoner-1 prompt when the strategy is `data_analysis` or `calculator`. Each module is generated via a separate LLM call whose prompt emphasises false-positive avoidance (flagging a non-violation would cause Correction to "fix" a correct draft); modules are wrapped in `try/except` so any code failure produces an empty result rather than crashing the pipeline.

3.5 Stage 3: Reasoner-1

The Reasoner-1 prompt concatenates, in order, four blocks: (1) a **system prompt** (the `REASONER_SYSTEM` scaffold) that instructs the model to honor the context as the source of truth and the IR as a checklist of constraints; (2) a **compiled checklist** rendering of the IR (role persona, rules grouped by polarity, exact terms, workflow, output spec); (3) a conditional **code-execution-results** block, injected only when the IR's `compilation_meta.recommended_strategy` is `data_analysis` or `calculator` and the per-context `data_analyzer` module ran successfully (the draft-time verifiers *never* run at this stage); and (4) the **original context**, truncated to the model's window via a head-and-tail policy ($\sim$70% head, $\sim$30% tail, with a padding marker), followed by the user's actual question. This separation lets the Reasoner focus on producing a correctly-formatted response that addresses the question while respecting the IR's constraints, without re-deriving the structural information already captured in the IR or repeating cached data-summarisation work.

3.6 Stage 4: Verifier Execution and Reasoner-2 (Correction)

After Reasoner-1 produces a draft, the *Execute* step in Figure 1 runs the draft-time verifier modules (`rule_checker` and `format_validator`, whichever were emitted) against the draft, collecting all violations into a single list (the per-context `data_analyzer` module is not re-run here). If $n_{\text{viol}} \geq 2$ (chosen to suppress correction from a single noisy verifier hit), a Reasoner-2 *Correction* call fires with three inputs: the violation list (truncated to the first 10 entries, 200 chars each), the original draft as a prior assistant message, and strict instructions to make only minimal local edits. The corrected output replaces the draft only when it is non-empty; otherwise the draft is retained, so Correction can only *add* verified improvement, never silently break an otherwise-correct response.

Essential prompt cores for the four LLM-call stages and the IR schema spec are in Appendix A; the two-phase pipeline is also given as pseudocode in Appendix B (Algorithms 1–2).

4 Experiments and Results

4.1 Benchmark

CL-bench (Dou et al., 2026) contains 1,899 tasks distributed across four domains and 18 subcategories: Domain Knowledge Reasoning (DKR, 663 tasks), Rule System Application (RSA, 566), Procedural Task Execution (PTE, 471), and Empirical Discovery & Simulation (EDS, 199). Each task consists of (i) a long context (median 20K characters, max 247K), (ii) a question, and (iii) a list of 5–20 rubric criteria. The rubric format follows "must include / must not include / must be formatted as", with each criterion graded independently and the task counted as pass only if every criterion is satisfied. We use the official rubric-based evaluation script that delegates per-criterion grading to a strong judge LLM.

4.2 Base Models

We evaluate four open LLMs accessed via AWS Bedrock: **Kimi K2.5** (Kimi Team, 2026) (`moonshotai.kimi-k2.5`, 32B active / 1T total), **GLM-5** (GLM-5 Team, 2026) (`zai.glm-5`, 40B active / 744B total), **DeepSeek-V3.2** (DeepSeek-AI, 2025) (`deepseek.v3.2`, 37B active / 671B total), and **Qwen3-Next-80B** (Qwen Team, 2025) (`qwen.qwen3-next-80b-a3b`, 3B active / 80B total). All four support input windows long enough to accommodate CL-bench's 247K-character maximum, so no input truncation is needed at the model boundary. All models receive identical prompts; temperature is fixed at 0.0 across all methods for reproducibility; output cap is 8,192 tokens (16,384 for DeepSeek-V3.2 per its native maximum).

4.3 Baselines

We compare CCA against three baselines spanning the major ICL strategies (full porting details and adaptation notes in Appendix E):

Vanilla. Direct prompting: the full original context is sent to the base model and the response is returned with no preprocessing or post-processing.

ReadAgent-P (Lee et al., 2024). A long-context retrieval method using episode pagination, page-level gisting, and parallel look-up. We port the official Colab implementation, using its prompts verbatim (only the answer prompt is adapted from QuALITY's multiple-choice format to CL-bench's free-form format). Pagination and gisting are amortized per context.

Ctx2Skill (Si et al., 2026). A multi-agent self-play method using Challenger, Reasoner, Judge, Proposer, and Generator agents to evolve a skill library across iterations. We port the official codebase to Bedrock and run with the default configuration: 5 iterations $\times$ 5 tasks per iteration $\times$ 500 unique contexts, sharing the same four base models.

4.4 Evaluation Protocol

We evaluate all graded outputs using GPT-5.1 (`gpt-5.1`) as the judge with the official CL-bench script, which returns a per-criterion yes/no plus an overall 0/1 score. Inference temperature is fixed at 0.0 across all methods for reproducibility and clean ablation attribution, removing sampling noise from per-component deltas so that lifts can be attributed to design choices rather than run-to-run variance; the judge uses default sampling. As a side effect our absolute numbers sit modestly below the 23.7% best reported in the original CL-bench paper (Dou et al., 2026), which evaluates a partially different (larger, closed-model-inclusive) model pool under default sampling; CCA's lifts over Vanilla under our stricter temp$=0$ regime are correspondingly conservative. To prevent truncation bias we ran a two-pass audit (176 close-to-cap suspects re-run at each model's native maximum; post-audit no

Method	Model	DKR	RSA	PTE	EDS	Overall	Tok/task
Vanilla	Kimi K2.5	16.7	13.4	17.2	**12.1**	15.4	**11.9K**
	GLM-5	17.9	15.2	17.2	10.1	16.1	**12.0K**
	DeepSeek	16.6	13.4	16.1	**11.6**	15.0	**11.7K**
	Qwen3	14.2	11.0	10.4	**10.6**	11.9	**12.9K**
ReadAgent-P	Kimi K2.5	14.3	15.4	17.8	6.5	14.7	14.0K
	GLM-5	13.3	15.0	13.4	5.5	13.0	13.1K
	DeepSeek	13.9	11.7	16.6	6.5	13.1	13.2K
	Qwen3	12.8	**11.1**	12.7	2.5	11.2	15.2K
Ctx2Skill†	Kimi K2.5	19.3	13.3	15.1	10.1	15.5	193.9K
	GLM-5	17.6	13.3	14.6	**14.1**	15.2	191.4K
	DeepSeek	16.1	13.4	16.8	**11.6**	15.0	183.0K
	Qwen3	14.0	9.9	11.9	**10.6**	11.9	193.3K
CCA (ours)	Kimi K2.5	**21.1**	**21.2**	**26.1**	11.6	**21.4**	34.0K
	GLM-5	**23.5**	**19.8**	**25.1**	8.0	**21.2**	30.6K
	DeepSeek	**17.8**	**16.8**	**22.5**	9.0	**17.7**	29.3K
	Qwen3	**15.4**	9.7	**14.6**	4.5	**12.4**	34.1K

Table 1: Per-domain pass rate (%) and average tokens per task on CL-bench. Domain codes DKR / RSA / PTE / EDS are defined in §4. **Bold** pass rates flag the best score per (model, domain) cell (ties both bolded); bold tokens flag the cheapest method per row. Tok/task sums all LLM-call tokens (input + output) per task, with any per-context offline stage amortised over tasks-per-context (Ctx2Skill amortises its self-play stage across all 1,899 tasks; †its inference-only cost is ∼15K/task, comparable to Vanilla). CCA achieves the best (model, Overall) cell on every base model while spending ∼5.7× fewer tokens per task than Ctx2Skill, the other offline-prep+online-infer method in the suite; full cost decomposition in Appendix H.

output lies within 10% of its raised ceiling; full details in Appendix F). To verify CCA's lift is not a greedy-decoding artifact, re-running Full CCA on Kimi K2.5 at temperature 1.0 drops pass rate modestly to 18.75% (−2.65pp vs temp= 0) while still beating Vanilla at temp= 0 by +3.35pp (full breakdown in Appendix I).

4.5 Results

We organize the findings against the three research questions from §1. Table 1 reports pass rates for every (method, model, domain) cell plus the Overall pass rate over all 1,899 tasks.

RQ1 — effectiveness of context compilation (first pass). CCA achieves the highest Overall pass rate on every base model. On the three larger-activation models, CCA's lift over Vanilla is substantial: **+6.0pp on Kimi K2.5** (15.4→21.4%), **+5.1pp on GLM-5** (16.1→21.2%), and **+2.7pp on DeepSeek-V3.2** (15.0→17.7%). All three improvements are highly significant by paired McNemar test ($z = 6.00$, 4.89, 3.01; $p < 0.01$; test statistics in Appendix C.3). On the smallest model (Qwen3-Next-80B, 3B active) the lift drops to +0.5pp and is not significant ($z = 0.38$, $p = 0.70$); we revisit this model-capacity boundary as part of RQ3 in §5.

