Designing a Domain-Specific Language for LLM Agent Workflows: Review-Equals-Runtime Planning with NormCode

Anonymous ACL submission

Abstract

LLM agent workflows in production are assembled by *agent engineers* but *reviewed, operated, and corrected* by domain experts who do not read code. Current frameworks (LangChain, LangGraph, AutoGen, DSPy, agent harnesses) leave the *plan* implicit, so the artifact a reviewer approves is not what the runtime executes and a failure at step k costs $O(k)$ to localise. We call this *review-runtime drift*. We present **NormCode**, a semi-formal planning language whose methodology factors every plan into *value concepts* (named results) and *functional concepts* (operations that derive them). The functional concepts come in a small atomic set: *semantic* **Imperatives** `::(...)` and **Judgements** `<{...}>` that invoke an LLM under a *norm* fixing the execution paradigm (prompted call, Python tool, or recruited capability), and *syntactic* helpers (`$/&/@/*`) that reshape tensors deterministically without invoking an LLM. The norm is the natural composition point for existing techniques: one per semantic node, with the cross-node binding load borne by the language and its runtime. A draft (`.ncds`) compiles deterministically to two projections of one formal plan: a plain-prose narrative (`.ncn`) for review, and a runtime artifact for the orchestrator. The scope rule — inputs at any step are exactly the concepts declared in its enclosing indented block — yields three structural capacities: **P1** per-node addressability, **P2** pre-execution reviewability, **P3** scope-bounded revisability. A multi-role **Canvas** surface (Designer / Reviewer / Operator / Reviser) exposes the same compiled artifact so the roles can engage without requiring the original agent engineer at each step. A constructive demonstration on IFEval (Zhou et al., 2023) wires LLM-synthesised per-instruction Python checkers as deterministic Judgements inside a draft–verify–revise loop. On a 209-prompt hard subset (`qwen-turbo-latest`, T=0), *NormCode preserves competitive constraint-following while producing audit/revision artifacts unavailable to the baselines*: prompt-strict 0.801 vs 0.807 for the constraint-aware-prompt baseline (matched-compute self-refine degrades by 6.4 pp), +21 pp over matched-compute self-refine on the hardest constraint family, and three structural artifacts — a reviewable narrative, bounded-input snapshots, and scope-bounded re-execution after mid-run edits — the baselines cannot produce.

1 Introduction

When a domain expert approves an LLM agent workflow before deployment, they are not approving what executes: they approve the code, but what runs is the code *together with* whatever conversation history, shared Python state, or templated references accumulate during the run. We call this gap *review-runtime drift*. Three consequences follow: pre-execution review is impossible because what runs is more than the code; failure localization at step k is $O(k)$ because diagnosing it requires reconstructing accumulated context across steps $1, \ldots, k-1$; and correction means restart because a checkpoint at k embeds an implicit dependency on context not encoded by the checkpoint. This is structural, not a missing feature — it follows from how LangChain (Chase, 2022), LangGraph (LangChain Inc., 2024b), AutoGen (Wu et al., 2023), PromptFlow (Microsoft, 2023), and their visual cousins pass information between steps.

Multi-step LLM agent workflows are nevertheless the dominant architecture for production AI. What ships is not a model but a *system* — foundation model, prompts, retrieval, tools, workflow scaffold, memory, harness — assembled by *agent engineers* composing harnesses (LangGraph, AutoGen), declarative DSLs (DSPy), and skill libraries (Claude Skills, OpenAI Assistants). But these systems are *used* by other roles — a domain expert verifying before deployment, an operator debugging at 3 AM, a user correcting an intermediate result without re-running the chain — each with

different expertise and access, typically none of them the original agent engineer. **The missing piece is a plan artifact** that exists independently of any run and that the orchestrator executes exactly as written.

Our claim and the central technical move. The existing post-training and agent stack (prompts, RAG, RLHF (Ouyang et al., 2022), Constitutional AI (Bai et al., 2022), fine-tuning, tool scaffolds, workflow frameworks, agent harnesses, DSLs) supplies real binding *at* a single step but not *across* steps at the granularity of an artifact a non-developer reviewer can recognise and amend. We present **NormCode**, a semi-formal planning language designed against that cleavage. Its methodology factors every plan into *value concepts* (named results, `<-`) and *functional concepts* (operations that derive them, `<=`). The functional concepts decompose into two kinds: *semantic* inferences — LLM-invoking **Imperatives** `::(action)` and **Judgements** `<{check}>`, each parametrised by a *norm* that fixes the execution paradigm (`composition`, `python_script`, or a recruited capability: a fine-tuned head, RAG, an agent harness, a recursive sub-plan) — and *syntactic* helpers (`$` assigning, `&` grouping, `@` timing, `*` looping) that reshape tensors deterministically without invoking an LLM. The norm is the composition point: one technique per semantic node, with cross-node binding borne by the runtime. From a single draft (`.ncds`) the compiler emits two deterministic projections of one formal plan — a plain-prose narrative (`.ncn`) for the Reviewer and a runtime artifact (`concept_repo.json + inference_repo.json`) for the orchestrator — so by construction *what is reviewed is what runs*.

The structural mechanism is the **scope rule**: an inference's inputs are exactly the concepts declared in its enclosing indented block; the compiler verifies it statically, the orchestrator passes only those declared tensors, with no implicit sharing of accumulated context. Three structural properties follow (§5): **(P1)** per-node addressability, **(P2)** pre-execution reviewability, **(P3)** scope-bounded revisability — *design consequences* we argue, not measure. The same compiled plan is exposed by **NormCode Canvas** as four interaction modes (Designer / Reviewer / Operator / Reviser; §4).

Constructive demonstration. We instantiate the language on IFEval (Zhou et al., 2023) with a single generic plan that wires LLM-synthesised per-instruction Python checkers as deterministic verification nodes inside a draft–verify–revise loop (K=3). Five conditions compare **B1** single-shot, **B2** constraint-aware prompting (matched-information), **B_COT** chain-of-thought single-shot, **B3** self-refine (Madaan et al., 2023) at matched compute, and **NC** the NormCode plan. The empirical reading is: *NormCode preserves competitive constraint-following while producing audit/revision artifacts unavailable to the baselines*. Concretely, on a 209-prompt hard subset (`qwen-turbo-latest`, T=0), NC matches B2 on aggregate prompt-strict accuracy (0.801 vs 0.807) while B3 *degrades* below B1 by 6.4 pp, NC obtains +21 pp over B3 on the hardest constraint family, and each run produces three artifacts exhibiting P1–P3 (reviewable `.ncn`, bounded-input snapshots, scope-bounded re-execution after a mid-run constraint edit) at no extra LLM cost (§6).

Contributions. (i) the language with its two-level methodology and review-equals-runtime compilation (§3); (ii) the structural account that P1–P3 follow from one scope rule, not three engineering choices (§5); (iii) a multi-role Canvas surface over the same compiled artifact (§4); (iv) the IFEval demonstration with matched-information and matched-compute baselines (§6).

2 Multi-Role Collaboration in LLM Agent Workflows

Four roles, four tasks. A production LLM agent workflow — a code-review assistant, a financial-report analyser, a clinical-pathway recommender — is touched by at least four roles after it leaves the developer's machine. A **Designer** writes the workflow and needs an artifact that admits revision without re-deriving the codebase. A **Reviewer** (domain expert, compliance officer, product owner) must confirm the workflow's reasoning structure matches the regulated process before deployment, reading the workflow itself rather than its description. An **Operator** runs the workflow daily and must answer *what did step k receive as input?* when a failure occurs. A **Reviser** spots an incorrect intermediate output during or after a run and must correct it at that step and have only downstream steps re-execute.

Why each role is blocked. The drift mechanism (§1) blocks each role differently: the Reviewer cannot approve before deployment because what they

read is not what runs; the Operator cannot read "what step k received" without reconstructing $O(k)$ accumulated context; the Reviser cannot override and resume because dependencies are implicit, not declared. These are structural consequences of implicit state-sharing as the inter-step communication mechanism, not bugs in any individual framework. We define a *plan-artifact medium* for multi-step LLM workflows as a language satisfying **pre-execution determinacy** (running the plan does not change it), **review-equals-runtime** (the Reviewer artifact and the orchestrator artifact are projections of the same formal plan), and **step-local addressing** (each step's inputs are exactly the concepts declared in its enclosing scope). §3 argues NormCode satisfies all three; §7 surveys related approaches.

3 NormCode: A Semi-Formal Language for AI Plans

NormCode's design rests on one methodological commitment: every step in an AI plan is either *semantic* (its output is created by an LLM call) or *syntactic* (its output is a deterministic reshaping of existing tensors), and the two are syntactically distinct from the moment the Designer writes the draft. This separation makes the LLM-call cost structure visible at authoring time, makes deterministic data flow auditable, and is what enables one source to compile into two projections without drift. This section presents the essentials needed for §4–§6; the fuller specification is in Appendix A.

3.1 Three Markers, Indentation, and the Scope Rule

The authoring format (`.ncds`) uses three concept markers: `<-` declares a **Value Concept** (a named value or intermediate result), `<=` declares a **Functional Concept** (an operation that derives its parent from declared inputs), and `<*` declares a **Context Concept** (in-loop state for the current iteration element). Indentation defines scope.

```
<- summary
    <= summarize the findings
    <- report
        <= read the uploaded file
        <- source_doc
    <- style_guide
```

The orchestrator's execution model is *top-to-bottom by flow index, inside-out by dependency*, realised by a single waitlist sorted on flow index: an inference fires the moment its declared inputs have resolved, and an enclosing inference fires the moment its inner block has resolved. In the example above the inner `<= read the uploaded file` resolves first, producing `report`; `report` and `style_guide` then become the declared inputs of the outer summarisation, which fires to produce `summary`. The *scope rule* states: an inference's execution context is exactly its scope. The compiler verifies it statically; the orchestrator enforces it at runtime by passing *only* the tensors of scope-declared concepts to each dispatched inference — no closure, no inheritance, no implicit sharing of accumulated context.

Kind	Marker	Role
Semantic (LLM-invoking, parametrised by a norm*)*		
Imperative	`::(...)`	creates a Value via an action
Judgement	`<{...}>`	decides a Proposition via an evaluation
Syntactic (deterministic; tensor-algebraic; no LLM)		
Assigning	`$`	route or rename a tensor
Grouping	`&`	collect concepts into a dict / list
Timing	`@:'`, `@:!`	gate execution on a Proposition
Looping	`*.`	iterate with optional carried state

Table 1: The two-class inference catalogue. Semantic markers (`::(...)`, `<{...}>`) signal an LLM call at compile time; syntactic markers (`$`, `&`, `@`, `*`) signal a deterministic tensor operation. A Reviewer can read cost structure off the plan because the two classes are visually distinct in both `.ncds` and the rendered `.ncn`.

3.2 The Design Methodology: Semantic vs. Syntactic Inferences

The functional concept's marker tells the compiler — and the Reviewer — which kind of work each step performs. Table 1 catalogues the six kinds; two invoke an LLM and four do not.

The norm as the execution paradigm of a semantic inference. A semantic functional concept always carries a *norm* that fixes *how* the call is performed: `composition` (a prompted LLM call assembled from the declared inputs), `python_script` (a deterministic Python tool the agent dispatches when the operation has a closed-form implementation), `method_NormCode` (recursive dispatch into a nested sub-plan), or a recruited capability such as RAG, a fine-tuned head, a structured-output decoder, or an external agent harness. The norm is the joining point between the

Subject (the agent with tools and faculties) and the Object (the data); it is what makes a semantic node a *configurable* LLM call rather than an opaque prompt. The norm is also the location at which external techniques compose into NormCode: at the granularity of one step, the workflow frameworks, declarative DSLs, agent harnesses, RAG variants, and post-training interventions surveyed in §7 can each serve as a candidate norm at a single semantic node.

Value-concept types. Value Concepts carry one of three non-functional types: *Object* (a named entity, the "noun"), *Proposition* (a boolean state of affairs read out by a Judgement and consumed by a Timing operator), or *Relation* (a collection iterated by a Looping operator). The vocabulary is small by design: enough to declare the inputs and outputs of LLM-invoking work without committing to a programming-language type system.

3.3 Flow Indices and References

The Formalization phase assigns each concept a dot-separated flow index. The functional concept is at `.1` under its parent; value inputs are siblings at `.2`, `.3`, etc.: `summary` → `1`, `report` → `1.2`, `source_doc` → `1.2.2`, `style_guide` → `1.3`. Every checkpoint is stored and retrieved by `(run_id, flow_index)`; this stable address is what P1 is built on (§5). Concept data are stored as **References** — named-axis tensors whose entries may be primitive values, file pointers, or *perceptual signs* of the form `%{norm_type}id(signifier)` that identify large content without loading it into a prompt; the orchestrator transmutes the sign at the moment of the LLM call.

