Functional Safety for Language Models

Anonymous authors
Paper under double-blind review

Abstract

Large language models (LLMs) are increasingly used for document and code editing, yet standard approaches typically lack formal assurances about the scope, structure, or side effects of their modifications. We introduce *Functional Safety*, a hierarchy-aware editing architecture that formalizes LLM-driven edits as typed plans over explicit hierarchies with deterministic execution. A stochastic planning stage operates on an explicit hierarchical representation and emits a structured plan of typed operations that separate structural reorganization from bounded content generation. We analyze each step with two footprints: a *structural footprint* (nodes whose relations may change) and a *payload footprint* (nodes whose local content may change), and the guarantees are scoped per step. Execution is performed by a deterministic, structure-constrained component that enforces locality, guards protected regions, preserves byte-for-byte payload outside each step's payload footprint, and confines structural changes to each step's structural footprint under the stated assumptions. We formalize the architecture, specify its invariants, and prove Deterministic Safety and Conditional Functional Safety theorems that bound side effects under those assumptions. Empirical evaluations on long-form document rewriting, code refactoring, and multi-page policy briefs show that Functional Safety substantially reduces side effects relative to ReAct-style tool agents reflecting current agentic editor practice. These results demonstrate that principles from functional programming—explicit structure, composability, and controlled side effects—provide a rigorous foundation for reliable LLM-driven editing.

1 Introduction

Large language models (LLMs) have demonstrated impressive fluency and task coverage, yet their behavior on structured editing tasks remains fragile. Human edit intent is typically local and specific, while the remainder of the artifact is expected to remain invariant. By contrast, LLMs operate over flat token sequences where structure is only implicit, so locality is inferred rather than enforced and formal safety guarantees are absent. Even when prompted to make narrowly scoped changes, current systems typically operate directly in token space, regenerating long spans with no explicit notion of hierarchy or change semantics. As a result, even minimal edits can induce structural drift and side effects outside the intended scope.

These failures are more than minor inconveniences. In settings such as legal drafting, scientific writing, and code maintenance, users often need strong assurances that an edit request will not silently alter definitions, assumptions, or program behavior outside the requested region. Two factors are entangled in current systems: open-ended generative decoding, which can re-sample content even for localized requests, and flat token-level representations, which provide no explicit boundaries for where changes are permitted. Together they make it difficult to bound edits or reason about their impact.

From an autoregressive model's perspective, a paragraph, a function, or a theorem is merely a contiguous sequence of tokens. The model typically infers boundaries and structure implicitly, and therefore does not explicitly enforce invariants such as preservation of protected regions, structural well-formedness, or strict edit locality. In domains where precision is essential, such implicit reasoning is insufficient. The central problem is the absence of a principled mechanism for enforcing edit locality and explicit structural constraints during transformation.

This paper describes *Functional Safety*, a hierarchy-aware architecture intended to provide stronger behavioral properties for LLM-driven editing. At a high level, the approach factors editing into two primary phases within a multi-stage pipeline. A stochastic planning stage works over an explicit hierarchical representation of the document or codebase, selecting which units to modify and how. A separate, functionally pure executor then applies these plans using deterministic tree transformations that respect invariants about protected regions, structure, and layout. By localizing randomness to the planning stage and treating execution as a pure function, the architecture makes it possible to reason about side effects in a way that is largely model-agnostic. We make this precise by distinguishing a per-step *payload footprint* (local content that may change) from a per-step *structural footprint* (nodes whose relations may change); plan-level unions summarize aggregate impact.

More broadly, the framework integrates insights from functional programming and hierarchical models of human cognition. Functional languages emphasize purity, composability, and explicit control of side effects; hierarchical cognitive theories emphasize chunking, multi-level representation, and compositional operations over symbols. Functional Safety combines these ideas to obtain a simple but expressive abstraction for LLM-mediated edits: plans are stochastic, but their interpretation is deterministic and structure-constrained. Although errors cannot be eliminated, they can be confined to isolated stochastic components and bounded in scope, enabling conditional safety guarantees.

1.1 Contributions

This paper makes four contributions toward a principled notion of functional safety in language models:

- **Functional Safety procedure.** We introduce a hierarchy-aware editing procedure that separates *planning* from *execution*. Documents and programs are represented as graphs of semantic units (sections, paragraphs, methods, blocks). A stochastic planning stage operates over this hierarchy to propose local edits, while a deterministic executor applies those edits using pure tree transformations. Stochasticity enters the system through two channels: the planning stage, which selects semantic units and proposes edits, and worker modules, which may call language models to generate content within plan-selected regions. In this formulation, both sources of stochasticity are confined to the targeted nodes' payload. A deterministic executor applies structural checks and merges results, confining model variability to each step's payload footprint while constraining structural changes to each step's structural footprint (with plan-level unions summarizing aggregate impact).

- **Verified semantics.** We provide a formal model for the hierarchical representation, the planning function, and the executor. Within this model we state and prove simple theorems that bound side effects by planning-stage reliability: if the planning stage respects a small set of interface contracts and worker outputs are confined to the plan-selected nodes, the executor does not modify protected regions' content outside each step's payload footprint and does not introduce structural drift outside each step's structural footprint in the formal model. This yields a notion of functional safety that is independent of any particular language model.