RQ2 — comparison with existing strategies (first pass). In every Overall column of Table 1, neither ReadAgent-P (gist-memory retrieval) nor Ctx2Skill (multi-agent self-play skill library) reaches CCA's score on any base model; the Tok/-task column further shows that Ctx2Skill's similar offline-prep+online-infer architecture costs ∼5.7× more tokens per task than CCA, visualised as a Pareto-dominance pattern in Figure 2. We unpack the design-property mechanism in §5.2 (porting details in Appendix E; cost decomposition in Appendix H).

RQ3 — where the lift holds (first pass). CCA's lift concentrates on rule-dense domains (PTE, RSA) where compiled rules and exact-term extraction directly support the rubric structure (CCA wins all four PTE columns and three of four RSA columns); DKR lift is smaller but still positive on every model. On open-ended EDS, CCA does not win on any model. The 18-sub-category lift table is in Appendix C; four CCA-only-wins case studies (one per domain) are in Appendix D; we synthesise the domain-density and model-capacity moderators in §5.3.

5 Discussion

This section synthesizes our findings against each of the three research questions from §1.

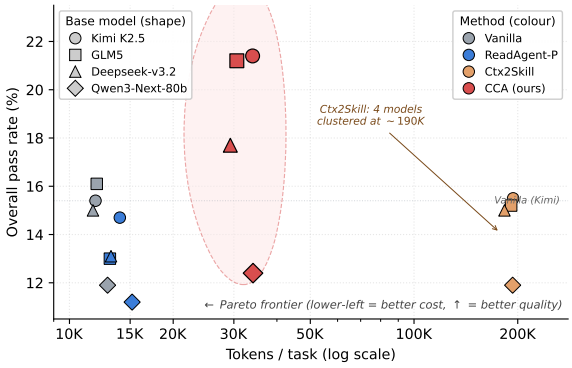

Figure 2: Cost–quality tradeoff for the 16 (method, model) cells of Table 1. Tokens per task on log x; Overall pass rate on y. **CCA (red) dominates the upper-middle Pareto frontier**: at $\sim 3\times$ Vanilla's tokens it adds $+6.0/+5.1/+2.7/+0.5$pp on Kimi/GLM-5/DeepSeek/Qwen3 (last is capacity-bounded; see §5.3). Ctx2Skill is the counter-example—$\sim 5.7\times$ CCA's cost, near-Vanilla pass rate—the lossy-abstraction failure of §5.2. A cost-conscious CCA variant (no R-2; $+4.72$pp at ~ 22.7K tokens) sits further left; see Appendix H.

5.1 Answering RQ1: Effectiveness of Context Compilation

RQ1 asked whether explicitly compiling context into a structured IR with executable verifiers improves rubric-graded ICL pass rate. The maintable results in §4.5 already answer affirmatively for every base model (significant by McNemar test on Kimi K2.5, GLM-5, and DeepSeek-V3.2). The deeper question is *which component* of CCA delivers the lift; we localise it via a controlled ablation on Kimi K2.5 against the full 1,899-task set, disabling one feature (or feature group) at a time from Full CCA (Table 2).

F2 is the load-bearing component. Reading the single-feature ablations in Table 2, **removing F2 (IR-as-checklist injection) causes the largest single drop** ($21.40 \rightarrow 18.33\%$, -3.07pp), nearly $2.5\times$ the drop from removing F6 (Reasoner-2 correction; -1.28pp). Surfacing the typed IR (rules grouped by polarity, verbatim exact-terms, output schema) inside the Reasoner-1 prompt is what most heavily lifts pass rate, even before any verifier code runs. F4+F6 jointly drop -1.81pp, so F4 alone (without correction) contributes the small remainder ~ 0.53pp: violations are logged but not acted on. Disabling the Compiler cascades F2/F4/F5/F6 off, dropping -4.18pp; the residual $+1.82$pp Vanilla-to-V6 gap measures the prompt scaffold plus F7 truncation alone. Re-combined in build-up order, these contributions form an additive decomposition that sums exactly to the $+6.00$pp gap: $+1.82$ (scaffold) $+ 2.37$ (F2) $+ 0.53$ (F4) $+ 1.28$ (F6); Figure 3 visualises this with the F2 step highlighted as the largest jump.

Where CCA's value lives. The IR-as-checklist injection is the dominant single mechanism; the correction loop is second; the verifier execution alone (without correction) is essentially a logging step; the Compiler is the enabling foundation, realised through F2/F4/F6 rather than as a standalone benefit. F2 also interacts positively with the F4/F6 loop ($+0.70$pp interaction term; derivation in Appendix G), so F2's measured contribution is larger when removed from Full CCA (-3.07pp) than when added on top of scaffold-only ($+2.37$pp). Per-domain ablation, fire rates, token costs, and the build-up visualisation (Figure 3) are in Appendix G; the cost-quality dial is in Appendix H.

5.2 Answering RQ2: Why Existing Strategies Don't Improve

RQ2 asked how context compilation compares to existing long-context strategies and which design properties of each are well- or poorly-matched to rubric-based grading. The shared failure mode of the baselines is a *lossy abstraction step* between context and reasoner: ReadAgent-P's gisting summarises away verbatim facts; Ctx2Skill's skill distillation summarises away per-criterion checks. Rubric grading penalises both losses (it requires verbatim reproduction of specific terms and binary satisfaction of each criterion), so neither baseline can match a method that keeps those signals intact. CCA's design is the inverse: the IR preserves verbatim `exact_terms` and CodeGen produces per-context executable checks; both are tightly coupled to what the rubric tests, which is why CCA beats both baselines on every (model, Overall) cell of Table 1 and dominates the Pareto frontier of Figure 2: the off-frontier baselines pay extra tokens over Vanilla but return near-zero lift, while CCA converts its $\sim 3\times$ Vanilla cost into the only large positive lift (full positioning vs all three baselines in Appendix H). Appendix D traces four CCA-only-wins (one per domain) end-to-end, showing exactly which IR field or verifier module differentiates CCA from each baseline on the same task.

Configuration	Pass	Acc.	Δ vs Full	Δ vs Vanilla
Full CCA (all features on)	**406**/1899	**21.40%**	—	**+6.00**
Single-feature ablations				
CCA w/o F6 (Reasoner-2 correction loop)	382/1899	20.12%	−1.28	+4.72
CCA w/o F2 (IR-as-checklist injection)*	348/1899	18.33%	**−3.07**	+2.93
Multi-feature ablations				
CCA w/o F4 + F6 (verifier execution + correction)	372/1899	19.59%	−1.81	+4.19
CCA w/o F1 cascade (Compiler off → F2/F4/F5/F6 disabled)	327/1899	17.22%	−4.18	+1.82
Vanilla (no CCA pipeline; default prompting)	292/1899	15.40%	−6.00	—

Table 2: Ablation on Kimi K2.5 over the full 1,899 CL-bench tasks. Each row removes one CCA feature (or group) from Full CCA, keeping the rest active. *Removing F2 (IR-as-checklist injection) causes the *single largest drop* (−3.07pp), identifying F2 as the most load-bearing feature. F-codes are defined in Appendix A.7; the four drops recombined as an additive decomposition (§5.1) sum exactly to the +6.00pp Full-vs-Vanilla gap.

5.3 Answering RQ3: Where the Compilation Harness Holds

RQ3 asked where the harness benefit holds, and how it depends on the underlying model's capacity and the task's structural density. Two moderators emerge.

Moderator 1 (task structural density). The largest sub-category lifts cluster where the context defines an explicit rule system (full 18-sub-category lift table in Appendix C.2): **Legal & Regulatory** (RSA, $n = 92$) gains +20 to +27pp across all models (Compiler extracts statute lists and jurisdictions as `exact_terms`); **Management** (DKR, $n = 112$) gains +11 to +25pp (the IR's `rules` and `output_spec.persona_voice` together encode the management persona's decision framework); **Workflow Orchestration** (PTE, $n = 229$) gains +6 to +11pp across all four models (rich `available_tools` and `workflow` steps are captured explicitly). Conversely, CCA underperforms Vanilla on open-ended creative tasks (e.g. Game Mechanics, Lifestyle, Simulation Environment), where the compiled IR acts as a Procrustean bed: the Reasoner produces a structurally clean but content-narrow response while Vanilla rambles freely and happens to cover more of what the grader expects. We view this not as a failure of the compilation principle but as a need to vary IR strictness as a function of `compilation_meta`'s recommended strategy.

Moderator 2 (model capacity). The lift on Qwen3-Next-80B (3B active) drops to +0.5pp and is not significant ($p = 0.70$), suggesting that harness benefit is bounded by the underlying model's capacity to integrate structured supplementary signal: a small-activation model cannot fully consume an IR checklist plus injected code-execution results *in addition to* the original context. On Kimi K2.5 the Reasoner-2 correction fires on ∼62% of tasks (Appendix G, Table 8); on the small-activation model the corrected output is empty more often (12 empty-output cases concentrated on Qwen3 in our audit, Appendix F), so the draft is retained; the conservative "only add verified improvement" design effectively no-ops on Qwen3. Model-capacity scaling laws are left to future work (§7).