3.4 Two Projections, One Source

The compiler takes a `.ncds` draft, produces a single *formal plan* in the Formalization phase, and from that single source emits two **deterministic** projections (Figure 1) — a plain-prose narrative `.ncn` for the human Reviewer (rendered by rule; no LLM invoked) and a runtime artifact (`concept_repo.json + inference_repo.json`) for the orchestrator. The projections cannot drift, because they are the same plan in two surface forms.

This is the structural basis for *review-equals-runtime* (P2, §5). Compilation proceeds in four phases — **Derivation** (LLM-assisted: extracts concepts and dependencies), **Formalization** (rule-based: assigns flow indices, emits `.ncn`), **Post-Formalization** (annotates paradigm IDs and tool faculties), and **Activation** (emits runtime artifacts) — detailed in Appendix C. Figure 2 (§6.2) shows a complete `.ncn`; after learning the four tags (`OUTPUT`, `ACTION`, `VALUE`, `CONTEXT`) and the marker dichotomy of §3.2, a Reviewer with the relevant domain expertise can answer *what the workflow does, what each step depends on, and which steps invoke an LLM* from the narrative alone. This is a structural property of `.ncn`, not a claim that any reviewer can audit any plan (§ Limitations).

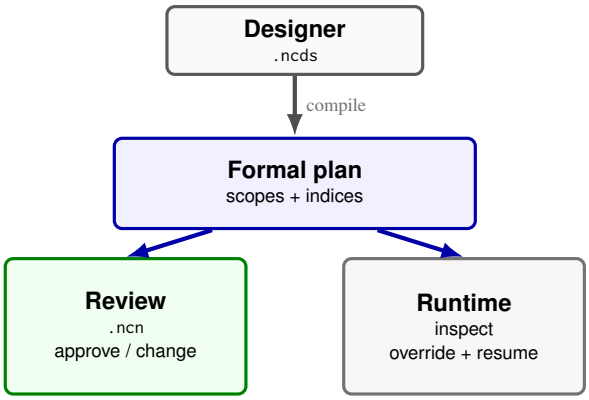

Figure 1: Four roles, four tasks, one artifact. Reviewers approve or request changes before execution; Operators inspect checkpoints and Revisers resume only scope-downstream work at runtime.

4 The Canvas: A Multi-Role Surface Over the Compiled Plan

Canvas is one deployed environment that exposes a single compiled plan as four interaction modes; the same plan artifact and execution state underlie all four. The point of this section is not the UI: it is to show that the four role-affordances are nothing more than the language's compiled artifact viewed at different times. UI specifics, architecture, and protocol details are in Appendix D.

Designer (authoring). Edits `.ncds`, runs the compiler, and inspects the resulting graph; semantic and syntactic nodes are rendered distinctly so the LLM-call cost structure is visible before any execution.

Reviewer (pre-execution review). Reads the `.ncn` narrative produced by Formalization, then approves or requests changes. No LLM call, no execution cost, no `.ncds` or `.ncd` syntax required. Because `.ncn` and the runtime artifact are projections of the same formal plan (§3.4), review-runtime drift is closed by construction. This is what P2 names.

Operator (live execution). The graph turns

live: each node's status streams via WebSocket; a Tool Call Monitor logs every LLM prompt, file access, and Python execution indexed by flow index. Breakpoints pause before a node runs; the Tensor Inspector opens the Concept Repository entry for any completed or paused node and shows its declared inputs by flow index. This is what P1 names.

Reviser (correct + resume). From any breakpoint or completed checkpoint the Reviser edits a tensor in the Inspector; on resume the orchestrator marks all transitively-downstream nodes stale (computed from the declared dependency graph), re-executes only those, and serves upstream nodes from cache. *Fork* is the read-only sibling: branch a new run from a checkpoint without modifying the original. This is what P3 names.

5 Three Structural Capacities from One Rule

P1–P3 are structural consequences of the scope rule (§3.1) and, for P2, the two-projection design (§3.4). They are *design properties* we argue, not measure; whether the language produces better outputs on any specific task is the separate question §6 addresses on IFEval.

P2 — Pre-execution reviewability (Reviewer). The Formalization phase operates on a fully-resolved formal plan and the `.ncn` renderer is rule-based, so what the Reviewer reads matches what executes — not a claim that approval guarantees correct behaviour, since semantic inferences remain probabilistic at every Imperative and Judgement node.

P1 — Per-node addressability (Operator). For an Operator diagnosing a failure at step k in a workflow of depth n, identifying the bounded inputs of step k is $O(1)$ in n: the scope rule fixes step k's inputs to exactly the concepts declared in its enclosing block, the compiler records that set, and the orchestrator writes every input tensor to the Concept Repository indexed by flow index — contrasted with the $O(k)$ conversation-replay cost in §2.

P3 — Scope-bounded revisability (Reviser). When a Reviser overrides the tensor at flow index f, the orchestrator re-executes exactly $\mathrm{Down}(f)$ in the declared dependency graph G and serves all other nodes from cache. By the scope rule, G captures every data dependency, so any $w \notin \mathrm{Down}(f)$ has identical bounded inputs before and after the override. In frameworks where the declared-argument graph is a strict subset of the actual dependency graph (via accumulated history or shared state), a conservative implementation must re-execute everything from f onwards — $O(n-f)$ per revision.

One mechanism, three properties. Removing the scope rule removes P1 and P3; removing the two-projection design removes P2. The properties are cheap together and expensive apart: they share an authoring cost (declaring scope) and a compiler cost (Formalization), and once paid, all three are present.

6 Evaluation: Expressivity and a Constructive Demonstration

P1–P3 are structural properties of the language (§5); on their own they do not answer two prior questions. **(Q1, expressivity)** Does the language describe production plans without escape hatches? **(Q2, performance)** Does it help on a representative compound-constraint task? §6.1 addresses Q1 with four deployed plans; §6.2–§6.4 address Q2 with a single constructive demonstration on IFEval whose execution also produces three artifacts that exhibit P1–P3 at no extra LLM cost.

6.1 Expressivity: Production Case Studies

Four plans deployed by the development team illustrate the language's range (`.ncds` sources in Appendix F): **PPT Generation** (∼30 inferences, ∼10–14 LLM calls/run; nested loop, six layouts, fourteen component types); **NC Compilations** (∼25, ∼12–20; the self-hosting case — a NormCode plan that runs the four-phase NormCode compiler); **Canvas Chat Assistant** (∼20, ∼4–8; request classification, scoped retrieval, frontend dispatch); **3D Model Report** (∼15, ∼6–10; multi-modal perceptual signs into a structured report). Across all four the same three markers and six functional kinds (§3.2) cover sequential pipelines, nested iteration, conditional routing, and multi-modal flows, and each plan's LLM-call count is decidable from the formal plan before any execution.

PPT Generation in detail. Takes a topic string and produces a `.pptx` deck plus an HTML preview through a nested loop: an outer Imperative synthesises an outline, a syntactic loop (`*.`) iterates over sections, and an inner loop iterates over slides per section — one Imperative drafts each

slide's body under norm `composition`, a second Imperative under norm `python_script` invokes a Python tool to compose the `.pptx` page. Six layouts and fourteen component types are typed References at the outer scope; the scope rule re-declares `{style_guide}` inside the inner block so every draft reads it from a known address, while *neighbouring* slides' drafts are out of scope unless explicitly declared — foreclosing implicit cross-slide drift. Of the ∼30 inferences, ∼10–14 are semantic and the rest syntactic; the split is readable off the markers before any execution, as for the other three plans (Appendix F). We claim sufficiency for these four production plans, not universal expressivity (§ Limitations).

6.2 IFEval: Task and Plan

IFEval (Zhou et al., 2023) is a public benchmark of 541 verifiable-instruction prompts shipped with a pre-extracted constraint structure: each prompt carries an `instruction_id_list` over a closed vocabulary of 25 instruction types (length, format, keywords, punctuation, combinations) and a corresponding `kwargs` list, with a deterministic Python checker per type. Scoring reports prompt-level and instruction-level compliance, each in strict and loose normalisation modes. We use IFEval because (i) the pre-extracted constraints let us wire one deterministic Judgement per constraint type into a NormCode plan without involving an LLM at the verification step; (ii) compound-constraint compliance is the regime where unstructured baselines are most prone to dropping one constraint between draft and revision — the regime in which the plan structure is most likely to register a measurable effect; (iii) the published checkers eliminate annotation disagreement, so the comparison concerns *whether* the structure helps, not scoring methodology.

Subset. The headline numbers are reported on a 209-prompt hard subset — all 57 three-constraint prompts and 152 of the 179 two-constraint prompts (the front of the stratified order). The 305 single-constraint prompts converge across conditions on this benchmark; the full 541-prompt run is reported in Appendix L as it completes.

Plan (`.ncds`). A single generic plan (Figure 2) is authored once and instantiated per prompt. It illustrates the language's two inference classes in their typical division of labour: two semantic **Imperatives** (`::(...)`) generate `[verifiers]` and `{initial draft}` (one LLM call each), a syntactic **Looping** (`*.`) iterates over the draft chain dispatching deterministic checks (norm: `python_script`) and a semantic **Judgement** (`<{...}>`) that decides whether to stop (all-passed or budget reached), and a syntactic **Timing** (`@:!`) gates a third semantic Imperative that revises from bounded inputs {`prompt`, `prior draft`, `prior checks`} — and nothing else. The loop self-seeds: each iteration conditionally appends its `{new draft}` to `[all drafts]`, which is also the loop's input collection, so the budget cap |`[all drafts]`|=3 ends the loop without an external scheduler. Final scoring uses IFEval's shipped checkers (not the plan's synthesised ones), so the agent acts honestly and the paper's numbers come from independent ground truth.

```
(OUTPUT) compliant answer
  (ACTION) is obtained by taking the last draft from all drafts.
  (VALUE) verifiers
    (ACTION) is obtained by synthesising one deterministic
      Python checker per constraint stated in the prompt.
  (VALUE) initial draft
    (ACTION) is obtained by generating an answer for the prompt.
  (VALUE) all drafts
    (ACTION) is obtained by initialising the chain with the
      initial draft.
  (VALUE) reviewed drafts
    (ACTION) is obtained by iterating over all drafts; each
      iteration, given (CONTEXT) prior draft, yields:
      (VALUE) prior checks
        (ACTION) is obtained by running each checker in
          verifiers on prior draft.
      (VALUE) prior passed
        (ACTION) is evaluated by checking whether every entry
          in prior checks is marked passed.
      (VALUE) should stop
        (ACTION) is evaluated by checking whether prior passed
          is true or all drafts already has three elements.
      (VALUE) new draft  (only when should stop is false)
        (ACTION) is obtained by revising prior draft to fix
          failing constraints; given prompt, prior draft, and
          prior checks.
      (VALUE) all drafts  (only when should stop is false)
        (ACTION) is obtained by appending new draft.
  (CONTEXT) prior draft (current element of all drafts)
```

Figure 2: The IFEval plan as compiled `.ncn` (leaf re-references elided; bare `.ncds` in App. H).

6.3 Conditions, Protocol, Metrics

Five conditions per prompt. B1 Single-shot original prompt → LLM → output (1 call), establishing the published-baseline floor. **B2** Constraint-aware prompting: system prompt augmented with an instruction to obey every stated constraint and self-check before responding (1 call). This is the *matched-information baseline*: it distinguishes "more information" from "structured plan." **B_COT** Chain-of-thought single-shot: system prompt requires identifying constraints, reasoning step-by-step, then emitting a `FINAL_ANSWER:` block (1 longer call). **B3** Self-refine (Madaan

et al., 2023) at matched compute: a single chat session with up to $K{=}3$ critique-revise turns (no per-constraint deterministic checks; the model generates and applies its own critique in natural language). This is the *matched-compute baseline*: it distinguishes "more compute" from "structured plan." **NC** the NormCode plan above. Without B2 a positive result for NC is confounded by "more information"; without B3 by "more compute"; without B_COT by "more reasoning text."

Protocol. All five conditions run on `qwen-turbo-latest` (DashScope, OpenAI-compatible) at $T{=}0$, with identical model and decoding parameters across conditions — the only variable is the prompting / plan structure. A smaller model is deliberate: on IFEval, frontier models leave little headroom and any structural effect would be invisible under saturation. We report IFEval's four published metrics per condition.

6.4 Findings and Illustrative Artifacts

Table 2 reports the four IFEval metrics on the 209-prompt hard subset (full per-instruction-type breakdown in Appendix L). The headline reading: *NormCode preserves competitive constraint-following while producing audit/revision artifacts unavailable to the baselines*. Three findings support it. **(F1) NC preserves aggregate compliance**: prompt-strict 0.801 vs B2 0.807 (within 1 pp); the plan adds structure without an accuracy cost. **(F2) Matched-compute self-refine hurts**: B3 scores -6.4 pp below B1 (0.725 vs 0.789); iteration without a deterministic checker is harmful for this model — the critique step hallucinates failures and rewrites previously-passing content. **(F3) NC dominates the hardest constraint family**: on `length_constraints:number_words` NC scores 0.97 (33/34) vs B3 0.76, B1 0.71, B2 0.62, B_COT 0.50. Since B3 shares NC's $K{=}3$ budget but lacks deterministic checkers, the $+21$ pp NC$-$B3 gap isolates the value of *checker-driven* revision, not iteration alone. The cost is real (rightmost column): NC consumes $\sim 500\times$ B2's tokens to preserve B2-level compliance, an overhead largely attributable to the current runtime's composition pipeline rather than the plan itself (Appendix L).