- **Empirical coverage across domains.** We instantiate the architecture in a practical system and evaluate it on large-document editing and non-trivial code refactoring tasks, comparing against a ReAct-style tool agent that represents current agentic editor practice (context-matching patches via an observe–think–act loop). Functional Safety achieves 100% strict success with zero side effects across all single-document tasks, whereas the ReAct agent succeeds on simple localized edits but degrades on multi-location changes and fails on structural operations. The comparison demonstrates that hierarchy-aware planning and deterministic execution provide reliability that tool-based editing without explicit structure cannot match.

- **Bridge to emerging literature.** Finally, we relate Functional Safety to work on long-context transformers, hierarchical cognition, tool-augmented language models, and functional programming. We argue that separating stochastic planning from deterministic execution provides a common language for these lines of research and outline how the proposed framework can serve as a foundation for a broader research program on side-effect-aware LLM systems.

1.2 Background

The limitations of token-level reasoning for structural editing have been widely recognized. Standard approaches treat documents as flat sequences of subword tokens and apply autoregressive generation to entire spans, even when the requested change is local. Without an explicit representation of sections, paragraphs, code blocks, or method boundaries, the model has no way to distinguish between regions that may be altered and regions intended to remain invariant. This mismatch between sequence-level operations and structured documents accounts for the unintended side effects observed in practice.

Human cognition, by contrast, relies heavily on hierarchical abstraction when manipulating complex artifacts such as proofs, programs, or manuscripts. People reason in terms of chapters, sections, lemmas, functions, and blocks; they perform local edits at one level of the hierarchy while keeping higher-level structure fixed. This style of reasoning supports both global coherence and local flexibility, and it suggests that reliable machine editing may call for explicit representations of semantic units rather than raw token streams.

Several strands of machine learning research move in this direction. Work on long-context models and efficient attention mechanisms extends the span of text that can be processed in a single forward pass, but typically leaves the underlying representation flat and token-based. ReAct-style tool agents (Yao et al., 2023) represent the current state of practice for agentic editing: an LLM observes document state, reasons about what to change, and applies patches or span edits through tool calls. This approach improves on flat generation for simple localized edits, but still operates over linear text without explicit hierarchy, leaving structural operations and multi-location edits unreliable. Other work explores constrained decoding and programmatic scaffolding to shape model outputs, yet often lacks a formally specified execution layer that isolates and controls side effects. Existing code-refactoring tools, meanwhile, provide strong assurances but are tailored to specific languages and do not leverage LLMs for high-level planning.

The approach developed in this paper builds on a different combination of ideas. We assume that language models are powerful but noisy planners over latent structure, and we place a deterministic, functionally specified executor between their plans and the underlying document. By making the hierarchical representation explicit and the executor pure, we obtain a system in which structural invariants can be stated and proven once, then reused across models and domains. Functional Safety thus aims to improve not only performance but also the correctness, predictability, and safety of LLM-driven editing.

2 Formal Model

In this section we present a mathematical framework for the hierarchy-aware editing architecture. Our goal is to formalize the core components—hierarchical representations, the router and planner, stochastic worker modules, and the deterministic executor—in a way that makes their roles, constraints, and interactions explicit. This formalization isolates the sources of stochasticity, specifies the symbolic structures on which plans operate, and establishes the invariants used for reliable, content-side-effect-bounded transformations. This is one formalization of the approach; alternative formulations or operator sets are also possible.

System Components. The architecture consists of four distinct components:

- **Router** g: a lightweight model that maps a user request and node summaries to either (i) a typed plan P in the edit algebra $\mathcal{O}$ or (ii) a delegate directive for a permutation-only symbolic planner. In this formalization, the router does not itself perform structural edits.

- **Planner** f_θ: a stochastic LLM-based symbolic planner invoked when the router delegates a pure permutation; it emits a section ordering (or equivalent plan fragment) that the host expands into a Move.

- **Workers** $\{W_j\}$: optional stochastic LLM modules that perform bounded local transformations on extracted spans. In this formalization, a worker is not given global structure and its scope is restricted to the node selected by the plan.

- **Executor** E: a deterministic executor for symbolic plans. The executor applies operators one by one and invokes workers within bounded spans *when replacement content is not provided by the plan.* Worker calls occur during execution under the executor's control. The executor enforces structural invariants and, under the model assumptions, establishes payload preservation outside each step's payload footprint (and thus outside the plan-level union).

Stochasticity may arise in the router, the delegated planner (when used), and the workers, but all stochastic effects are confined to plan-selected regions. The executor itself is pure and deterministic.