6 Conclusion

CCA compiles novel context into a structured IR, generates executable verifier code from it, and runs a checked reasoning loop with code-grounded correction. On CL-bench, CCA attains the highest pass rate on every base model: +6.0/ + 5.1/ + 2.7pp over Vanilla on Kimi K2.5 / GLM-5 / DeepSeek-V3.2 (all $p < 0.01$), and a smaller, non-significant +0.5pp on Qwen3-Next-80B, the capacity-bounded case (§5.3); CCA also outperforms ReadAgent-P and Ctx2Skill. The harness-engineering framing positions CCA as an auto-derived production harness around a frozen base LLM, supporting the broader claim that progress on long-context rubric-graded benchmarks benefits more from compiling structure *out of* the context than from training models to better self-recall.

7 Limitations

Scaling the ablation. Our component-level isolation (§5.1) is run on Kimi K2.5 over the full 1,899-task set. Running the same isolation across the other three base models (particularly probing the

F2/F6 interaction at smaller activation, §5.3) would sharpen the model-capacity boundary at which harness benefit saturates, and is a natural extension of this work.

Schema coverage. The current IR captures the structural primitives that recur across rule-heavy, procedural, and data-analysis contexts: rules, exact terms, output specs, agent workflows, and tabular data profiles. We see direct extensions to contexts whose relevant structure is *spatial* (geometric reasoning, layout) or *temporal* (long-horizon planning, event ordering) .

Compiler robustness. Unparseable IRs occurred for $< 1\%$ of context groups in our evaluation, all absorbed by graceful fallback to direct prompting; hardening this into a formally verified property for safety-critical adversarial deployments is left to future work.

References

Birgitta Böckeler. 2026. Harness engineering for coding agent users. martinfowler.com.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research (TMLR)*.

DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Shihan Dou, Ming Zhang, Zhangyue Yin, Chenhao Huang, Yujiong Shen, Junzhe Wang, Jiayi Chen, Yuchen Ni, Junjie Ye, Cheng Zhang, Huaibing Xie, Jianglu Hu, Shaolei Wang, Weichao Wang, Yanling Xiao, Yiting Liu, Zenan Xu, Zhen Guo, Pluto Zhou, and 8 others. 2026. CL-bench: A benchmark for context learning. *arXiv preprint arXiv:2602.03587*.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. AgentScope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. 2024. In-context autoencoder for context compression in a large language model. In *International Conference on Learning Representations*.

GLM-5 Team. 2026. GLM-5: from vibe coding to agentic engineering. *Preprint*, arXiv:2602.15763.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. RULER: What's the real context size of your long-context language models? In *Proceedings of the First Conference on Language Modeling (COLM)*.

Alon Jacovi, Yonatan Bitton, Bernd Bohnet, Jonathan Herzig, Or Honovich, Michael Tseng, Michael Collins, Roee Aharoni, and Mor Geva. 2024. A chain-of-thought is as strong as its weakest link: A benchmark for verifiers of reasoning chains. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4615–4634.

Ziyan Jiang, Xueguang Ma, and Wenhu Chen. 2024. LongRAG: Enhancing retrieval-augmented generation with long-context llms. *arXiv preprint arXiv:2406.15319*.

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. 2023. DSPy: Compiling declarative language model calls into self-improving pipelines. *Preprint*, arXiv:2310.03714.

Kimi Team. 2026. Kimi K2.5: Visual agentic intelligence. *Preprint*, arXiv:2602.02276.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. 2024. A human-inspired reading agent with gist memory of very long contexts. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2024a. Chain of code: Reasoning with a language model-augmented code emulator. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Jianyang Gu, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Tianyang Wang, Jihun Hamm, Smita Krishnaswamy, Jun Huan, and Chandan K. Reddy. 2026. Agent harness engineering: A survey. OpenReview preprint.

Zhuowan Li, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. 2024b. Retrieval augmented generation or long-context LLMs? a comprehensive study and hybrid approach. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 881–893.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics (TACL)*, 12:157–173.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*.

Adnan Masood. 2026. Agent harness engineering—the rise of the ai control plane. Medium.

Ning Miao, Yee Whye Teh, and Tom Rainforth. 2024. SelfCheck: Using llms to zero-shot check their own step-by-step reasoning. In *International Conference on Learning Representations*.

Vishal Mysore. 2026. Harness engineering for ai agents in 2026. Medium.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 963–981.

Pouya Pezeshkpour and Estevam Hruschka. 2026. AutoPyVerifier: Learning compact executable verifiers for large language model outputs. *arXiv preprint arXiv:2604.22937*.

Qwen Team. 2025. Qwen3-Next-80B-A3B. Hugging Face model card.

Haebin Seong, Li Yin, Haoran Zhang, and Zhan Shi. 2026. The last harness you'll ever build. *arXiv preprint arXiv:2604.21003*.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shuzheng Si, Haozhe Zhao, Yu Lei, Qingyi Wang, Dingwei Chen, Zhitong Wang, Zhenhailong Wang, Kangyang Luo, Zheng Wang, Gang Chen, Fanchao Qi, Minjia Zhang, and Maosong Sun. 2026. From context to skills: Can language models learn from context skillfully? *arXiv preprint arXiv:2604.27660*. Introduces the `Ctx2Skill` self-evolving framework.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling next-gen llm applications via multi-agent conversation. In *Proceedings of the First Conference on Language Modeling*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. 2026. AutoSkill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*.

Tan Yu, Anbang Xu, and Rama Akkiraju. 2024. In defense of RAG in the era of long-context language models. *arXiv preprint arXiv:2409.01666*.

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö. Arik. 2024. Chain of agents: Large language models collaborating on long-context tasks. In *Advances in Neural Information Processing Systems*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

A Implementation Reference: Prompt Cores and IR Schema

This appendix documents the essential portions of each LLM-call prompt used in CCA, plus the IR schema produced by the Compiler. Boilerplate (formatting reminders, "no markdown fences", general politeness) is elided; the goal here is to expose the *design-relevant* instructions that, if changed, would change pipeline behaviour. Full prompts and code are released with the supplementary material.

A.1 Compiler Prompt — Essential Sections

The Compiler is the load-bearing per-context stage. Three sections of its prompt are essential for the pipeline's behaviour:

(1) Context fidelity (counterfactual preservation).

```
Context may contain fictional/modified/
counterfactual knowledge. Extract
    EXACTLY
as stated, even when contradicting your
training. Do not "correct" or skip
    rules
that seem unusual.
```

This instruction is what lets CCA pass CL-bench tasks whose contexts deliberately use counterfactual rules.

(2) `exact_terms` extraction.

```
Rubrics frequently check for VERBATIM
appearance of specific strings from
the context. Extract every term that
downstream output must reproduce
literally:
- Agent role identifiers
- Tool/command names
- Status codes / enum values
- Specific identifiers (e.g. MEDIA-002)
- Rating tier names
- Exact phrases the persona must use

DO NOT paraphrase or generalize ---
the rubric is checking for the
literal string.
```

This drives the dominant ablation finding: F2 (IR-as-checklist injection) is the largest single contributor, primarily because the `exact_terms` field surfaces strings the rubric will literally check for.

(3) `output_spec.format_type` classification. The Compiler tags each context with one of {`natural_artifact`, `tool_sequence`, `structured_data`, `hybrid`} (see Table 3), used by Reasoner-1 to choose response format. The classifier prompt enumerates examples for each label and defaults to `natural_artifact` on uncertainty.

IR field	Contents
`role`	persona, voice, tone
`rules`	`must_do`/`must_not`/`conditional`; each with a `codeable` flag and `code_hint`
`knowledge`	concepts, entities, formulas, `exact_terms` (verbatim strings rubrics require)
`workflow`	ordered procedural steps and decision points
`output_spec`	`format_type` ∈ {`natural_artifact`, `tool_sequence`, `structured_data`, `hybrid`}, tone, `persona_voice`, `formatting_rules`, length
`available_tools`	tool name, purpose, I/O schema, when-to-call
`data_profile`	`none` / `tsv` / `csv` / `inline_table` + optional analysis hints (consumed by CodeGen)
`compilation_meta`	recommended strategy ∈ {`rule_engine`, `calculator`, `data_analysis`, `soft_reasoning`} (consumed by Reasoner)

Table 3: IR fields emitted by the Compiler.

A.2 IR Schema

Table 3 summarises every top-level field of the JSON IR emitted by the Compiler. Each rule under `rules.must_do/must_not/conditional` carries an `id`, the verbatim `rule` text, the `source` quote it was extracted from, a boolean `codeable` flag (whether CodeGen should attempt to verify it mechanically), and an optional `code_hint` string (a regex or check the CodeGen prompt is encouraged to reuse). Each item in `knowledge.exact_terms` carries a `category`, the verbatim `term`, and an `example_use` describing how the rubric is expected to test for it.