The same execution produces three artifacts unavailable to the baselines, exhibiting P1–P3 at no extra LLM cost (full exhibits in Appendix H H.4–H.5). **(A1, P2)** A reviewable `.ncn` narrative emitted at compile time, from which the Reviewer reads off *what each round does, what the reviser sees, where constraints are applied*. **(A2, P1)** A bounded-input snapshot at a failed verification node — e.g., the round-1 `number_words` rejection on prompt 1069 (draft 1: 414 words; draft 2: 556, passes) — showing only the declared inputs `{verifier spec}`, `{draft}`; the accumulated-trace baseline at the same point contains hundreds of tokens of role-tagged context. **(A3, P3)** A reviser-intervention example: tightening `number_words` from `at_least:300` to `at_least:500` and resuming re-fires only Down(`[verifier specs]`); the dump of re-fired vs. cache-served flow indices is the exhibit. With aggregate compliance preserved, A1–A3 are the contribution the baselines cannot replicate.

Condition	Pr-S	Pr-L	In-S	In-L	tok.
B1	0.789	0.854	0.901	0.932	383
B2	**0.807**	**0.877**	**0.911**	**0.943**	385
B_COT	0.708	0.766	0.867	0.893	948
B3	0.725	0.778	0.869	0.893	2,387
NC	0.801	0.854	0.901	0.930	192,245

Table 2: IFEval metrics on the 209-prompt hard subset (57 three-constraint + 152 two-constraint; `qwen-turbo-latest`, $T{=}0$, single seed). Pr-S/Pr-L: prompt-level strict/loose; In-S/In-L: instruction-level strict/loose; tok.: average tokens per prompt. Conditions: B1 single-shot; B2 constraint-aware prompt; B_COT chain-of-thought; B3 Self-Refine ($K{=}3$); NC NormCode plan ($K{=}3$). NC preserves B2-level constraint-following at $\sim 500\times$ the token cost while B3 *degrades* below B1 by 6.4 pp on prompt-strict — and NC additionally produces audit/revision artifacts (A1–A3) the baselines cannot.

7 Related Work

The framing of this paper's contribution against the existing stack is deliberately *compositional*, not adversarial. Each technique below continues to do useful work as a recruited capability at a semantic node inside a `.ncds` plan; what none supplies, at the granularity of an artifact a non-developer binding party recognises, is the runtime binding load *across* nodes. That load is borne by the scope-rule-bound runtime plus the `.ncds` artifact.

LLM agent frameworks and visual UIs. LangChain (Chase, 2022), LangGraph (LangChain Inc., 2024b), AutoGen (Wu et al., 2023), and PromptFlow (Microsoft, 2023) pass information between steps through implicit mechanisms (accumulated conversation history, shared Python state,

templated references to prior outputs), so the artifact a Reviewer might read is not the artifact the orchestrator executes. Node-based visual UIs such as ComfyUI (comfyanonymous, 2023), Flowise (FlowiseAI, 2023), and Langflow (LangChain Inc., 2024a) inherit the same implicit-state problem; Canvas's graph view is generated from a compiled formal plan with declared scopes, so the graph *means* a projection of a formal artifact rather than visual scripting. A NormCode imperative node's body may dispatch to any of these frameworks as a leaf capability.

Programmatic prompting, neurosymbolic, and tool-using LLMs. DSPy (Khattab et al., 2024) optimises prompts against a declared signature; ReAct (Yao et al., 2023), Toolformer (Schick et al., 2023), and chain-of-thought (Wei et al., 2022) reshape what an LLM does *at* a step. These are per-node concerns and compose inside a semantic node; none addresses the multi-step workflow surface or the multi-role collaboration problem.

Iterative refinement and self-critique. Self-Refine (Madaan et al., 2023) composes draft and critique through accumulated context. It is the matched-compute baseline (B3) in §6; the NormCode plan reuses the same drafter and reviser LLM calls but the scope rule turns the critique into a typed list of deterministic check results at a named flow index rather than appended text in the next prompt.

Tracing and human-in-the-loop. LangSmith (LangChain Inc., 2023), Langfuse (Langfuse, 2023), and W&B Prompts (Weights and Biases, 2023) format post-hoc accumulated state for human reading — they improve the Operator's experience but operate downstream of the structural problem. RLHF (Ouyang et al., 2022), Constitutional AI (Bai et al., 2022), and red-teaming (Perez et al., 2022) intervene during or after probabilistic computation. The `.ncn` review, to our knowledge, is the only LLM-system intervention that costs zero LLM calls and operates on the structure of the workflow before any execution.

Workflow languages and CBR. BPEL (OASIS, 2007), CWL (Amstutz et al., 2016), and Nextflow (Tommaso et al., 2017) formalise non-stochastic, well-typed compute graphs but do not address LLM-specific concerns. Canvas's checkpoint/fork/override mechanism draws on case-based reasoning (Aamodt and Plaza, 1994; Bergmann, 2002; Minor et al., 2014); we focus on the human-AI collaboration interpretation of the same machinery. Our IFEval demonstration (§6) wires per-instruction checkers from Zhou et al. (2023) as `python_script` norms on Judgement nodes inside a single generic plan.

8 Discussion

What scales. The same three markers and six inference kinds compose at any depth, each compiled plan is a deterministic artifact re-instantiable with new inputs, and the norm at any semantic node can be swapped (prompted LLM, Python tool, fine-tuned head, RAG variant, agent harness) as plan metadata. The plan-artifact response is intended to apply beyond LLM agents to other multi-role procedures (scientific pipelines, regulated financial workflows, clinical pathways), though such generalisation is not exercised here.

What does not scale yet. Reviewer attention scales with plan size (structural diffs and change highlighting are not yet implemented); LLM-assisted Derivation can produce a wrong formal plan that nonetheless compiles — the `.ncn` review is the defence but transfers responsibility to the Reviewer; plan library curation grows with deployments.

When the bargain is worth paying. The scope rule carries an authoring cost — unjustified for one-developer, one-user, one-time workflows, worthwhile when multiple roles engage with the plan, when wrong-execution cost outweighs authoring overhead, or when partial-recovery economics dominate. NormCode is one constructive witness, not the only viable design; review-equals-runtime does not eliminate dispositional risk inside semantic leaves.

9 Conclusion

NormCode factors plans into value/functional concepts, splits functional concepts into semantic Imperatives/Judgements (LLM-invoking, norm-bound) and syntactic helpers `$`/`&`/`@`/`*`, and binds them with an indentation-defined scope rule whose two-projection compilation yields per-node addressability, pre-execution reviewability, and scope-bounded revisability — capacities current frameworks do not offer at Reviewer-facing granularity. On IFEval, NormCode preserves competitive constraint-following (+21 pp over matched-compute self-refine on the hardest family) while producing audit/revision artifacts unavailable to the baselines.

Limitations

LLM-assisted Derivation can produce an incorrect formal plan. The compiler's Derivation phase uses an LLM to extract concepts and dependencies. The output is a formal plan that subsequent phases process deterministically, but it may not match the Designer's intent. The `.ncn` review is the structural defence but transfers verification responsibility to the Reviewer.

Reviewer attention scales linearly with plan size. A multi-node plan produces a multi-page narrative; sustained attention is required to verify structural defects. Production deployment will require Reviewer-aid tooling — structural diffs, change highlighting, automated defect-pattern checks — none of which is implemented in the current Canvas.

Scope-rule authoring overhead. Designers must declare every input scope explicitly. This is the price of the multi-role bargain. For one-developer, one-user, one-time workflows the bargain is not worth paying.

Semantic inferences remain probabilistic. P1, P2, and P3 are structural properties of the workflow surface; they do not bound the correctness of individual Imperatives and Judgements. A structurally faithful plan can still produce wrong outputs at any semantic node.

P1 and P3 are argued, not measured. P1 (per-node addressability) and P3 (scope-bounded revisability) are derived as $O(1)$ and $O(|\mathrm{Down}(f)|)$ consequences of the scope rule (§5); we present them as design properties of the language rather than as empirically measured capacities. Controlled latency benchmarks against LangGraph-style implicit-state baselines would strengthen the structural argument and are deferred follow-up work. P2 (pre-execution reviewability) is exhibited by the A1 artifact rather than measured against a reviewer task distribution.

Empirical scope. The constructive demonstration is on a 209-prompt hard subset of IFEval (`qwen-turbo-latest`, $T{=}0$, single seed); the constrained-output regime is exactly where per-constraint deterministic Judgements are expected to help. NormCode preserves competitive constraint-following (NC 0.801 vs B2 0.807 prompt-strict) and obtains a structurally clean $+21$ pp gap over matched-compute self-refine on `length_constraints:number_words`, but the accuracy contribution is *not* across-the-board outperformance — it is competitive aggregate compliance plus the three audit/revision artifacts the baselines cannot produce. Tasks with tacit constraints, creative outputs, or no closed verifier vocabulary are out of scope. The two-level CBR study, SWE-bench Lite recovery-cost benchmark, and multi-role qualitative study referenced in earlier drafts are deferred follow-up evidence and not part of the current submission.

Compute cost. At the headline accuracy NC consumes $\sim 500\times$ B2's tokens and $\sim 65\times$ B2's wall-clock per prompt. Most of that overhead is the current orchestrator's composition pipeline (~ 10 inner LLM calls per "semantic" inference for prompt assembly) rather than a fundamental property of the plan. A leaner compilation of the same plan should approach B3's call count for the same semantic budget; we cannot yet measure this. Production deployment of NormCode should weigh the audit-trail value against this cost, not assume parity.

Constraint-extraction is itself a semantic node. The plan's first phase extracts `[verifier specs]` from the natural-language prompt via an LLM call. This step is reviewable in the `.ncn` and recoverable by P3, but it is not deterministic. A mis-extraction is one of the failure modes the scope rule does not eliminate.

Self-refine baseline is a proxy for "more compute." The matched-compute argument relies on Self-Refine (Madaan et al., 2023) as the unstructured-revision baseline. A LangGraph re-implementation of the same plan would be a stronger compositional baseline; it is deferred for budget reasons.

Environmental replication is outside the language. A checkpoint is structurally self-contained, but resuming from it requires referenced external resources to remain accessible. The plan encodes *what* was used; whether the environment persists is outside the language's control.

The Canvas UI is a single implementation. The derivations of P1 and P3 hold for any implementation of the scope rule, but constant factors depend on Canvas-specific UI choices.

Ethical Considerations

Intended use. The intended use cases are settings where the cost of wrong execution justifies the scope-rule authoring overhead: regulated workflows (financial, legal, clinical), workflows de-

ployed at scale with non-developer operators, and workflows whose outputs are inputs to consequential human decisions.

Risk of false trust in reviewed plans. P2 reduces the gap between what a Reviewer approves and what executes only at the *structural* level. A structurally-faithful plan can still produce wrong outputs at semantic nodes. A Reviewer who interprets `.ncn` approval as guaranteeing correct behaviour will be misled. Deployment should pair Reviewer training with explicit distinction between structural approval and output approval.

Reviewer accountability. Making non-developer Reviewers a load-bearing element of the workflow surface raises the stakes of Reviewer error. A Reviewer who approves a defective plan in a regulated setting may bear professional or legal liability they did not bear when the plan was developer-only.

Multi-agent configurations. The system supports per-inference assignment of different LLM agents. A plan that mixes agents may produce results whose failure modes are harder to predict than any single agent's. We recommend plan metadata always include the agent configuration that produced any given checkpoint.

Data residency and perceptual signs. In multi-tenant deployments, checkpoints store pointers rather than content — but pointers still encode access patterns. Database-access policies should reflect this.

Reproducibility. IFEval (Zhou et al., 2023) is publicly available under the Apache 2.0 licence, including the 541 prompts, their pre-extracted constraint structure, and the Python checker implementations. The NormCode plan source (`.ncds`), the post-formalised plan, the compiled `concept_repo.json` / `inference_repo.json`, the provisions (LLM prompts and deterministic Python scripts the plan calls), the harness that runs B1/B2/B_COT/B3/NC against `qwen-turbo-latest`, the per-(condition, prompt) result JSONs, and the aggregate summaries on the n=209 overlap and the partial n=541 full subset will be released as supplementary material under an open licence. The three illustrative artifacts (A1 `.ncn` narrative, A2 bounded-input snapshot, A3 reviser-intervention dump) are included as concrete exhibits, accompanied by three side-by-side sample cases (`NC_wins`, `all_pass`, `NC_loses`).

No human-subjects component in the current submission. The current submission contains no human-subjects evidence; the deferred multi-role study (Limitations) is the only such component planned.