A.3 CodeGen Prompt Cores

CodeGen emits up to three Python modules. The single design property they share is *false-positive avoidance*:

```
NO FALSE POSITIVES. Only flag a
violation if you are CERTAIN it's
violated. When in doubt, skip the
rule. False positives are catastrophic
--- they cause the downstream Reasoner
to "fix" things that aren't broken.
```

rule_checker. Generates `check_rules(response_text) -> dict` returning a violation list. The prompt enumerates common false-positive patterns to avoid: negated assertions, phrases inside quoted text, case sensitivity (default insensitive), word boundary leakage (e.g., matching "art" inside "start").

format_validator. Generates `check_format(response_text) -> dict` for formatting rules: "must end with X" → regex anchored at end-of-string; "must include JSON field Y" → `json.loads` + key check; "must be < 200 words" → word-count split.

data_analyzer. Generates `analyze_data(raw_data) -> dict`. Unlike the previous two, this module is executed at the per-context phase on the raw data block. Its core design property is *defensive parsing* (real-world data is messy):

```
DEFENSIVE PARSING --- try multiple
strategies in order: strip
BOM/whitespace; auto-detect delimiter
(tab, comma, semicolon, space, pipe);
pandas with sep=None,
engine='python', on_bad_lines='skip';
manual csv module on failure; per-row
column-count validation.
```

The module is wrapped in a top-level `try/except` that always returns partial results (never raises) so a malformed data block cannot poison the downstream pipeline.

A.4 Reasoner-1 System Prompt — Essential Sections

The system prompt is the constant component of every Reasoner-1 call. Two sections do the heavy lifting:

(1) Context-first principle (counterfactual handling).

```
The CHECKLIST and ORIGINAL CONTEXT are
the AUTHORITATIVE source. When context
contradicts your prior knowledge ---
TRUST THE CONTEXT.
- Do NOT add disclaimers
  ("in reality...", "actually...",
   "note that the standard X is...")
- Do NOT correct, qualify, or
  fact-check the context using your
  training knowledge
- Apply the rules AS GIVEN, even if
  they seem strange.
```

This mirrors the Compiler's context-fidelity instruction at the Reasoner level: the structured IR alone is not enough — the Reasoner must *accept* the counterfactual content rather than "fix" it.

(2) Rule precedence within context.

```
A specific situational rule beats a
general one. An explicit example in
context beats a generic principle
from your training. The rule that
mentions the EXACT entities/conditions
in the task wins.
```

This is the conflict-resolution layer for contexts containing overlapping rules, common in legal/regulatory and management domains.

A.5 Reasoner-2 Correction Prompt

Correction is invoked as a follow-up turn with the original Reasoner-1 messages still in the conversation, plus the Reasoner-1 draft appended as a prior `assistant` turn, plus a new `user` message containing the violation feedback:

```
## CODEABLE/FORMAT VIOLATIONS (n):
- [R3] {rule text, <=200 chars}
    Evidence: {snippet, <=200 chars}
- [N1] {rule text, <=200 chars}
    Evidence: ...
...

## YOUR DRAFT:
{Reasoner-1 draft, <=6000 chars}

These are objective, code-verifiable
violations. Revise your response to
fix them precisely. Do NOT make other
changes --- keep the structure, tone,
and content otherwise identical.
Output the revised response only.
```

The violation list is truncated to the first 10 violations (each <=200 characters for both rule text and evidence span) and the draft is truncated to <=6000 characters, to stay within the model's prompt budget. The instruction to make "only the flagged edits" is essential: it is what makes correction monotone — the corrected output strictly fixes at least one verified violation, rather than reorganising the draft in ways that might introduce new ones.

A.6 Head/Tail Context Truncation (F7)

When the original context block exceeds the Reasoner's input budget (set to 200,000 characters in our runs), we truncate it as follows: keep the first 70% of the budget as "head" (preserves the system prompt and rule definitions, which are typically front-loaded in CL-bench contexts) and the last 30% as "tail" (preserves the user's actual question and any most-recent turn), with

a 200-character padding marker `"[. . . context truncated . . . ]"` inserted between them. This policy is the F7 component of the ablation (§5.1); its contribution on Kimi K2.5 is folded into the prompt-scaffold step (+1.82pp Vanilla → V6).

A.7 F-Code Definitions for the Ablation

The labels F1–F7 attached to Eq. 1 and used throughout §5.1 are defined here with cross-references to their detailed treatment in the main text.

- **F1 — Compiler** (§3.3): per-context LLM call that emits the typed JSON IR I_c.
- **F2 — IR-as-checklist injection** (§3.5): renders I_c (rules, exact terms, workflow, output spec) into the Reasoner-1 prompt.
- **F3 — CodeGen dispatcher** (§3.4): per-context LLM call(s) deciding the module set $\mathcal{M}_c$ and generating each module's Python source.
- **F4 — Draft-time verifier execution** (§3.6, the "Execute" step): applies the modules in $\mathcal{V}_c$ to the Reasoner-1 draft and produces the violation list v_t.
- **F5 — Per-context `data_analyzer`** (§3.4): runs once on the raw context's data block in the per-context phase; its cached summary s_c is injected into Reasoner-1's prompt as a separate block when the IR's recommended strategy is `data_analysis` or `calculator`.
- **F6 — Reasoner-2 correction loop** (§3.6): regenerates the response when $|v_t| \geq \theta$, with v_t and d_t as input.
- **F7 — Head/tail context truncation policy** (§3.5): $\tau(\cdot)$ keeps ∼70% head and ∼30% tail of the original-context block in the Reasoner-1 prompt.

B Algorithm Pseudocode

We present pseudocode for the two phases of CCA introduced in §3.2 and visualised in Figure 1. The notation follows Equation 1: $\mathcal{C}, \mathcal{G}, \mathcal{R}_1, \mathcal{R}_2$ are LLM-call operators; $\mathcal{I}$ is the bag of instruction prompts; $\texttt{RC}, \texttt{FV}, \texttt{DA}$ are the three Python verifier modules; $\tau(\cdot)$ truncates the original context to fit the Reasoner-1 window; and $\theta = 2$ is the violation threshold.

Algorithm 1 CCA Per-Context Phase

Require: raw context c; instruction prompts $\mathcal{I}$
Ensure: per-context artifacts $(I_c, \mathcal{M}_c, s_c)$
1: $I_c \leftarrow \mathcal{C}(c \mid \mathcal{I})$ {**F1**: Compiler emits JSON IR}
2: $\mathcal{M}_c \leftarrow \emptyset$ {**F3**: CodeGen dispatcher}
3: **if** $\exists\, r \in I_c.\texttt{rules}$ with $r.\texttt{codeable} = \text{True}$ **then**
4: $\quad \mathcal{M}_c \leftarrow \mathcal{M}_c \cup \{\texttt{RC} \leftarrow \mathcal{G}_{\texttt{RC}}(I_c)\}$
5: **end if**
6: **if** $|I_c.\texttt{output_spec.formatting_rules}| \geq 3$ **then**
7: $\quad \mathcal{M}_c \leftarrow \mathcal{M}_c \cup \{\texttt{FV} \leftarrow \mathcal{G}_{\texttt{FV}}(I_c)\}$
8: **end if**
9: **if** $I_c.\texttt{data_profile.format} \in \{\texttt{tsv}, \texttt{csv}, \texttt{inline_table}\}$ **then**
10: $\quad \mathcal{M}_c \leftarrow \mathcal{M}_c \cup \{\texttt{DA} \leftarrow \mathcal{G}_{\texttt{DA}}(I_c)\}$
11: **end if**
12: **if** $\texttt{DA} \in \mathcal{M}_c$ **then**
13: $\quad s_c \leftarrow \texttt{DA}(c.\texttt{data})$ {**F5**: pre-Reasoner-1 execution}
14: **else**
15: $\quad s_c \leftarrow \varnothing$
16: **end if**
17: **return** $(I_c,\ \mathcal{M}_c,\ s_c)$

Algorithm 2 CCA Per-Task Phase

Require: per-context artifacts $(I_c, \mathcal{M}_c, s_c)$; original context c; question q_t; instruction prompts $\mathcal{I}$; threshold $\theta = 2$
Ensure: final answer y_t
1: $\widetilde{c} \leftarrow \tau(c)$ {**F7**: ∼70% head, ∼30% tail}
2: **if** $I_c.\texttt{strategy} \notin \{\texttt{data_analysis}, \texttt{calculator}\}$ **then** $s_c \leftarrow \varnothing$ {strategy gating}
3: $d_t \leftarrow \mathcal{R}_1(I_c,\ s_c,\ \widetilde{c},\ q_t \mid \mathcal{I})$ {**F2**: IR-as-checklist injection}
4: $\mathcal{V}_c \leftarrow \mathcal{M}_c \cap \{\texttt{RC}, \texttt{FV}\}$
5: $v_t \leftarrow \bigcup_{m \in \mathcal{V}_c} m(d_t)$ {**F4**: draft-time verifier execution}
6: **if** $|v_t| \geq \theta$ **then**
7: $\quad d'_t \leftarrow \mathcal{R}_2(d_t,\ v_t \mid \mathcal{I})$ {**F6**: Reasoner-2 correction}
8: $\quad$ **if** d'_t **is non-empty then** $y_t \leftarrow d'_t$ **else** $y_t \leftarrow d_t$
9: **else**
10: $\quad y_t \leftarrow d_t$
11: **end if**
12: **return** y_t

Correspondence to Eq. 1. Algorithm 1 computes the three per-context factors $P(I_c|c,\mathcal{I}), P(\mathcal{M}_c|I_c), P(s_c|\mathcal{M}_c, c)$ in sequence (lines 1, 2–11, 12–16). Algorithm 2 computes the per-task product $P(d_t|I_c, s_c, \tau(c), q_t) \cdot P(y_t|d_t, \mathcal{V}_c, \theta)$ as lines 3 and 5–11 respectively. The strategy-gating step (line 2 of Algorithm 2) realises the gating noted parenthetically in the where-clause of §3.2.