Use of AI Assistants

In preparing this submission we used general-purpose conversational large-language-model assistants for writing-support tasks: drafting and revising LaTeX prose (sections, captions, the abstract), compressing paragraphs to meet page limits, porting markdown drafts to LaTeX, and cross-checking section and label references for consistency. The research design, technical claims, language specification, experimental protocol, plan sources (`.ncds`), the NormCode compiler and orchestrator implementation, the related-work positioning, and all final wording were authored and verified by the human authors, who take full responsibility for the manuscript.

References

Agnar Aamodt and Enric Plaza. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59.

Peter Amstutz, Michael R. Crusoe, and Nebojša Tijanić. 2016. Common Workflow Language, v1.0. *Figshare*.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, and 32 others. 2022. Constitutional AI: Harmlessness from AI feedback. arXiv preprint arXiv:2212.08073.

Ralph Bergmann. 2002. *Experience Management: Foundations, Development Methodology, and Internet-Based Applications*. Springer LNAI 2432.

Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain.

comfyanonymous. 2023. ComfyUI. https://github.com/comfyanonymous/ComfyUI.

FlowiseAI. 2023. Flowise. https://github.com/FlowiseAI/Flowise.

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. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *Proceedings of ICLR*.

LangChain Inc. 2023. LangSmith. https://www.langchain.com/langsmith.

LangChain Inc. 2024a. Langflow. https://github.com/langflow-ai/langflow.

LangChain Inc. 2024b. LangGraph. https://github.com/langchain-ai/langgraph.

Langfuse. 2023. Langfuse: Open Source LLM Engineering Platform. https://langfuse.com.

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 *Proceedings of NeurIPS*.

Microsoft. 2023. PromptFlow. https://github.com/microsoft/promptflow.

Mirjam Minor, Stefania Montani, and Juan A. Recio-García. 2014. Process-oriented case-based reasoning. *Information Systems*, 40:103–105.

OASIS. 2007. Web Services Business Process Execution Language Version 2.0. OASIS Standard.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Proceedings of NeurIPS*.

Ethan Perez, Saffron Huang, Francis Song, Trevor Cai, Roman Ring, John Aslanides, Amelia Glaese, Nat McAleese, and Geoffrey Irving. 2022. Red teaming language models with language models. In *Proceedings of EMNLP*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Proceedings of NeurIPS*.

Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35:316–319.

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 *Proceedings of NeurIPS*.

Weights and Biases. 2023. W&B Prompts. https://wandb.ai/site/prompts.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *arXiv preprint arXiv:2308.08155*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of ICLR*.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. arXiv preprint arXiv:2311.07911.

A NormCode Language Reference

This appendix gives the concrete syntax and semantics of NormCode at the level needed to read every fragment in the body. It complements §3: the body inlines the essentials and refers back to this reference for the full vocabulary, including the eight concept types, the four syntactic operator families, the unified modifier system, flow indices, and the `.ncds` → `.ncd` relationship.

A.1 The three concept markers

A `.ncds` draft consists of *concept declarations* written one per line. Each line begins with one of three markers; indentation defines scope.

Marker	Name	Role
`<-`	**Value Concept**	Declares a named value, intermediate result, or input.
`<=`	**Functional Concept**	Declares the operation that produces the value above it; the body of the operation distinguishes its kind (Imperative, Judgement, or one of the four syntactic operators below).
`<*`	**Context Concept**	Declares per-iteration state inside a loop; bound by the enclosing Looping operator.

A fourth lexical element, the **grounding operator** `::`, binds a Functional Concept to a specific tool (e.g., `:: read the file` grounds an Imperative to a file-system tool); see §A.5.

A.2 The scope rule

> **A Functional Concept at depth d receives, as its declared inputs, exactly the Value Concepts (and Context Concepts) declared one level below it, at depth $d + 1$, in the same indented block — and nothing else.**

Consequences. An operation *cannot* read accumulated conversation history, cannot reach upward to an enclosing scope, cannot inherit a sibling's outputs, and cannot accept implicit globals — any value needed must be *re-declared* at the depth where it is used. The compiler verifies this statically in the Formalization phase (Appendix C); the orchestrator additionally enforces it at runtime by passing only the scope-declared tensors to each dispatched inference.

A.3 An annotated example

A two-step summarisation plan, with one tool-grounded reader and one prompted summariser:

```
<- summary                      # value at 1 (output of the plan)
    <= summarize the findings   # functional at 1.1: an Imperative
                                #   inputs declared one level down:
    <- report                   # value at 1.1.1
        <= :: read the file     # functional at 1.1.1.1: an Imperative
                                #   grounded to a file-system tool
    <- style_guide              # value at 1.1.2: a static external input
```

Reading order is *top-to-bottom by flow index, inside-out by dependency*: the innermost `1.1.1.1` fires first, producing `report`; `report` and `style_guide` are then the only declared inputs of `summarize the findings`. The outer Imperative does not see the file contents directly, does not see the caller's context, and cannot accumulate state across runs.

A.4 Concept types

Every concept name in `.ncds` carries a type implied by its bracketing. The type vocabulary partitions into non-functional types (the "nouns" of the plan) and functional types (the "verbs").

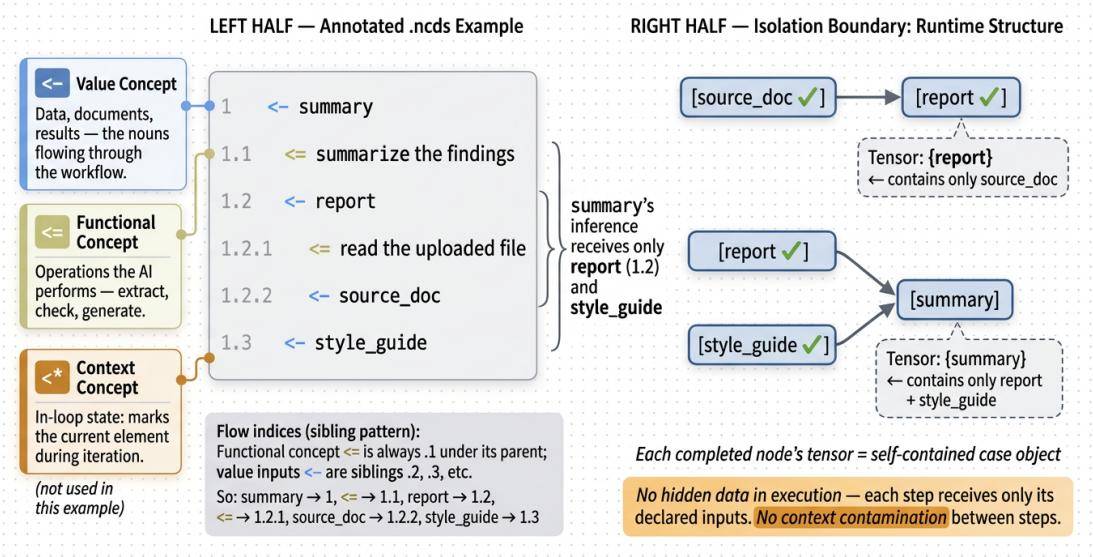

Figure 3: NormCode syntax and the scope rule. Left: an annotated `.ncds` excerpt showing the three markers and indentation. Right: the runtime isolation boundary — each node's tensor is self-contained, addressable at its flow index, and visible only to inferences declared within the enclosing indented block (P1).

Non-functional concepts (data).

- `{name}` — **Object**: a named entity holding concrete data (string, integer, file pointer, structured record). Example: `{document}`, `{result}`.
- `<name>` — **Proposition**: a boolean state of affairs read out by a Judgement and consumed by a Timing operator. Example: `<file exists>`, `<all checks passed>`.
- `[name]` — **Relation**: a collection (list or dict) iterated by a Looping operator. Example: `[input files]`, `[verifier specs]`.
- `:S:` — **Subject**: the agent or system executing the operation (a body of tools and capabilities). Often left implicit; declared when a plan mixes agents.

Functional concepts (operations).

- `::(action)` — **Imperative**: an LLM-invoking operation that *creates* a Value from its declared inputs. Each Imperative carries a *norm* (§A.5) that fixes *how* the call is performed.
- `<{check}>` — **Judgement**: an LLM- or tool-invoking operation that *decides* a Proposition from its declared inputs. Also parametrised by a norm.

Boundary markers.

- `:>:{name}` — declared input at the plan's outermost scope.
- `:<:{name}` — declared output at the plan's outermost scope.

The role of these eight types is that NormCode can be read at one glance: the bracketing tells the Reviewer whether each name is data (and of what shape), an operation (and of what kind), or a plan-boundary input or output — before reading the operation's description.

A.5 Norms: the execution paradigm of a semantic node

A Functional Concept is incomplete without a *norm*. The norm fixes the paradigm under which the operation is executed and is what makes a semantic node a *configurable* LLM call rather than an opaque prompt. The compiler annotates the norm in the Post-Formalization phase (Appendix C).

Norm	What the orchestrator does at dispatch
`composition`	Assemble a prompt from the declared inputs and dispatch to the configured LLM.
`python_script`	Call a deterministic Python tool with the declared inputs as positional or keyword arguments.
`method_NormCode`	Recurse into a nested `.ncds` plan (self-hosting).
`recruited`	Dispatch to a registered external capability: a fine-tuned head, a RAG pipeline, a structured-output decoder, an agent harness.

A consequence of treating the norm as a per-node parameter is that NormCode *composes with the existing stack*: every framework, declarative DSL, agent harness, and post-training technique surveyed in §7 can serve as a norm at a single semantic node. The cross-node binding load — declaring which value goes where, when, and under what scope — is borne by the language and its runtime.

A.6 Syntactic operators: deterministic tensor reshaping

Syntactic operators are deterministic, LLM-free reshapings of declared tensors. There are four families.

Family	Markers	Meaning
Assigning	`$=`, `$%`, `$.`, `$+`, `$-`	Route data: alias (`$=`), literal (`$%`), pick first valid (`$.`), append (`$+`), extract (`$-`).
Grouping	`&[{}]`, `&[#]`	Collect declared concepts into a structured dict / list.
Timing	`@:'`, `@:!`, `@.`	Gate execution on a Proposition: yield-if (`@:'`), yield-if-not (`@:!`), or sequence (`@.`).
Looping	`*.`	Iterate over a Relation with optional carried Context.

Syntactic operators contribute zero LLM cost. The combination of Looping (over a Relation), Timing (gating on a Judgement's Proposition), and Assigning (selecting the first all-passing draft) is the standard NormCode idiom for "draft, verify, revise, select" loops; the IFEval plan in Appendix H is a worked example.

A.7 The modifier system

Modifiers attach metadata to a concept declaration. They are written with a percent sign followed by a symbol and an argument:

Modifier	Meaning
`%>`	**Source**: this concept is read from a named upstream value.
`%<`	**Target**: this concept is written to the named downstream value.
`%:`	**Axis**: declares a named dimension of indeterminacy on a tensor.
`%^`	**Carry**: in-loop state propagated from the previous iteration.
`%@`	**Index**: a positional index into a tensor axis.
`%+`	**Create**: introduces a new axis on the target tensor.
`%-`	**Collapse**: collapses an axis on the source tensor.

A Looping construct typically combines all four kinds: `*. %>([items]) %<({out}) %:(item) %@(1)` reads as "iterate over the `[items]` Relation along axis `item` from position 1, writing each result into `{out}`."

A.8 Flow indices and the sibling pattern

Every concept and operation receives a hierarchical address during Formalization. The Functional Concept that produces `summary` is at `1.1`; its declared inputs are siblings at `1.1.1`, `1.1.2`, ... (not children at `1.1.1.1`). The inner operation that produces `1.1.1`'s value is itself at `1.1.1.1`.

```
<- summary                      1
    <= summarize the findings   1.1
    <- report                   1.1.1
        <= :: read the file     1.1.1.1
    <- style_guide              1.1.2
```

This sibling pattern is what makes Operator inspection $O(1)$ in plan depth (§5): given a flow index, the bounded inputs are exactly the indices one level deeper in the same scope. Every checkpoint stored by the orchestrator is keyed by `(run_id, flow_index)`, which is the stable address the Operator and Reviser use throughout the workflow.

A.9 Authoring (`.ncds`) vs. formal (`.ncd`)

`.ncds` is the human-written form. The compiler emits a fully-annotated `.ncd` that makes every flow index, paradigm ID, axis declaration, and sequence type explicit. The same two-step summary above compiles to:

```
:<:{summary} | ?{flow_index}: 1
    <= ::(summarize the findings) | ?{flow_index}: 1.1
                                  | ?{sequence}: imperative
                                  | ?{paradigm}: composition
    <- {report} | ?{flow_index}: 1.1.1
        <= ::(read the file) | ?{flow_index}: 1.1.1.1
                             | ?{sequence}: imperative
                             | ?{paradigm}: python_script
                             | ?{tool}: file_system
    <- {style_guide} | ?{flow_index}: 1.1.2
```

`.ncd` is the source of truth for the orchestrator; `.ncn` (Appendix B) is the plain-prose projection generated deterministically from the same `.ncd`. No LLM is invoked between `.ncd` and either projection — this is the structural basis for the review-equals-runtime property of §3.4.

A.10 Perceptual signs

Perceptual signs are pointers to content stored outside the Concept Repository (files, database rows, API responses, large model outputs). Syntax: `%{norm_type}id(signifier)`. The orchestrator performs *transmutation* (sign → content) at the moment an inference needs the value, and otherwise passes the sign unchanged. The benefit is two-fold: (i) checkpoints remain small even when the underlying content is large, and (ii) tenant-data-residency policies can be enforced at the storage layer without modifying plans. Appendix E gives the full reference/axis model.

B The `.ncn` Narrative Format

The `.ncn` is a compiler-generated plain-prose narrative of a formal plan. It is produced at the end of Phase 2 of the compilation pipeline (Appendix C) — before any execution resource is assigned, before any LLM is called. It is **not** a description of what the system does; it is a verbatim translation of the formal plan structure into prose that any stakeholder can read. The same formal plan deterministically maps to exactly one `.ncn` string (modulo whitespace), so the Reviewer-facing artifact is identical-by-construction to what the orchestrator executes.

B.1 Token mapping

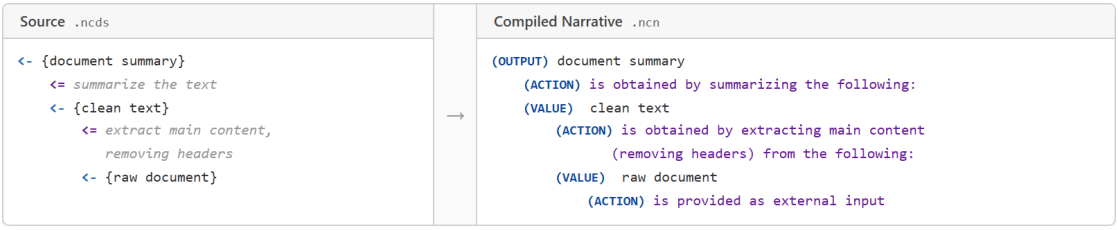

Figure 4: The .ncn narrative compiled from a .ncds source. Left: three-marker .ncds source. Right: deterministic plain-prose rendering used for pre-execution review (P2). A non-developer Reviewer reads the narrative to verify that the plan's logic matches intent before any LLM call.

Formal element	.ncn token
<- Data marker	(VALUE)
<= Action marker	(ACTION)
<* State marker	(CONTEXT)
terminal output marker	(OUTPUT)
flow index	rendered as a referent prefix

Sentence templates per inference paradigm. Imperative: *"(ACTION) is to {operation_verb} the following:"*. Judgement: *"(ACTION) checks whether {condition} holds, given:"*. Iteration: *"(ACTION) is obtained by iterating over {collection}, where each iteration:"*. Conditional: *"(ACTION) is executed only if {condition}:"*. Specification: *"(VALUE) is selected from the first non-skip among:"*.

B.2 Worked example: PPT slide generation (nested loop)

.ncds source (outer + inner loop, excerpt):

```
<- slides
    <= *. for each section in [outline_sections]
    <- *<{outline_section}>
    <- {style_guide}
    <- slide
        <= ::(generate slide content from outline_section + style_guide)
        <- *<{outline_section}>
        <- {style_guide}
```

.ncn rendering (excerpt):

> "(OUTPUT) **slides** is obtained by iterating over (VALUE) outline_sections, where each iteration produces a (VALUE) **slide**. To produce (VALUE) slide, (ACTION) generates slide content from the current outline section and the style guide; the inputs available to this action are exactly (VALUE) outline_section (the current iteration value) and (VALUE) style_guide."

A Reviewer reading the .ncn can verify, before any LLM call: (i) that slide generation is per-section, not over the full outline; (ii) that the style guide is passed into every iteration; (iii) that no other ambient state is consulted. These are properties of the plan, not of any particular run.

B.3 What .ncn does *not* capture

Paradigm IDs, tool faculties, agent assignments, and tensor axis declarations are post-Formalization concerns and intentionally absent from the .ncn. The narrative is what a non-developer Reviewer needs to verify the **structure** of the plan; the technical bindings are reviewed by the Designer.

B.4 Reviewer compliance checklist (template)

For each plan, the Reviewer answers a fixed set of structural questions against the `.ncn`. A 10-row template is included in the supplementary materials. Example rows: "What external inputs does this plan require?", "Can the slide generator access the full outline?" (expected: no — `outline` is not in its declared scope), "Is there any accumulated state between iterations?", "Are there any LLM calls before the Review step?" (expected: no — `.ncn` is produced at compile time). A rejection on any row sends the Designer back to revise the `.ncds`; the recompile produces a fresh `.ncn` for re-review. No LLM has been called.

C Compilation Pipeline

The compiler maps a `.ncds` source to two projections — a `.ncn` narrative for review and a pair of JSON repositories for execution — through four phases. Only Phase 1 involves an LLM.

C.1 Phase 1 — Derivation (LLM-assisted)

Input: `.ncds` draft (free-form). **Output:** a hierarchical concept tree with declared dependencies. **Mechanism:** a fixed prompt template that walks the source bottom-up, asking the LLM to identify each `<-`/`<=`/`<*` token, classify the inference paradigm, and list the declared inputs.

Failure modes (the only ones not eliminated by later phases): hallucinated dependency, missed iteration, wrong inference paradigm. The Reviewer's `.ncn` read is the structural defence (§5, §6). Phase-1 failure rate on the case-study plans is logged in the released compiler trace.

C.2 Phase 2 — Formalization (rule-based)

Input: the hierarchical concept tree from Phase 1. **Output:** a flow-indexed formal plan + the `.ncn` narrative. **Mechanism:** deterministic — assigns flow indices via the sibling pattern (`1`, `1.1`, `1.1.1`, ...) with `.1` reserved for the producing action; infers sequence types (iteration, conditional, parallel, sequential) from the marker stack; renders the `.ncn` using the templates in Appendix B. Given a fixed Phase-1 output, Phase 2 produces exactly one formal plan and exactly one `.ncn` string.

C.3 Phase 3 — Post-Formalization (annotation)

Input: the formal plan. **Output:** the same plan annotated with paradigm IDs, tool faculties, tensor axis declarations, and per-inference agent assignments. **Mechanism:** rule-based; the paradigm taxonomy and the tool registry are external configuration. No LLM call. This phase produces the metadata an orchestrator needs but a Reviewer does not.

C.4 Phase 4 — Activation (serialisation)

Input: the post-formalised plan. **Output:** `concept_repo.json` (every declared concept with its scope, axes, and current value or perceptual sign) and `inference_repo.json` (every inference with its flow index, declared inputs, paradigm ID, agent assignment, and tool binding). These two files are what the orchestrator dispatches against; nothing else is consulted at runtime.

C.5 Self-hosting

The compiler is itself implemented as a NormCode plan (`nc_compilations.ncds`, Appendix F). Each of the four phases above is a separate inference grounded to a compiler tool. The same compiler binary therefore compiles its own source, providing a sanity check that the formal-plan abstraction is expressive enough to describe a real-world non-LLM workflow.

C.6 Recompilation discipline

When the Designer edits `.ncds`, Phase 1–2 always re-run (cheap; deterministic given the Derivation output). Phase 3–4 re-run only if the formal plan changed. The orchestrator detects structural diffs at the inference level and re-invalidates downstream checkpoints accordingly (§5, P3).

D System Architecture

A condensed account of the deployed Canvas system. Full integration details accompany the released code.

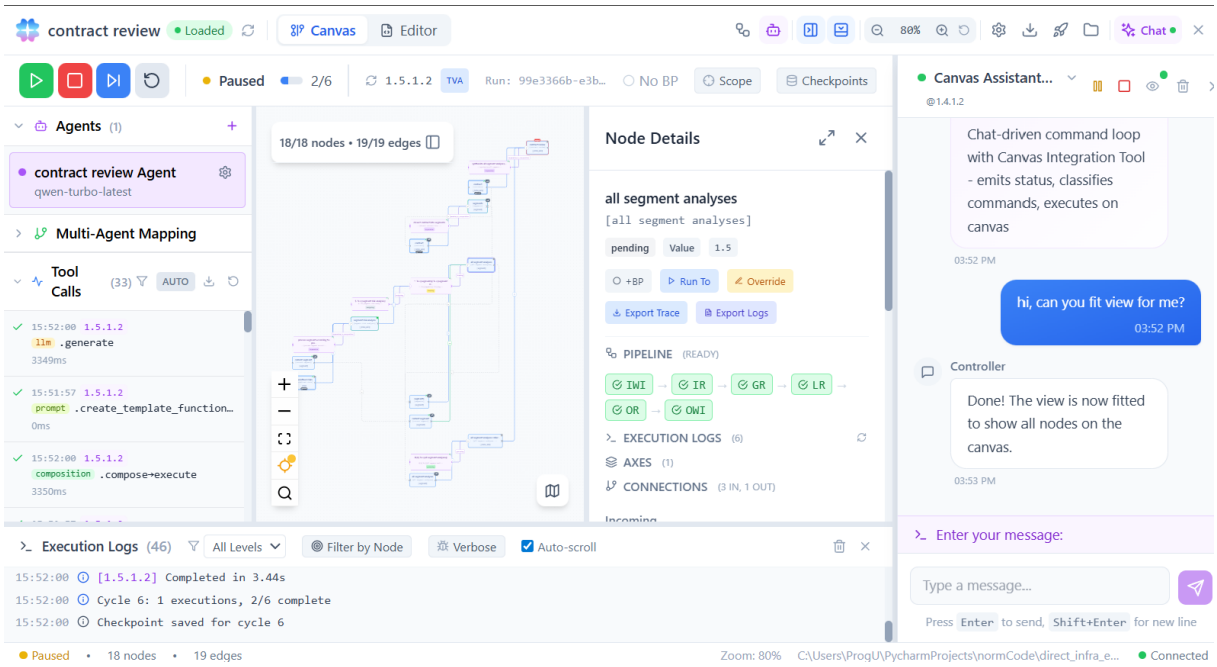

Figure 5: NormCode Canvas. The Graph View (centre) renders the full inference graph from the compiled plan; the Tensor Inspector (right) shows the bounded inputs of any selected node at its flow index — the visual surface of P1 (per-node addressability) for the Operator role. Top controls expose Fork and Value Override for the Reviser role (P3). The left panel shows the per-inference agent assignment.

D.1 Component overview

The Canvas system has a React 18 + TypeScript frontend (React Flow for the inference graph, Zustand state, WebSocket client) and a FastAPI backend with an async orchestrator and a SQLite checkpoint store. Frontend and backend communicate via REST for control operations and WebSockets for real-time execution events.

The orchestrator is a waitlist priority queue keyed by flow index. A blackboard tracks per-node status (`pending`/`running`/`completed`/`failed`/`skipped`); the Concept Repository (a keyed tensor store) holds the bounded inputs and current values. **Scope-rule enforcement happens at dispatch:** before invoking an inference at flow index f, the orchestrator gathers exactly the concepts declared in f's indented block from the Concept Repository — nothing else.

D.2 Runtime data flow (Operator path)

1. Designer compiles `.ncds` → `concept_repo.json` + `inference_repo.json` (Appendix C).
2. Operator opens the Canvas, sets optional breakpoints by clicking flow-index badges, presses Run.
3. The backend instantiates a run, the orchestrator dispatches inferences in flow-index order, the WebSocket streams per-node `running`/`completed` events.
4. At a breakpoint, dispatch pauses; the Tensor Inspector shows the bounded-input snapshot at that flow index. The Operator inspects, optionally modifies (becoming Reviser), and resumes — `Resume` from any flow index re-executes only the scope-downstream set (P3).
5. Every state change writes a row to the SQLite checkpoint table; `Fork` clones the run state at a flow index, creating a parallel run with shared upstream history and independent downstream execution.

Three trace views accompany this flow: an agent-centric view (LLM call detail at one flow index), a graph-centric view (scoped reference data between selected indices — the visual surface of P1), and an

orchestration-centric view (execution log with per-flow-index error chain and round-level breakdown). Screenshots are in the supplementary material.

D.3 Persistence schema (essentials)

Four tables: `plans(plan_id, ncds_source, concept_repo, inference_repo, ncn, compiled_at)`; `runs(run_id, plan_id, started_at, status, parent_run_id, fork_flow_index)`; `checkpoints(run_id, flow_index, blackboard_snapshot, concept_repo_snapshot, created_at, is_override)`; `tool_calls(run_id, flow_index, call_type, prompt, response, started_at, completed_at)`. A checkpoint is **self-contained** in the sense that it captures the full Concept Repository at a flow index — the row alone is sufficient to resume from that index, given the same plan and environment (cf. Limitations on environmental replication).

D.4 WebSocket protocol (event names)

Server → client: `node_status_change`, `tool_call_started`, `tool_call_completed`, `checkpoint_written`, `run_completed`, `run_failed`. Client → server: `set_breakpoint`, `clear_breakpoint`, `override_tensor`, `fork_at_checkpoint`, `resume_execution`. All events carry the `run_id` and the flow index they apply to.