C Full Result Matrix and Statistical Tests

C.1 Full per-model per-domain pass rates

Table 4 reports the per-domain pass-rate matrix for all four methods × four base models × four domains. The Overall rows aggregate over the 1,899 tasks and match the corresponding column in the main results (Table 1).

Method	Domain	Kimi	GLM-5	DeepSeek	Qwen3
Vanilla	DKR	16.7	17.9	16.6	14.2
	RSA	13.4	15.2	13.4	11.0
	PTE	17.2	17.2	16.1	10.4
	EDS	12.1	10.1	11.6	10.6
	Overall (1899)	**15.4**	**16.1**	**15.0**	**11.9**
ReadAgent-P	DKR	14.3	13.3	13.9	12.8
	RSA	15.4	15.0	11.7	11.1
	PTE	17.8	13.4	16.6	12.7
	EDS	6.5	5.5	6.5	2.5
	Overall (1899)	**14.7**	**13.0**	**13.1**	**11.2**
Ctx2Skill	DKR	19.3	17.6	16.1	14.0
	RSA	13.3	13.3	13.4	9.9
	PTE	15.1	14.6	16.8	11.9
	EDS	10.1	14.1	11.6	10.6
	Overall (1899)	**15.5**	**15.2**	**15.0**	**11.9**
CCA (ours)	DKR	21.1	23.5	17.8	15.4
	RSA	21.2	19.8	16.8	9.7
	PTE	26.1	25.1	22.5	14.6
	EDS	11.6	8.0	9.0	4.5
	Overall (1899)	**21.4**	**21.2**	**17.7**	**12.4**

Table 4: Full per-model per-domain pass rates (%). Bold rows give Overall over 1,899 tasks.

C.2 Per-sub-category lift over Vanilla

Table 5 reports CCA's pass-rate lift over Vanilla (in percentage points, computed on the same model and sub-category) for every CL-bench sub-category and every base model. Sub-categories are sorted by mean lift across models (descending). Rule-dense sub-categories cluster at the top with consistently positive lifts across all four models; open-ended sub-categories at the bottom show consistent reversals. This is the empirical basis for the task-structural-density moderator in §5.3.

C.3 McNemar paired-significance test

For each base model we test whether CCA's Overall pass rate differs significantly from Vanilla's on the same 1,899 tasks using a McNemar test for paired binary outcomes. Let b = number of tasks where CCA passes and Vanilla fails, c = number of tasks where Vanilla passes and CCA fails; the test statistic is $z = (b - c)/\sqrt{b + c}$ and we report the two-sided p-value under the normal approximation.

The three larger models pass the $\alpha = 0.01$ significance bar; on Qwen3-Next-80B the lift is small ($+0.5$pp) and the test is consistent with the null. This pattern is the empirical basis for Moderator 2 (model capacity) in §5.3: a 3B-active reasoner cannot fully consume the structured IR plus injected code-execution results in addition to the original context, so the harness contributes little incremental signal on top of Vanilla.

D Case Studies: CCA-Only Wins, One per Domain

We select one task from each of CL-bench's four domains where CCA passes the rubric and *all three* baselines (Vanilla, ReadAgent-P, Ctx2Skill) fail, then trace the mechanism. All four cases are on Kimi K2.5; matching task IDs and full outputs are released with the supplementary material. For each case we report the cca_meta block produced by the Compiler so that the reader can see which IR fields and verifier modules did the work.

D.1 DKR / Management — three-tier escalation routing

Task. The model plays a *Human Expert Coordinator* for an Executive-Chef multi-agent system. Given a 9.5K-character conversation transcript involving a high-visibility private event with allergen risks, the rubric (13 criteria) requires correctly routing escalations under a three-tier priority system (CRITICAL = allergens / toxic ingredients; HIGH = cultural sensitivity / medical nutrition; MEDIUM = low confidence / rare cuisine) and matching each escalation to a specific expert type (food safety specialists, registered dietitians, cultural anthropologists, culinary safety instructors).

Baseline failures.

- **Vanilla** misclassified at least one safety-critical scenario, missing the CRITICAL tier requirement (rubrics 1–2).

Sub-category	Domain	n	Kimi	GLM-5	DeepSeek	Qwen3
Legal & Regulatory	RSA	92	+21.7	+20.7	**+27.2**	+20.7
Management	DKR	112	**+25.0**	+20.5	+10.7	+17.0
Legal Advisory	DKR	76	+11.8	+9.2	+5.3	+17.1
Programming Syntax	RSA	67	+14.9	+10.4	+13.4	−1.5
Operational Procedures	PTE	185	+12.4	+9.7	+11.4	+3.2
Workflow Orchestration	PTE	229	+8.7	+10.5	+6.6	+6.6
Experimental Data	EDS	45	+11.1	+6.7	+8.9	−4.4
Healthcare	DKR	105	+5.7	−1.0	+7.6	+7.6
Mathematical Formalism	RSA	69	+7.2	+7.2	+1.4	+2.9
Humanities	DKR	124	+4.0	+12.9	−0.8	−4.8
Technical Standards	RSA	201	+9.5	+3.5	+0.5	−4.0
Simulation Environment	EDS	59	−5.1	+1.7	+1.7	−5.1
Science	DKR	88	−6.8	−1.1	+2.3	−9.1
Lifestyle	DKR	57	−1.8	+3.5	−14.0	−7.0
Instructional Procedures	PTE	57	−1.8	−8.8	−10.5	−1.8
Observational Data	EDS	95	−3.2	−8.4	−10.5	−7.4
Game Mechanics	RSA	137	−7.3	−8.8	−12.4	−13.9
Finance	DKR	101	−11.9	−8.9	−8.9	−13.9

Table 5: Pass-rate lift of CCA over Vanilla on each (sub-category, base model) cell, in percentage points. Sub-categories are sorted top-to-bottom by mean lift across the four models. The top block (above the horizontal rule) is the rule-dense regime where CCA's mean lift across models is positive (individual cells may still be slightly negative on the small-activation Qwen3 in a few rows); the bottom block is the open-ended regime where CCA underperforms Vanilla on most or all models. The maximum gain (bold) is RSA Legal & Regulatory on DeepSeek-V3.2 (+27.2pp); the maximum DKR gain is Management on Kimi K2.5 (+25.0pp). Mapping to figures/tables in the main text: this table is the empirical basis for §5.3 Moderator 1.

Base model	z	p
Kimi K2.5	6.00	< 0.01
GLM-5	4.89	< 0.01
DeepSeek-V3.2	3.01	< 0.01
Qwen3-Next-80B	0.38	0.70

Table 6: Paired McNemar test of CCA vs Vanilla on the full 1,899-task CL-bench set, per base model. The three larger-capacity models pass $\alpha = 0.01$ with highly significant lifts; Qwen3-Next-80B (3B active) is the one model-capacity-bounded case discussed in §5.3.

- **ReadAgent-P** did not give the 1-hour response target to all CRITICAL scenarios; the gist summary lost the priority-timing detail (rubric 2).
- **Ctx2Skill** routed all safety issues to "Food Safety Specialists" and never invoked the distinct "Culinary Safety Instructor" role required by rubric 6 — the skill library lacked the exact-term distinction.

What CCA did. Compiler classified strategy as `rule_engine`, extracted 24 rules (15 `codeable`), including the 3-tier escalation table as `must_do` rules and the 5 expert-type strings as `exact_terms`. Verifier ran on the Reasoner-1 draft and surfaced 2 violations; Reasoner-2 correction fired and produced the final accepted answer (`cca_meta.corrected = True`).

D.2 RSA / Programming Syntax — directed multigraph visualization

Task. Generate a Python program (using `networkx + matplotlib.pyplot + flowpaths`) that illustrates a directed multigraph with specific node labels (`s, a1, a2, b1, b2, b3, c1, c2, t`). The rubric (10 criteria) specifies: pre-established node labels, two separate graphs (before/after edge split), diagonal layout per layer, no edge–node overlap, brief comments, no surrounding prose.

Baseline failures.

- **Vanilla** silently rewrote the node labels to `s1, s2, t1, t2`, violating rubric 2's verbatim node-set requirement.
- **ReadAgent-P** produced undirected curves with no arrow artists, violating the directional-edge requirement.
- **Ctx2Skill** used long natural-language comments inside the code ("Create directed multigraph with flow values", "Nodes in diagonal layers (square layout)"), violating rubric 10 (brief / single-word comments).

What CCA did. Compiler classified strategy as `rule_engine`, extracted 17 rules (14 `codeable`) including the node labels as `exact_terms`. Verifier flagged 3 violations on the Reasoner-1 draft (one of which was likely the comment-length rule), Reasoner-2 corrected (`cca_meta.corrected = True`).