D.5 Multi-agent assignment

Each inference can declare a `paradigm.agent` field naming a specific LLM provider/model. A single plan can mix agents — e.g., extraction by one model, drafting by another, judgement by a third. The orchestrator routes per-inference, the WebSocket carries the agent identifier with each tool-call event, and the checkpoint records which agent produced each value. Plan metadata always includes the agent configuration that produced a given checkpoint so that forked runs can be compared apples-to-apples.

E References, Tensors, and Perceptual Signs

The runtime data structure that makes a checkpoint self-contained.

E.1 N-dimensional named-axis tensors

Every declared concept is stored as an N-dimensional tensor with named axes. A scalar value uses a single `_none_axis`. Iteration produces a tensor with an additional axis named after the iteration variable (e.g., `outline_section`, in the PPT plan). Conditional skips do not delete a slot; they insert a `@#SKIP#@` token, propagated by downstream consumers. Loop invariants — concepts that are stable across iterations — use the `*-1` notation in the declared scope.

E.2 Reference operations

The orchestrator manipulates concepts through axis-aware operations: **index** (pick a single slot along an axis), **slice** (a range), **group** (collect across an axis into a list), **transpose** (swap two axes), **concatenate** (extend along an axis). These are all declared at compile time in `concept_repo.json` and dispatched at runtime; no LLM is invoked for reference manipulation.

E.3 Perceptual signs and transmutation

Content that lives outside the Concept Repository — files, database rows, large model outputs, multimedia — is referenced via a **perceptual sign**: `%{norm_type}id(signifier)`. The Concept Repository stores the sign, not the content. When an inference declares the concept as a bounded input and actually needs the value (not just the metadata), the orchestrator performs **transmutation**: it resolves the sign, fetches the content, passes it to the inference, and continues. Otherwise the sign is passed through unchanged.

This has two consequences for the architecture: (i) checkpoints stay small even when the workflow produces large artifacts — only the signs travel; (ii) data-residency policies on the underlying storage (file system permissions, database access controls) automatically apply at the time of resolution.

E.4 What is in a checkpoint

The full Concept Repository at a flow index f: each declared concept's name, axis names, axis shapes, and either values or perceptual signs. Plus the inference-level metadata needed for resumption: the plan ID, the run ID, the flow index, the blackboard status for downstream nodes. A checkpoint is **structurally** sufficient to resume; whether it is **operationally** sufficient depends on whether the perceptual-sign targets remain accessible — a constraint discussed in the Limitations section.

F Production Plans Gallery

The case-study plans cited in the body — Canvas Chat Assistant, PPT Generation, NC Compilations, and a 3D-model report generator — are not toy examples. They are the plans the development team uses to validate the system and the plans available to operators immediately after installation. Their variety demonstrates that NormCode can describe workflows from interactive canvas manipulation to document generation and meta-compilation.

F.1 Overview

Plan	`.ncds` file	Inferences	LLM calls per run	Key techniques
Canvas Chat Assistant	`canvas_assistant.ncds`	~20	~4–8	context-aware canvas manipulation, scoped Q&A
PPT Generation	`ppt_generation.ncds`	~30	~10–14	nested loop, 6 slide layouts, 14 component types
NC Compilations	`nc_compilations.ncds`	~25	~12–20	meta-compilation (NormCode compiling NormCode)
3D Model Report	`model_report.ncds`	~15	~6–10	multi-modal perceptual signs, structured output

Figure 6 shows core excerpts of three of these plans side-by-side; the full `.ncds` sources are in the supplementary material.

```
NC Compilations nc_compilations.ncds

<- compilation_result
  <= report compilation status
     and output paths
  <- task_specification
  <- project_context
  <- specification_analysis
      <= analyze specification:
         identify root output,
         ground inputs, transforms
      <- …
  <- ncds_draft
      <= derive hierarchical
         concept structure
      <- …
  <- formal_ncd
      <= formalize: assign flow
         indices, types, bindings
      <- …
  <- post_formal_ncd
      <= post-formalize: paradigms,
         shapes, provision paths
      <- …
  <- repositories
      <= activate into executable
         concept + inference repos
      <- …

<- Value   <= Functional   <* Context
```

```
PPT Generation ppt_generation.ncds

<- {演示文稿包}
  <= 完成演示文稿报告
  <- [内容参考]
      <= 对于内容参考元数据中的每个元数据
          <= 从文件路径加载内容
          <- {当前内容元数据}
      <- [内容参考元数据]
      <* {当前内容元数据}
  <- [模板参考] …
  <- [幻灯片规格]
      <= 运行extract_slide_specs脚本
      <- {大纲}
          <= 创建带幻灯片规格的大纲
          <- {分析}
              <= 分析演示文稿需求
              <- {演示主题}
              <- {目标受众}
              <- [内容参考]
          <- [内容参考]
  <- [所有已渲染的幻灯片]
      <= 对于幻灯片规格中的每个幻灯片规格
          <- {选定的html模板} …
          <- {html幻灯片}
              <= 通过用内容填充模板来渲染
              <- {增强后的幻灯片内容} …
              <- {幻灯片内容} …
          <- {幻灯片渲染结果} …
      <- [幻灯片规格]
      <* {当前幻灯片规格}
  <- {pptx演示文稿}
      <= 从幻灯片规格组装pptx
      <= 组装带导航的html幻灯片
      <- [所有已渲染的幻灯片]

<- Value   <= Functional   <* Context
```

```
Code Assistant code_assistant.ncds

<- task_result
  <= report completion
  <- user_request
      <= receive task
  <- project_context
      <= retrieve project state
  <- cycle_helper
      <= initiate as 0th cycle
  <- [cycle_results]
      <= for each ReAct cycle
          <= return edit results
          <- [exploration_results]
              <= explore: grep, reads
              <- user_request
          <- [change_plan]
              <= generate change plan
              <- all exploration_results
          <- [edit_results]
              <= for each change
                  <- edit_result …
                  <- lint_result …
              <- [change_plan]
              <* current_change
          <- verification
              <= run tests, build
          <- quit_react_loop?
          <- cycle_helper …
      <- cycle_helper
      <* this_cycle_count
  <- all_editing_results
      <= collect all results
      <- [cycle_results]

<- Value   <= Functional   <* Context
```

Figure 6: Core excerpts from three production NormCode plans as `.ncds` sources. Left: NC Compilations — sequential meta-compilation pipeline. Centre: PPT Generation — nested loop with section/slide-level scopes. Right: Canvas Chat Assistant — context-aware operation routing.

F.2 PPT Generation

Takes a topic string and produces a `.pptx` file plus an HTML preview. The structure is a nested loop over outline sections: the outer plan generates a section list and the inner loop generates a slide per section. The `style_guide` input is declared at the outer plan level and re-declared within the inner loop's scope so it is available to every slide-generation action without being re-fetched. The renderer is a grounded action (`::`) calling a Python tool; LLMs draft slide text and the tool produces the `.pptx`. About 30 inferences total, of which 10–14 are semantic (LLM calls); the rest are syntactic routing nodes at zero token cost.

F.3 NC Compilations

The self-hosting case: a NormCode plan that runs the four-phase NormCode compiler (Appendix C) as orchestrated inference steps. LLM calls handle the language-level reasoning (Derivation, Formalization sanity check); rule-based tool calls handle the formal transformations (Post-Formalization, Activation). Output: `.ncds`, `.ncd`, `.ncn`, and the compiled JSON repositories. An optional verification stage re-reads the generated `.ncn` and asks a Judgement inference whether the compiled plan matches the natural-language workflow description it was derived from. About 25 inferences and 12–20 LLM calls per run.

```
NC Compilations  nc_compilations.ncds

<- compilation_result
    <= report compilation status and output paths
    <- task_specification
    <- project_context
    <- specification_analysis
        <= analyze specification: identify root output,
           ground inputs, and transformations
        <- …
    <- ncds_draft
        <= derive hierarchical concept structure
           from specification analysis
        <- …
    <- formal_ncd
        <= formalize draft: assign flow indices,
           sequence types, and value bindings
        <- …
    <- post_formal_ncd
        <= post-formalize: assign paradigms,
           reference shapes, and provision paths
        <- …
    <- repositories
        <= activate post-formal plan into executable
           concept and inference repositories
        <- …

<- Value    <= Functional    <* Context
```

Figure 7: NC Compilations plan as `.ncds` source: the four-phase NormCode compiler implemented as a NormCode plan. Each phase is a separate inference grounded to a compiler tool, with declared inputs and outputs visible at compile time — the self-hosting evidence cited in §6.1.

F.4 Canvas Chat Assistant

The plan that powers the in-Canvas conversational interface. Classifies the user request, retrieves the relevant plan and canvas context (bounded to the user's current view), proposes a canvas-manipulation operation, dispatches the operation to the frontend. ~20 inferences; ~4–8 LLM calls per interaction. The Q&A scope is enforced by the scope rule — the assistant cannot accidentally read the entire run history,

only the concepts declared in its query block.

F.5 What the gallery demonstrates

Three structural properties of the plan-artifact medium become visible at deployment scale: (i) **variety of paradigms** — sequential pipeline (NC Compilations), nested iteration (PPT), conditional routing (Canvas Chat Assistant), multi-modal signs (3D Model Report) — all expressible with the same three markers; (ii) **bounded LLM-call counts** — every plan's LLM-call count is decidable from the formal plan alone, before any execution, because semantic vs syntactic inferences are distinguishable by paradigm; (iii) **self-hosting** — NC Compilations compiles other NormCode plans (and itself), evidence that the formal-plan abstraction is expressive enough to describe a real-world non-LLM workflow.

G Deferred Empirical Studies

Three empirical components referenced in earlier drafts are deferred to follow-up work. The body of this submission does not depend on any of them: the structural arguments in §5 (P1, P2, P3 follow from the scope rule plus orchestrator dispatch) are derivable from the grammar alone, and the headline empirical claim is the IFEval study (§6, Appendix H).

Multi-Role Study. A pilot study (N=6, three non-developer Reviewers + three non-developer Operators) used the case-study plans of Appendix F as stimuli with planted structural defects (missing input declaration, wrong loop scope, wrong sequence type, mis-named concept) to test whether the `.ncn` narrative is read with sufficient fidelity to localise the defect. Protocol, defect taxonomy, scoring rubric, and Cohen's κ calculation are documented in `experiments_plan.md` (supplementary). A scale-up protocol (N=24, three roles $\times$ eight participants) is outlined there for follow-up.

Failure-Localization Micro-Benchmark. A planted-error trace benchmark across Canvas Chat Assistant, NC Compilations, PPT Generation, and report-generation runs ($\sim$25 anonymised traces; single trained Operator counter-balanced across NormCode and accumulated-trace conditions) is described as a P1 illustration. Time-to-correct-identification and binary correctness are the scoring criteria. P1 is argued structurally in §5 and illustrated by artifact A2 (Appendix H, H.4); the micro-benchmark would make "$O(1)$ in n" a human-performance measurement rather than an architectural property.

Scope-Rule Ablation. A controlled ablation contrasting NormCode-with-scope-rule against NormCode-no-scope (orchestrator filtering disabled; full Concept Repository passed to each node) would be the load-bearing test that the scope rule, not Canvas, is the mechanism behind P3. Without the scope rule, recovery cost degrades to the baseline even with Canvas's UI affordances intact, because the dependency graph used to compute the stale set after a Reviser override no longer captures all data dependencies. The P3 argument is given structurally in §5; the ablation would make it empirical.

H IFEval Benchmark: Protocol and Bounded-Input Exhibit

Supports §6. Documents the protocol for the IFEval constructive demonstration, the bounded-input snapshot artifact A2, and the `.ncn` narrative artifact A1. The numerical results are reported in Appendix L.

H.1 Benchmark and subset

IFEval (Zhou et al., 2023, arXiv 2311.07911) ships 541 prompts with pre-extracted `instruction_id_list` + `kwargs`; each id maps to a published Python checker class, so scoring is deterministic. We use the published metrics: **prompt-level strict/loose** and **instruction-level strict/loose**. Local copy at `refs/ifeval/`.

The headline subset is **209 prompts**: all 57 three-constraint prompts (the compound regime where unstructured baselines drop constraints between draft and revision) plus 152 two-constraint prompts (the front of the stratified order; the remaining 27 two-constraint prompts and all 305 single-constraint prompts complete the full 541 in the ongoing run). The full n=541 partial results — with NC still in flight — appear in Appendix L N.6.