D.3 PTE / Workflow Orchestration — SEV2 incident with user override

Task. The model is *Nebula Orchestrator*, a cloud-provider incident-management automation brain. Mid-incident the user asks the model to switch from `Mode: TOOL_SEQUENCE` to `Mode: CHAT` for one turn. The internal manual mandates running the SEV2 diagnostic workflow regardless of the user override. The rubric (13 criteria) requires: first line `Mode: TOOL_SEQUENCE`, a TOOL_PLAN, four specific tool calls (`diagnose_service`, `query_metrics`, `fetch_logs`, `post_status_update`) with exact arguments, an `append_audit_log` call with `event_type="USER_OVERRIDE_IGNORED"`, and a USER_MESSAGE explaining what was overridden.

Baseline failures.

- **Vanilla** began the response with `Mode: CLARIFICATION_NEEDED` (acceding to the user's override), violating rubric 1.
- **ReadAgent-P** produced a TOOL_SEQUENCE structure but omitted the `append_audit_log` call entirely — the gist summary lost the audit-logging mandate (rubric 8).
- **Ctx2Skill** also went into `Mode: CLARIFICATION_NEEDED`: the skill library encoded "defer to user instructions" as a general principle, overriding the context's domain-specific mandate.

What CCA did. Compiler classified strategy as `rule_engine with strict workflow enforcement for SEV2 mandatory diagnostics and audit logging despite user override requests` (the recommended_strategy field captures the conflict explicitly). Extracted 64 rules (29 `codeable`), including `must_do` rules for the mandated tool calls and a `must_not` rule against switching modes mid-incident. Verifier flagged 4 violations on the Reasoner-1 draft; Reasoner-2 corrected (`cca_meta.corrected = True`).

D.4 EDS / Experimental Data — proportionality constant with ASCII-only constraint

Task. The model plays *physicsLawAI*, an assistant in a parallel-dimension physics lab with unknown laws; it must analyse provided data and report a proportionality constant. The rubric (5 criteria) requires: a numeric value, units (if dimensional), uncertainty estimate, uncertainty in matching units, and **ASCII characters only**.

Baseline failures.

- **Vanilla** reported $1.27 \times 10^{-7}\ \mathrm{W\,m^{-2}\,K^{-5}} \pm 0.15 \times 10^{-7}$ — correct number and units, but the typeset multiplication sign "×" is a non-ASCII character, violating rubric 5.
- **ReadAgent-P** reported the value as 2.3 ± 0.15 with no units at all, violating rubrics 2 and 4.
- **Ctx2Skill** refused: "I cannot complete this task because the dataset does not contain repeated measurements…" — violating rubric 1 (no numerical value produced).

What CCA did. Compiler classified strategy as `data_analysis` (the only domain that triggered this strategy among our four cases), extracted 11 rules (5 `codeable`), and emitted the `data_analyzer` module. The module precomputed a power-law fit on the per-context data block; the result entered the Reasoner-1 prompt as a separate CODE EXECUTION RESULTS block, and the IR's ASCII-only formatting rule appeared in the checklist. Reasoner-1's draft used pure ASCII (1.18e-7 rather than $\times 10^{-7}$) from the start; the verifier still found 3 minor violations and Reasoner-2 corrected the final response (`cca_meta.corrected = True`).

D.5 Summary across the four cases

In every case the verifier flagged ≥ 2 violations and Reasoner-2 fired, illustrating that the correction loop is exercised at exactly the kinds of context-dense tasks the ablation in §5.1 identifies as benefiting most from CCA. The IR feature that "carries the day" differs across cases: `exact_terms` (DKR Management role names; RSA node labels), `must_do` / `must_not` rules (PTE incident-override resolution), `output_spec.formatting_rules` (EDS ASCII-only). The CodeGen `data_analyzer` module is the additional non-LLM compute path used only in the EDS case.

E Baseline Porting Notes

To ensure that our baseline comparisons in §4.5 reflect each method's intended design and not implementation artifacts, we port both baselines as faithfully as possible, deviating only where CL-bench's response format strictly requires it. This appendix documents those deviations.

E.1 ReadAgent-P

Algorithm. We follow the four-step pipeline of Lee et al. (2024): (i) pagination of the raw context into episodes (target ∼600 words per page, ≥280-word start threshold); (ii) per-page LLM-driven gist compression (parallel across pages); (iii) parallel lookup that, given the query and all gists, returns up to 5 page indices to re-fetch; (iv) answer generation conditioned on (gists ∪ re-fetched pages ∪ query). The pagination and gisting stages are amortised per context (a context shared across $|\mathcal{T}_c|$ tasks pays them only once).

Prompts. We use the prompts from the official Colab notebook verbatim for pagination, gist, and lookup. The only modification is to the *answer* prompt: ReadAgent's original was specialised to QuALITY's multiple-choice format ("which of A/B/C/D is correct?"). CL-bench tasks expect free-form natural responses, so we replaced the multiple-choice scaffolding with the same generic free-form instruction used for our Vanilla baseline. All other ReadAgent prompts are unchanged.

Configuration. Look-up cap of 5 pages (the value used in the original ReadAgent-P paper for QuALITY). No grid-search over this hyperparameter on CL-bench.

E.2 Ctx2Skill

Algorithm. We follow the self-play loop of Si et al. (2026): a Challenger synthesises probing tasks and rubrics from the context, a Reasoner solves them with the current skill set, a Judge issues per-task verdicts, and Proposer / Generator agents transform failures into skill-library updates. At inference time, the accumulated skill library is retrieved by similarity and injected into the prompt alongside the original context.

Prompts. All five agent prompts (Challenger, Reasoner, Judge, Proposer, Generator) are taken verbatim from the official Ctx2Skill codebase; we did not modify them. The only change is the model backbone: the original implementation runs against a single OpenAI-style endpoint, and we adapted the LLM-call layer to AWS Bedrock so the same four base models (Kimi K2.5, GLM-5, DeepSeek-V3.2, Qwen3-Next-80B) are used both for self-play skill generation *and* for inference, exactly matching our Vanilla and CCA conditions.

Configuration. Default Ctx2Skill self-play config: **5 iterations** × **5 tasks per iteration** × **500 unique contexts**. The resulting skill library is shared across all 1,899 inference tasks for each (base model) cell.

E.3 Why the porting is fair

Both ports use the original prompts verbatim wherever possible; the only modifications are (i) ReadAgent's answer prompt format-adapter (mechanically required to produce free-form rather than multi-choice output) and (ii) Ctx2Skill's Bedrock LLM-call layer (a back-end swap, with the same base models). No method-specific hyperparameters were retuned for CL-bench, so the comparison in §4.5 reflects each method's published design rather than any benchmark-fitting.

F Output Truncation Audit Details

Rubric-based grading is unforgiving of truncated outputs: a cut-off sentence often misses a required element, and the grader cannot distinguish "the model didn't say it" from "the model would have said it but ran out of tokens." We performed a two-pass audit of all 16 (method×model) inference files used to populate the main results table.

Pass 1: Detection. For each inference file, we estimated each task's output length in tokens (chars/3.5 approximation) and flagged tasks whose estimated length exceeded 98% of the per-call output cap. This produced **176 truncation suspects** across the Vanilla, ReadAgent-P, and Ctx2Skill methods; the bulk came from ReadAgent's answer-stage cap (a setup error we corrected in the audit), with a smaller number contributed by CCA's Reasoner cap.

Pass 2: Re-run. All 176 suspects were re-run with the per-call cap raised to each model's native maximum (8,192 for kimi/glm5/qwen3; 16,384 for deepseek-v3.2), and results were merged back into the original inference files by `task_id`. We additionally identified 60 CCA-base special cases (47 output-length suspects, 1 compiler-failed context,

and 12 empty-output cases concentrated on Qwen3-Next-80B); the compiler-failed context spanned 3 downstream tasks, bringing the task-level re-run count to 62. All were re-run at the raised cap.

Pass 3: Post-audit verification. Across all 16 final inference files (4 methods $\times$ 4 models $\times$ 1,899 tasks $=$ 30,384 task instances), the maximum observed output length after re-runs is 14,594 tokens (mean 5,611), well under the raised ceiling, and no task in any file lies within 10% of the cap. We therefore certify that the pass rates reported in §4.5 are not biased by output truncation.

Suspect counts per (method, model). Vanilla: kimi=18, glm5=20, qwen3=4, deepseek=0 (total 42). ReadAgent-P: kimi=69, qwen3=22, deepseek=16, glm5=9 (total 116). Ctx2Skill: kimi=14, qwen3=4, glm5=0, deepseek=0 (total 18). Grand total: 176.

G Extended Ablation Details

This appendix expands the component-level ablation of §5.1 along four axes: a visualisation of the additive build-up from Vanilla to Full CCA (Figure 3), per-domain pass rates of the ablation variants, correction-loop fire rate per variant, and token consumption per variant. All numbers are on Kimi K2.5 over the full 1,899-task set.

G.1 Per-domain pass rates of the ablation variants

Table 7 breaks down each variant's Overall pass rate by the four CL-bench domains. The pattern is consistent with the task-density moderator: per-domain swings are largest on the rule-dense RSA / PTE columns, where adding F2 (V6$\rightarrow$V3) flips RSA from 16.6% to 18.0% and PTE from 20.6% to 24.2%; the open-ended EDS column is comparatively flat across V6/V3/V2/V4.

G.2 Correction-loop fire rate per variant

Table 8 tabulates how often the violation count crossed the $\theta = 2$ threshold for each variant, and how often Reasoner-2 actually corrected the draft when triggered.

G.3 Token consumption per variant

Table 9 reports average per-task token consumption decomposed into Reasoner-1 (always one call) and Reasoner-2 (zero or one call, dependent on $|v_t| \geq \theta$). Tokens for the per-context Compiler

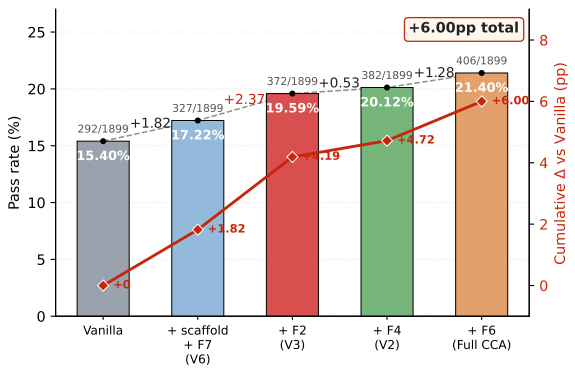

Figure 3: **Cumulative buildup from Vanilla to Full CCA on Kimi K2.5 (full 1,899 tasks).** The bar+line combo packs four quantities into one view, one per column of Table 2: (i) bar height = *pass rate* at each configuration; (ii) bar-top label = *absolute pass count* out of 1,899; (iii) annotations between bars = *marginal* Δpp from adding one feature on top of the previous configuration (F2 highlighted in red as the largest single jump, +2.37pp); (iv) red diamond line on the right axis = *cumulative* Δ *vs Vanilla*, sweeping from 0 to +6.00pp as features are added. Build-up path: Vanilla $\rightarrow$ V6 $\rightarrow$ V3 $\rightarrow$ V2 $\rightarrow$ Full; the four marginal Δpp annotations sum exactly to +6.00pp.

and CodeGen stages are not included here (they are amortised over many tasks).

G.4 F2 amplifies the F4/F6 correction loop

The F2 IR-as-checklist injection interacts positively with the F4/F6 verifier-and-correction loop. With F2 on, F4+F6 jointly contribute Full $-$ V3 $= 21.40 - 19.59 = +1.81$pp; with F2 off, they contribute only V4 $-$ V6 $= 18.33 - 17.22 = +1.11$pp. The +0.70pp gap is the positive interaction: the IR checklist pushes the Reasoner-1 draft closer to satisfying the checked rules, so when the correction loop fires it has a tighter, more actionable violation list and is more likely to produce a verified improvement. This interaction is also why F2's measured contribution is larger when removed from Full CCA (-3.07pp drop) than when added on top of scaffold-only (+2.37pp marginal).

G.5 Reading the three tables together

The ablation variants exhibit a clean cost-benefit decomposition: F2 adds $\sim$2K tokens to R-1 but contributes +2.37pp to the overall pass rate ($\sim$1.10pp per K tokens). F6 adds $\sim$10–11K tokens (when it fires) but contributes only +1.28pp ($\sim$0.11pp per K tokens) — it is roughly an order of magnitude less token-efficient than F2 per percentage point of lift. The verifier execution itself (F4) is non-LLM

Variant	DKR	RSA	PTE	EDS	Overall
Vanilla	16.7	13.4	17.2	12.1	15.40
V6 (no F1; cascade)	18.6	16.6	20.6	6.5	17.22
V3 (V6 + F2)	20.1	18.0	24.2	11.6	19.59
V2 (V3 + F4)	20.5	20.5	22.9	11.1	20.12
V4 (Full − F2)	17.9	18.9	21.2	11.1	18.33
Full CCA	**21.1**	**21.2**	**26.1**	**11.6**	**21.40**

Table 7: Per-domain pass rate (%) of each ablation variant on Kimi K2.5, bracketed by Vanilla (top) and Full CCA (bottom) for reference. The largest single-domain swing is RSA when F2 is added on top of V6 ($16.6 \rightarrow 18.0\%$, $+1.4$pp on RSA alone) and PTE when F2 is added ($20.6 \rightarrow 24.2\%$, $+3.6$pp). The EDS column is comparatively flat across V6/V3/V2/V4/Full ($\sim$6–12%), and Full CCA's EDS (11.6) sits below Vanilla's (12.1), supporting the §5.3 reading that CCA's components offer little or negative lift on open-ended tasks.

Variant	F4	F6	Fire %
V6 (no F1 cascade)	—	—	0.0
V3 (V6 + F2)	—	—	0.0
V2 (V3 + F4)	✓	—	63.7
V4 (Full − F2)	✓	✓	66.7
Full CCA	✓	✓	61.7

Table 8: Fraction of the 1,899 tasks on which $|v_t| \geq \theta = 2$ (verifier-fire rate). Columns F4 and F6 indicate whether draft-time verifier execution and Reasoner-2 correction are active in each variant. V6 / V3 have no F4 so no violation list is computed and the threshold is never crossed. V2 has F4 active but F6 disabled, so the verifier runs but R-2 never fires — this is why V2 only adds $+0.53$pp over V3 (Table 2): violations are logged but not acted on. V4 fires at the highest rate (66.7%); Full CCA fires slightly less (61.7%) because F2 already pushes the Reasoner-1 draft toward rule satisfaction, leaving fewer post-draft violations.

and cost-free at inference, contributing $+0.53$pp purely through giving F6 something to work on (V2 vs V3 in Table 8 confirms F4 alone, with F6 disabled, fires 63.7% of the time but the violations are never acted upon).

H Cost Analysis: Offline Prep + Online Inference

This appendix re-examines CCA's token consumption through the lens of its *architectural two-phase structure*: an ***offline prep*** phase that runs once per context and an ***online infer*** phase that runs once per task. This split is not a deployment trick—it is a property of Equation 1, where the per-context factor $P(I_c \mid c, \mathcal{I}) \cdot P(\mathcal{M}_c \mid I_c) \cdot P(s_c \mid \mathcal{M}_c, c)$ runs once per c and the per-task factor runs once per $t \in \mathcal{T}_c$. We verified by code inspection (the main loop in our reference implementation groups tasks by `context_id` and runs Compiler+CodeGen+`data_analyzer` exactly once per group) that the token totals reported in §4.5 already reflect this amortisation.

Variant	$\bar{T}_{\mathcal{R}_1}$	$\bar{T}_{\mathcal{R}_2}$	$\bar{T}_{\text{total}}$
V6 (no F1 cascade)	14,190	0	14,190
V3 (V6 + F2)	16,353	0	16,353
V2 (V3 + F4)	16,384	0	16,384
V4 (Full − F2)	14,206	9,927	24,133
Full CCA	16,468	11,192	27,660

Table 9: Average per-task Reasoner-1 ($\bar{T}_{\mathcal{R}_1}$), Reasoner-2 ($\bar{T}_{\mathcal{R}_2}$), and total ($\bar{T}_{\text{total}}$) token counts (input + output) per ablation variant. Two patterns: (i) F2 adds $\sim$2,100 tokens to the Reasoner-1 input (14,190 $\rightarrow$ 16,353 V6$\rightarrow$V3) because the IR checklist is rendered into the prompt; (ii) enabling F6 with the correction prompt costs an additional $\sim$10–11K tokens per task on average (V4 vs V3; Full vs V2), reflecting the fact that R-2 fires on $\sim 2/3$ of tasks and re-encodes the prior conversation when it does.

H.1 All four methods as offline-prep + online-infer architectures

Vanilla and ReadAgent-P also admit a pre/infer split (ReadAgent's pagination+gist is per-context offline); Ctx2Skill is the closest architectural match to CCA, with a heavy self-play offline phase plus a per-task online inference. Table 10 reports each method's split on Kimi K2.5.

Method	Offline	Online	Lift
Vanilla	0	11.9K	—
ReadAgent-P	$\sim$3K	$\sim$11K	-0.7pp
Ctx2Skill	$\sim$178K	$\sim$15K	$+0.1$pp
CCA (Full)	**6.4K**	**27.7K**	**+6.00pp**

Table 10: Per-task tokens decomposed into offline (per-context, amortised) and online (per-task) phases on Kimi K2.5. Ctx2Skill's offline cost is $\sim 30\times$ CCA's, yet returns negligible lift; CCA pays a heavier online cost but converts it into a $+6.00$pp lift. As tasks-per-context $N \rightarrow \infty$ in deployment, CCA's marginal per-task cost converges to the online column (27.7K); Ctx2Skill's offline phase must be re-paid whenever the context pool changes.