H.2 Five conditions

ID	Condition	LLM calls (avg)	Controls for
B1	Single-shot: *prompt* → *output*; system prompt "You are a helpful assistant."	1	Lower-bound floor
B2	Constraint-aware system prompt: tells the model to obey every stated constraint and self-check before responding	1	**Matched-information** (load-bearing)
B_COT	Chain-of-thought single-shot: identify constraints → reason step-by-step → emit `FINAL_ANSWER:` block	1	Reasoning-text variant
B3	Self-Refine (Madaan et al., 2023): single chat session with up to K=3 critique-revise turns; same model	2.0	**Matched-compute** (load-bearing)
NC	NormCode plan: synthesise checkers → draft → per-constraint deterministic verifier → conditional reviser (K=3), bounded inputs	37.5 raw / 2.46 revise rounds	The intervention

Same model (`qwen-turbo-latest`), same decoding parameters (T=0) across all five; differences in IFEval pass-rate are attributable to surrounding structure, not the generator. The raw 37.5 LLM-call count for NC reflects the current orchestrator's composition pipeline ($\sim$ 10 inner calls per "semantic" inference for prompt assembly), not 37 semantic revisions; the matched-compute axis is `revise_rounds` (2.46 for NC, 1.01 for B3).

H.3 The NormCode plan (full `.ncds`)

The single generic plan is authored once and instantiated per IFEval task. Below is the actual `.ncds` source as released in the supplementary archive (`plan/ifeval_plan.ncds`). The body figure (Figure 2) shows the same plan with action and judgement bodies elided; here we keep the load-bearing comments, section dividers, norm annotations, and the explicit STEP labels so the reader can follow the self-seeding loop step by step.

```
/: IFEval task plan -- "honest codegen": the agent synthesises its own checkers and is dual-scored by IFEval.
/: Shape:  Ground concepts (prompt, synthesised verifiers, initial draft, [all drafts]); then a self-seeding
/:      bounded loop. Each iteration: verify the prior draft, judge whether to stop (all-passed OR 3-draft
/:      budget reached), conditionally revise and append to [all drafts]. Loop terminates when the append
/:         is gated off. Final answer = the last draft in [all drafts].

:<: {compliant answer}
    <= take the last draft from [all drafts] as the final answer

    /: ================================================================
    /: GROUND CONCEPTS  (computed once, before the loop starts)
    /: ================================================================

    <- {prompt}                                        /: ground value, supplied externally

    <- [verifiers]
        <= ::(synthesise one deterministic Python checker per constraint stated in the prompt)
        <- {prompt}                                    /: LLM call 1  (norm: composition)

    <- {initial draft}
        <= ::(generate an initial answer that satisfies the prompt)
        <- {prompt}                                    /: LLM call 2  (norm: composition)

    <- [all drafts]
        <= $. initialise the draft chain with the initial draft as its only element
        <- {initial draft}                             /: syntactic; no LLM call
```

```
/: ================================================================
/: MAIN LOOP  (self-seeding: each iteration appends to [all drafts],
/:            which is also the loop's input collection)
/: ================================================================

<- [reviewed drafts]
    <= *. for each prior draft in the chain, return its reviewed record:

        /: STEP 1 -- verify the prior draft against every synthesised checker
        <- [prior checks]
            <= $. run each callable in [verifiers] on {prior draft}; collect (passed, reason)
            <- {prior draft}                        /: norm: python_script; no LLM
            <- [verifiers]

        /: STEP 2 -- did the prior draft pass everything?
        <- <prior passed>
            <= <{ are all entries in [prior checks] marked passed? }>
            <- [prior checks]                       /: Judgement; LLM call

        /: STEP 3 -- should we stop?  (all-passed OR budget reached)
        <- <should stop>
            <= <{ <prior passed> is true OR [all drafts] already has 3 elements }>
            <- <prior passed>
            <- [all drafts]                         /: Judgement; LLM call

        /: STEP 4 -- bundle this iteration's record for the audit trail
        <- {reviewed draft}
            <= $. pair the prior draft with its check report
            <- {prior draft}
            <- [prior checks]                       /: syntactic; no LLM

        /: STEP 5 -- revise the prior draft  (gated: skipped when stopping)
        <- {new draft}
            <= ::(revise the prior draft to fix the failing constraints)
                @:! <should stop>                   /: Timing gate: fire iff NOT <should stop>
            <- {prompt}
            <- {prior draft}                        /: LLM call (only when gate fires)
            <- [prior checks]

        /: LOOP CONTROL -- conditionally append the new draft to [all drafts]
        <- [all drafts]
            <= $. append {new draft} to [all drafts] if not <should stop>
                @:! <should stop>                   /: same gate; loop ends when append is off
            <- {new draft}
            <- [all drafts]

    /: ============================================================
    /: LOOP BASE -- the collection iterated over (the chain itself)
    /: ============================================================

    <- [all drafts]                                 /: declared as a value sibling of the loop

<* {prior draft} from [all drafts]                 /: loop context: current element of the chain
```

Node inventory. *LLM-call nodes:* `[verifiers]`, `{initial draft}`, `<prior passed>` (per iter.), `<should stop>` (per iter.), `{new draft}` (per iter., only when gate fires). *Deterministic nodes:* `[prior checks]` (Python checker dispatch), `[all drafts]` initialise and append (both syntactic `$.`), `{reviewed draft}` (pair/bundle), `{compliant answer}` (last of chain).

Why the loop terminates. `[all drafts]` starts with one element (`{initial draft}`). Each iteration appends one `{new draft}` unless `<should stop>` fires. `<should stop>` fires either when the prior draft passes every checker or when `[all drafts]` already has 3 elements — so the loop runs at most 3 iterations regardless of LLM behaviour. Average revise rounds on n=209: 2.46 (Appendix L N.3).

Why `{new draft}` sees nothing else. The reviser node's declared siblings are exactly `{prompt}`, `{prior draft}`, and `[prior checks]` — the scope rule (§3.1) forbids it from reading earlier drafts, earlier checks, or `[verifiers]`. This is the bounded-input guarantee exhibited by artifact A2 (§H.4).

H.4 The bounded-input exhibit (illustrative artifact A2)

The reviser node `{new draft}` has **exactly three declared inputs**: `{prompt}`, `{prior draft}`, `[prior checks]`. It does not see earlier drafts in `[all drafts]`, earlier check tensors, or `[verifiers]` (which is in scope but not declared as an input to this particular Imperative). This is the scope-rule guarantee, visible structurally in `inference_repo.json`:

```
{
  "flow_index": "1.5.5",
  "node": "{new draft}",
  "paradigm": "imperative_generate_revision",
  "norm":     "composition",
  "gate":     {"timing": "@:!", "on": "<should stop>"},
  "declared_inputs": [
    {"name": "{prompt}",       "flow_index": "1.1",          "axis": "scalar"},
    {"name": "{prior draft}",  "flow_index": "1.5.<*>",      "axis": "scalar"},
    {"name": "[prior checks]", "flow_index": "1.5.1",        "axis": "verifier*"}
  ]
}
```

For comparison, the same revision step rendered as a Self-Refine (B3) accumulated trace contains the entire conversation history — initial prompt, every prior draft and critique, plus any scratchpad — with no per-step boundary an Operator could address. Artifact A2 in §6.4 is exactly this contrast.

H.5 The `.ncn` narrative exhibit (artifact A1)

Compiled deterministically from the plan above, before any LLM call:

> "(VALUE) compliant answer is the last draft from (VALUE) all drafts. To produce (VALUE) all drafts, the orchestrator first initialises the chain with (VALUE) initial draft, and then at each iteration — one per prior draft in the chain — conditionally appends (VALUE) new draft. To produce (VALUE) new draft, (ACTION) revises the prior draft to fix the failing constraints, given (VALUE) prompt, (VALUE) prior draft, and (VALUE) prior checks; this step is performed only if (VALUE) should stop is false. To decide (VALUE) should stop, (JUDGE) checks whether (VALUE) prior passed is true or (VALUE) all drafts already has 3 elements."

A Reviewer reading the `.ncn` can spot, without running anything: (i) that `[verifiers]` is synthesised once from the prompt and reused at every iteration; (ii) that the reviser sees only the immediately prior draft and its checks; (iii) that the loop terminates either on full compliance or on a 3-draft budget cap; (iv) that no LLM is invoked at the verification step (`python_script`).

H.6 Model, sampling, replication

`qwen-turbo-latest` on DashScope (OpenAI-compatible endpoint) across all conditions; temperature 0, single seed (where the API supports it). One run per (prompt, condition). Approximate total: $209 \times 5 \approx 1{,}045$ ground-truth-scored outputs, ~ 4 wall-clock hours for the four single-call baselines and ~ 3 wall-clock hours for the NC condition with 4 parallel workers (~ 211 s per prompt). Final scoring uses IFEval's shipped `evaluation_lib.test_instruction_following_strict` / `_loose` functions on the agent's final answer, not the agent's synthesised checkers. A second seed and paired-bootstrap CIs are deferred to the camera-ready.

H.7 Materials release

`prompts_subset.jsonl` (the 541-prompt subset with the NC-completed 209 at the front); the plan in three forms (`ifeval_plan.ncds`, post-formalised `*.pf.ncd`, compiled `concept_repo.json`/`inference_repo.json`); the provisions (three Markdown prompts plus shared paradigm files and deterministic Python scripts) under `provisions/v2_active/`; the harness (`main.py`, `conditions.py`, `nc_runner.py`, `scorer.py`, `analysis*.py`, `targeted_*.py`); per-(condition,

prompt) result JSONs plus aggregate summaries (`overlap_209_per_condition_summary.json`, `full_541_per_condition_summary.json`); three concrete sample cases in `samples/{NC_wins, all_pass, NC_loses}/` with each condition's `result.json` side-by-side and NC's full `plan_run.json` artifact trail (verifiers + all drafts + reviewed drafts). Three illustrative artifacts: `ncn_narrative.md` (A1), `bounded_input_snapshot.md` (A2), `reviser_intervention.md` (A3). Repository URL [`anonymized for review; provided on acceptance`].

I Failure Modes from Production Deployment

Honest accounting of what goes wrong when NormCode + Canvas is used at deployment scale. Each category corresponds to a structural property the medium does and does not guarantee.

I.1 Compiler Phase-1 failures (LLM-assisted Derivation)

The Derivation phase uses an LLM to extract concepts and dependencies from the `.ncds` source. The output is a formal plan that subsequent phases process deterministically, but it may not match the Designer's intent. Common shapes: hallucinated dependency, missed iteration (an outer-loop concept passed as a per-iteration concept or vice versa), wrong inference paradigm (a judgement classified as an imperative, losing the conditional structure). The Reviewer's `.ncn` read is the structural defence — if Phase 1 produced a wrong plan, the `.ncn` is the wrong narrative, and the Reviewer rejects. Recovery: Designer revises `.ncds`, recompiles, re-Reviews.

I.2 Reviewer-missed defects

Even with a correctly compiled `.ncn`, a Reviewer may approve a defective plan. Two failure shapes observed in production: **subtle scope errors** (e.g., a `style_guide` accidentally re-declared inside the per-slide loop rather than as a loop invariant, producing the right outputs on small inputs but failing at scale) and **mis-named concepts** (the `.ncn` reads plausibly but the concept name refers to a different upstream node than the Reviewer assumed). Mitigation: production deployment pairs Reviewer training with explicit "what could go wrong" checklists; structural diffs and change highlighting are open work (Limitations).

I.3 Operator misdiagnoses

The Operator may identify the wrong failing step. The most common cause is an upstream step producing valid-looking but semantically wrong output — e.g., a section-extraction step returns three plausible-looking sections, the slide-generation loop succeeds on all three, but a downstream consistency check rejects the whole report. The Operator's first inspection lands at the failing flow index, not the upstream root cause; correct diagnosis requires walking the bounded-input chain upward. P1 makes this walk $O(\text{depth})$, not $O(k)$, but it is not $O(1)$ when the root cause is several inferences upstream of the visible failure.

I.4 Environmental replication failures

A checkpoint is structurally self-contained but operationally dependent on the perceptual-sign targets (Appendix E). A forked run that references files moved or deleted after the original run will fail to resolve the signs and abort dispatch. Mitigation: deployment policy attaches a content hash to every perceptual sign at creation time; fork failures are explicit rather than silently corrupting the run.

I.5 Recovery time in production

A production histogram of (revision-to-recovery wall-clock) is included in the supplementary materials. The headline shape: most revisions complete in less than the cost of a single end-to-end run, which is the expected P3 economy; tail cases that require re-running upstream Derivation (when the revision changes the structural shape of the plan) recover at restart-from-scratch cost.

J Feature Comparison Table

NormCode + Canvas vs. the named competitors at a glance. Y = supports the property at the granularity a non-developer Reviewer/Operator/Reviser could recognise; partial = the underlying capability exists but is not exposed as a Reviewer-facing artifact; - = not provided.