Positioning vs all three baselines. On the Kimi K2.5 cost axis, CCA spends $\sim 3\times$ Vanilla's and $\sim 2.5\times$ ReadAgent-P's per-task tokens—a deliberate trade for the only large positive lift among the three methods—while using just $\sim$18% of

Ctx2Skill's tokens to deliver a substantially larger lift. The two off-frontier baselines (ReadAgent-P and Ctx2Skill) each pay a non-trivial token cost over Vanilla but return near-zero lift, so only CCA actually converts extra tokens into measurable rubric pass-rate gains; this is the cost-quality picture visualised in Figure 2.

Direct comparison with Ctx2Skill. Ctx2Skill is the closest architectural analogue to CCA: both methods do per-context work offline and use cached artifacts at inference. The differences are stark:

- **Offline phase**: Ctx2Skill runs a $5 \times 5 \times 500$ self-play loop (Challenger / Reasoner / Judge / Proposer / Generator), spending $\sim$178K tokens per task on amortised average. CCA runs one Compiler call and up to three CodeGen calls, spending $\sim$6.4K — roughly 30$\times$ cheaper.
- **Online lift**: Ctx2Skill's skill library encodes general procedural guidance and contributes $+0.1$pp (statistically indistinguishable from Vanilla); CCA's IR + executable verifier code is tightly coupled to rubric structure and contributes $+6.00$pp.

The combined effect is an order-of-magnitude better *offline-token-per-pp-lift* ratio for CCA. The architectural lesson is that what the offline phase produces matters more than how much it spends: Ctx2Skill produces natural-language skills (lossy abstraction); CCA produces a typed IR plus executable checks (lossless for what the rubric tests).

H.2 Stage-level decomposition of CCA's 34K per-task budget

Table 11 attributes each token to a specific stage. The first two rows are amortised across the $\sim$3.5 tasks per context in CL-bench; the last two are per-task.

Where the online cost goes. Compared to Vanilla's 11.9K per task, CCA's Reasoner-1 adds $\sim$4.5K extra input tokens (the IR checklist for F2, plus the cached code-execution result s_c for F5, on top of the same head/tail-truncated original context). The Reasoner-2 stage is a second LLM call that fires on $\sim$62% of tasks, contributing an average of 11.2K tokens. Summing: $4.5\text{K} + 11.2\text{K} = 15.7\text{K}$ extra online over Vanilla.

Stage	Tokens/task	Phase
Compiler (amortised)	4,035	offline
CodeGen (amortised)	2,332	offline
Reasoner-1 (per-task)	16,468	online
Reasoner-2 (per-task)	11,192	online
Total	**34,027**	—

Table 11: Token decomposition of CCA (Full) on Kimi K2.5. Compiler and CodeGen are per-context (amortised over the $\sim$3.5 tasks per context in CL-bench); Reasoner-1 runs once per task, Reasoner-2 fires on $\sim$62% of tasks (the average 11,192 tokens already reflects that fire rate). The offline subtotal (6.4K) is the amortisation budget that shrinks with deployment scale; the online subtotal (27.7K) is the marginal per-task cost.

H.3 CCA ablation variants as cost-quality operating points

Each ablation variant in Table 2 is a real, deployable configuration. Table 12 reports their token cost on Kimi K2.5 alongside their lift, sorted by total token consumption.

Config.	Tok/task	Lift	Δtok
Vanilla	11.9K	—	—
CCA-V6	14.2K	+1.82	+2.3K
CCA-V3	20.4K	+4.19	+8.5K
CCA-V2	**22.7K**	**+4.72**	**+10.8K**
CCA-V4	30.5K	+2.93	+18.6K
CCA-Full	34.0K	+6.00	+22.1K

Table 12: Cost–quality operating points for CCA on Kimi K2.5; variant definitions are in Table 2. Δtok is the marginal cost over Vanilla. Token-efficiency (pp lift per thousand extra tokens): V6 $= 0.79$, V3 $= 0.49$, V2 $= 0.44$, V4 $= 0.16$, Full $= 0.27$. **CCA-V2** (Full minus Reasoner-2; verifier execution still active) is the cost-conscious endpoint: it recovers 4.72 of the 6.00pp lift ($\sim$79%) while saving $\sim$51% of Full CCA's marginal tokens over Vanilla. CCA-Full is the max-quality endpoint, paying $+22.1$K tokens for the full $+6.00$pp.

Reading the dial. The cheapest non-trivial configuration is CCA-V6 (just the prompt scaffold and F7 truncation), at $+1.82$pp for an extra 2.3K tokens (0.79 pp/Kt). Adding the IR-as-checklist injection (V6 $\rightarrow$ V3) more than doubles the lift at modest extra cost. Activating draft-time verifier execution (V3 $\rightarrow$ V2) adds another $+0.53$pp at $+2.3$K tokens (the CodeGen modules emitted but executed cheaply at inference). Activating Reasoner-2 correction (V2 $\rightarrow$ Full) is by far the most expensive single step, adding $+1.28$pp at $+11.3$K tokens (0.11 pp/Kt) — the lowest pp/Kt of any single com-

ponent, because R-2 fires on ∼62% of tasks regardless of whether each firing turns into a verified improvement.

When is R-2 worth it? The R-2 correction loop is the natural cost-quality knob. Disabling it (V2) saves 11.3K tokens per task (∼51% of Full CCA's marginal cost over Vanilla) at the cost of 1.28pp (21% of total lift). In a deployment that already cares about token budget — e.g., user-facing latency-sensitive serving — V2 is a clean choice; in offline or batch settings where the lift matters more than the per-query cost, Full CCA is the right endpoint.

H.4 Could CCA use the raw context only once?

By construction the Compiler reads the raw context once per context and produces an IR that summarises every constraint downstream stages need to honour. In principle, the Reasoner stages could then operate on the IR alone, dropping the per-task input by the ∼5–10K tokens currently spent on the truncated original-context block (F7). In practice we retain the truncated context in Reasoner-1's prompt because CL-bench rubrics frequently require verbatim reproduction of context strings, and the original surrounding text is the most reliable ground for verbatim grounding. We did not isolate this design choice as a separate ablation variant; CCA-V3 (Compiler on, IR injection, no verifier, no correction) is the closest existing variant and reaches +4.19pp at 20.4K tokens/task — already ∼14K below CCA-Full at ∼70% of the lift.

H.5 Practical recommendation

- **Latency-/cost-sensitive serving** (e.g., interactive applications, large query volumes): CCA-V2. +4.72pp at 22.7K tokens/task.
- **Maximum-quality batch processing** (e.g., compliance audits, legal review, document scoring): CCA-Full. +6.00pp at 34.0K tokens/task.
- **Hybrid deployment**: both variants share the same offline artifacts (IR, CodeGen modules, s_c), so a single per-context cache supports both V2 and Full inference paths — the system can route high-stakes queries to Full and bulk queries to V2 with no extra offline work.

I Temperature Robustness

The main results in §4.5 fix inference temperature at 0.0 across all methods (§4). To verify that CCA's pass-rate lift is not an artefact of greedy decoding, we re-ran the full CCA pipeline on Kimi K2.5 over the full 1,899-task CL-bench set at temperature 1.0, holding all other settings constant (same prompts, same audit-corrected output caps, same GPT-5.1 judge with default sampling).

Config.	Pass	Acc.	Δpp
Vanilla, temp= 0	292/1899	15.40%	—
Full CCA, temp= 0	**406/1899**	**21.40%**	**+6.00**
Full CCA, temp= 1	356/1899	18.75%	+3.35

Table 13: Pass rate of Full CCA on Kimi K2.5 over the full 1,899-task CL-bench set at temperature 0 (main result, bold) and 1 (robustness re-run), alongside the temp= 0 Vanilla baseline. Column "Δpp" reports lift over Vanilla (temp= 0). CCA at temp= 1 loses 2.65pp of absolute pass rate vs temp= 0 (sampling noise) but still beats Vanilla at temp= 0 by +3.35pp.

Reading the table. CCA-v10 at temp= 1 still outperforms Vanilla (at temp= 0) by +3.35pp, so the harness benefit is *not* an artefact of greedy decoding. Temp= 0 yields a moderately stronger ceiling (+2.65pp over temp= 1 on the same Full CCA pipeline), consistent with rubric-based grading rewarding lower-variance outputs: under the strict "all 5–20 criteria must pass" rule (§4), any per-task sampling noise that flips a single criterion costs the whole task. This same noise-floor effect is part of why our temp= 0 Vanilla numbers (15.4%–16.1% across the three larger models) sit modestly below the 23.7% best reported in the original CL-bench paper (Dou et al., 2026) under default sampling.

Why this matters for the main result. The +6.00pp Full-vs-Vanilla gap on Kimi K2.5 (§4.5, Table 1) is measured under matched temp= 0 conditions, so it isolates CCA's design contribution from any sampling-variance differences. The temp= 1 run shows that even when the comparison is asymmetric—CCA pays a sampling-noise tax while Vanilla keeps deterministic decoding—CCA still wins by over +3pp, a conservative lower bound on CCA's structural advantage.

Inference and grading provenance. The temp= 1 re-run was executed on 2026-05-25 with 32 parallel inference chunks and graded the same day by GPT-5.1 (40-worker parallel pool) using the offi-

cial CL-bench script, identical to the main-results grading pipeline (§4).