Feature	NormCode	LangChain	LangGraph	AutoGen	PromptFlow	DSPy	Langflow
Pre-execution determinacy (P2)	Y	-	partial	-	partial	Y	-
Review-equals-runtime artifact (P2)	Y	-	-	-	-	-	-
Step-local addressing (P1)	Y	-	Y	-	Y	partial	Y
Explicit declared dependencies	Y	-	Y	-	Y	partial	Y
Auto stale-boundary on revision (P3)	Y	-	-	-	-	-	-
Plain-prose plan narrative (`.ncn`)	Y	-	-	-	-	-	-
Multi-agent per-step assignment	Y	manual	Y	Y	-	-	Y
Plan portability (text artifact)	Y	partial	partial	partial	Y	Y	-
Designer surface	Y	Y	Y	Y	Y	Y	Y
Reviewer surface (non-developer)	Y	-	-	-	-	-	-
Operator surface (failure-localization)	Y, $O(1)$	-	partial, $O(k)$	-	partial, $O(k)$	-	partial

Feature	NormCode	LangChain	LangGraph	AutoGen	PromptFlow	DSPy	Langflow
Reviser surface (override + scope-bounded resume)	Y	-	-	-	-	-	-

Methodology. Each cell was assessed by reading the system's published documentation and (where open source) inspecting its execution loop, asking three operational questions: *(i) is the artifact a non-developer Reviewer reads identical to what executes? (ii) are bounded inputs declared at each node and recorded at a stable address? (iii) can a correction at a node trigger re-execution of exactly the scope-downstream set?* Versions are pinned in `requirements.txt`; assessment date is the submission date. "Partial" means the capability exists at some level (often inside the orchestrator) but is not exposed as a Reviewer-facing artifact at the granularity the body's argument requires. A NormCode Imperative node's body may dispatch to any of these frameworks as a leaf capability (the contribution is compositional with the stack, not adversarial).

K Reproducibility Checklist

Software. NormCode language, compiler, and orchestrator are pinned in `requirements.txt`; the Canvas frontend in `package.json` (Python 3.11; Node 20; React 18; FastAPI 0.110; SQLite 3.45). The IFEval evaluation harness and Self-Refine baseline scaffold are also pinned there. Exact version numbers are released alongside de-anonymisation.

Model and inference. A frontier instruction-tuned LLM at a single fixed checkpoint is used across all four conditions (B1, B2, B3, NC); differences in IFEval pass rate are therefore attributable to surrounding structure, not the generator. Decoding parameters: temperature 0.7 for drafting/revision, 0.0 for the constraint-extraction node; top-p 1.0; max-tokens 4096. The temperature-0 extraction is critical: the extractor must produce the same `[verifier specs]` across runs of a given prompt so that the verification signal is comparable.

Seeds, hardware, budget. Seeds: `42` (100-prompt headline sampler), `123` (30-prompt transparency sampler), `7` (bootstrap CI). Hardware: commodity laptop or server (Apple M2 Pro / AMD EPYC 7763); 16 GB RAM minimum; no GPU required (API-based inference); ∼5 GB disk. Budget: $\approx 100 \times 4 \times 3 \times 2 = 1{,}800$ LLM calls, 3–4 wall-clock hours, low-tens of USD on cheap models. Direct local energy is below 1 kWh; API-side energy depends on the provider's per-token estimate.

Data and code. IFEval is distributed under its published licence and included for offline reference at `refs/ifeval/`. NormCode + Canvas source: Apache 2.0. Production plan `.ncds` files: CC-BY-4.0. Repository URL: `[anonymized for review; provided on acceptance]`. Install: `pip install -r requirements.txt && cd canvas-frontend && npm install`. Reproduce the headline: `python experiments/run_ifeval.py -subset 100 -runs 3`.

Partial reproducibility without API access. The compiler self-hosting sanity check (Appendix F, NC Compilations) requires no LLM and runs from the code release alone; the deterministic verification half of the IFEval plan (the Python checker calls in `[round k checks]`) runs against any draft input without further LLM access; the bounded-input snapshot exhibit (artifact A2) is a JSON dump and requires no execution.

Anonymisation and pre-registration. For the ARR review version: author names, affiliations, contact emails, and deployment specifics are redacted; repository URLs are replaced with `[anonymized]`. The experimental design — subset stratification, four-condition structure, NormCode plan structure, scoring

metrics, statistical treatment — is pre-registered in the supplementary materials; any deviations are noted in §6.

L Experimental Results (IFEval)

This appendix reports the numerical outcomes of the IFEval constructive demonstration whose protocol is given in Appendix H and whose framing is given in §6. The numbers reported here are from a partial run — the 209-prompt hard subset of the 541-prompt benchmark — on `qwen-turbo-latest` (DashScope, OpenAI-compatible) at temperature 0, single seed. *The full 541-prompt run is still in flight at submission time; the remaining 332 prompts are predominantly single-constraint and will be added in the camera-ready. The headline ordering and finding directions are not expected to change.*

N.1 Headline metrics (n=209 hard subset)

All five conditions evaluated on the same 209 prompts (all 57 three-constraint prompts plus 152 of the 179 two-constraint prompts; the 305 single-constraint prompts are omitted from this subset and reported in N.5 below). IFEval's four published metrics; final scoring uses IFEval's shipped checkers, not the agent's synthesised ones (§6.2).

Condition	Pr-Strict	Pr-Loose	In-Strict	In-Loose
B2 (constraint-aware)	**0.807**	**0.877**	**0.911**	**0.943**
NC (K=3, NormCode plan)	0.801	0.854	0.901	0.930
B1 (single-shot)	0.789	0.854	0.901	0.932
B3 (Self-Refine K=3)	0.725	0.778	0.869	0.893
B_COT (chain-of-thought)	0.708	0.766	0.867	0.893

Aggregate ordering (prompt-strict). $\mathrm{B2} \approx \mathrm{NC} > \mathrm{B1} > \mathrm{B3} > \mathrm{B_COT}$. The B2–NC gap is 0.6 pp on n=209 and lies within sampling noise on a single seed; we read it as a tie. The structurally interesting orderings are *below* the top: pure single-shot (B1) beats self-refine (B3) by 6.4 pp, and chain-of-thought (B_COT) is the worst of the five.

N.2 Per-instruction-type breakdown (strict, $n = 209$)

The 22 instruction types appearing in the 209-prompt subset. The two diagnostic rows (the NC win on `length_constraints:number_words` and the NC loss on `combination:repeat_prompt`) are bolded.

Instruction type	NC (cnt)	B1	B2	B3	B_COT
length_constraints:number_words	**.97** (33/34)	.71	.62	.76	.50
length_constraints:number_paragraphs	.91 (10/11)	.91	.91	.73	.73
length_constraints:nth_paragraph_first_word	.75 (3/4)	.75	.75	.75	.75
keywords:existence	.96 (25/26)	.96	1.00	1.00	1.00
keywords:forbidden_words	.91 (21/23)	.87	.87	.87	.83
keywords:frequency	.88 (21/24)	.83	.88	.92	.92
keywords:letter_frequency	.85 (17/20)	.75	.80	.75	.85
detectable_format:title	1.00 (18/18)	1.00	1.00	1.00	1.00
detectable_format:json_format	1.00 (7/7)	1.00	1.00	1.00	1.00
detectable_format:number_bullet_lists	.93 (13/14)	.86	.86	.86	.93
detectable_format:number_highlighted_sections	.91 (20/22)	.95	1.00	.95	1.00
detectable_format:multiple_sections	1.00 (7/7)	1.00	1.00	1.00	1.00
detectable_content:number_placeholders	.93 (13/14)	1.00	1.00	1.00	1.00
detectable_content:postscript	1.00 (11/11)	1.00	1.00	1.00	1.00
change_case:english_capital	1.00 (10/10)	.90	.80	.80	1.00

Instruction type	NC (cnt)	B1	B2	B3	B_COT
change_case:english_lowercase	.76 (16/21)	.90	.95	.95	.81
combination:repeat_prompt	**.50** (11/22)	.91	**1.00**	.14	.36
combination:two_responses	.86 (12/14)	.86	.93	.93	1.00
punctuation:no_comma	.95 (41/43)	.93	.93	.98	1.00
startend:end_checker	.83 (5/6)	1.00	1.00	1.00	.83
startend:quotation	.95 (19/20)	1.00	1.00	.95	.95
language:response_language	1.00 (12/12)	1.00	1.00	1.00	1.00

N.3 Compute cost ($n = 209$)

Condition	avg wall (s)	avg LLM calls	avg tokens	revise rounds
B1	3.2	1.0	383	0.00
B2	3.2	1.0	385	0.00
B_COT	8.9	1.0	948	0.00
B3 (K=3)	16.8	2.0	2,387	1.01
NC (K=3)	210.7	37.5	192,245	2.46

At tied prompt-strict accuracy (NC vs B1/B2), NC consumes $\sim 65\times$ B2's wall-clock, $\sim 37\times$ its LLM calls, and $\sim 500\times$ its tokens. The 37.5 call count is *not* 37 semantic revisions: the dominant factor is the current orchestrator's composition pipeline, which makes ~ 10 inner LLM calls per "semantic" inference for prompt assembly. The matched-compute axis the paper's hypothesis cares about is `revise_rounds` (2.46 for NC, 1.01 for B3, 0 for the single-shots), not raw `calls` or `tokens`. A leaner runtime (or direct API compilation) should approach B3's call count for the same semantic work.

N.4 Where NC wins / loses, and the B3 finding

NC wins by $\geq$ 5 pp: `length_constraints:number_words` (0.97 vs best baseline B3 0.76, $+21$ **pp vs matched-compute B3**); `change_case:english_capital` (1.00 vs B3 0.80); `keywords:forbidden_words` (0.91 vs 0.87). NC's revision loop structurally helps where the model can miscount and the checker provides a measurable gradient.

NC loses by $\geq$ 5 pp: `combination:repeat_prompt` (0.50 vs B2 1.00, -50 pp — the revision loop disrupts the verbatim-structural pattern of emitting the original prompt before answering); `change_case:english_lowercase` (0.76 vs B2/B3 0.95 — the synthesised checker over-triggers on `\n`-separated sections); `startend:end_checker` (0.83 vs 1.00). These losses are addressable by editing the per-step prompts the plan calls (the *provisions*) without changing the plan itself; that constitutes the P3 exhibit at provision granularity.

B3 isolates iteration without checkers. B3 averages 2.0 LLM calls per prompt (1 initial + 1 critique-revise turn) and 1.01 average revise rounds. Comparing B1 (0.789 prompt-strict, no refinement) to B3 (0.725, same model and chat thread) gives $-6.4\,pp$. Two failure modes recur: the critique step rewrites previously-passing content, and the model "agrees" with imagined failures it then "fixes", breaking a previously-passing constraint. The B3 result is a clean control: the $+21$ pp NC$-$B3 gap on `length_constraints:number_words` therefore isolates the contribution of NC's deterministic-checker-driven revision specifically, not iteration in general.

N.5 Full-541 partial results (4 baselines)

On the full 541 prompts the four single-LLM-call baselines have run to completion; NC is still in flight (the 332 remaining prompts are predominantly single-constraint and will be added in the camera-ready). The B2 > B1 > B_COT > B3 ordering is preserved on n=541 (B2 0.872, B1 0.864, B_COT 0.832, B3 0.826 prompt-strict); aggregate accuracies are ~ 7 pp higher than on the n=209 hard subset because single-constraint prompts are easier across the board. NC's relative advantage on hard constraint families

is therefore likely *understated* once normalised to $n{=}541$, since the easier constraints are overrepresented there.

N.6 Caveats

- **Subset bias.** The $n{=}209$ is the front of a stratified order (hardest prompts first). Accuracies here are lower than what the full $n{=}541$ will report. NC's relative win on `length_constraints:number_words` is a stable family-level pattern, not a subset artefact; the `combination:repeat_prompt` loss is also stable.
- **Single model / single seed / single temperature.** `qwen-turbo-latest`, $T{=}0$, single seed. Generalisation to other models (`qwen-plus`, Claude Haiku, GPT-4o) is explicitly not claimed; it is the most urgent follow-up.
- **Verifier synthesis quality.** The LLM-synthesised checkers occasionally include non-ASCII characters in comments/string literals that crash `exec()` on load; this affects ~ 1 check per prompt on average and does not propagate to the other checks. The headline accuracy uses IFEval's shipped checkers, so verifier quality affects what NC *tries to fix*, not what we *count as passed*.
- **Runtime overhead.** The ~ 37.5 LLM-call count per NC prompt is dominated by the current orchestrator's composition pipeline (~ 10 inner calls per semantic inference for prompt assembly), not by 37 semantic revisions. The semantic-budget axis is `revise_rounds` (2.46), and at that budget NC is comparable to B3.
- **No statistical tests.** Single-seed numbers are reported as point estimates. Paired-bootstrap CIs and Wilcoxon signed-rank on per-prompt deltas will be added once a second seed completes.

N.7 What the artifacts give (independent of headline)

Each NC run emits, alongside the final answer, three structural artifacts (§6.4, sample files in the supplementary package): `verifiers` — the synthesised Python checkers a reviewer can read before any draft is generated (P2); `all_drafts` — the full chain of drafts addressable per-iteration (P1); and `reviewed_drafts` — for each draft, which checker passed/failed with what reason (basis for the scope-bounded re-execution of P3). B1, B2, and B_COT produce only a single final string; B3 produces a chat transcript without per-constraint attribution. This contrast is what the cost in N.3 buys.