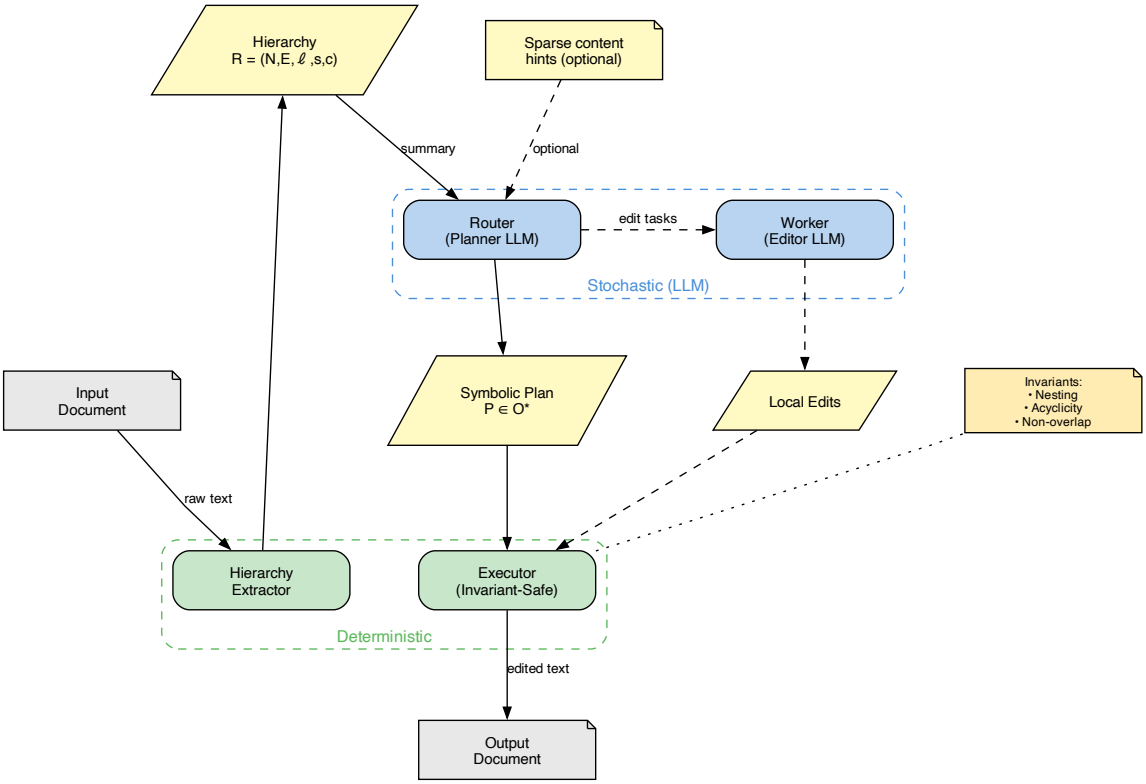

Figure 1: System architecture separating stochastic LLM components (Router/Planner, Workers) from deterministic components (Extractor, Executor). The symbolic plan P serves as the interface between the two domains, enabling formal reasoning about execution behavior.

2.1 Raw Input and Token Sequence

Let the raw input document or code corpus be a token sequence

$$X = (x_1, x_2, \ldots, x_T).$$

Beyond initial parsing, the architecture does not operate directly on X. For a consistent extraction, the hierarchy R satisfies $\mathrm{render}(R) = C(X)$, where C is a canonicalization operator that is identity when X already respects the paragraph-separator convention described below.

2.2 Hierarchical Representation

A hierarchical representation R can be described both (i) compositionally as a tuple and (ii) declaratively through its constituent components.

Tuple Form.

$$R = (N, E, \ell, s, p),$$

Component Form.

- N: a finite set of nodes corresponding to semantic units (paragraphs, sections, functions, etc.).
- $E \subseteq N \times N$: directed edges defining parent–child relations.
- $\ell : N \to \mathcal{L}$: labeling function assigning each node a semantic type.
- $s : N \to [1, T] \times [1, T]$: span function mapping each node to an interval (i, j) in the rendered document.
- $p : N \to X^*$: payload function storing the *local* content of each node (text owned by the node outside its child spans, e.g., a paragraph's text, heading tokens, or structural separators).

Paragraph boundary convention. For prose formats, paragraph boundaries are defined by blank lines. The canonicalization $C(X)$ enforces at least one blank line between adjacent paragraph siblings so boundaries survive structural moves; if the raw input omits a separator (e.g., the final paragraph in a document), C inserts the minimal blank-line separator. Inter-paragraph whitespace is represented as explicit gap nodes; only the minimal separator is guaranteed to be preserved across moves, while extra whitespace can remain anchored between the original siblings. Separator insertion is applied only when a structural edit would otherwise collapse paragraph boundaries. When X already contains these separators, C is identity and the rendering is byte-identical to the original input (Appendix D.1).

Rendered Content. We define a deterministic render function

$$\text{render}_R(n) = p(n) \parallel \text{render}_R(c_1) \parallel \cdots \parallel \text{render}_R(c_k),$$

where $(c_1, \ldots, c_k)$ are the ordered children of n and $\parallel$ denotes concatenation (including any format-specific separators encoded in payload or node metadata). The document text is $\text{render}(R) = \text{render}_R(\text{root})$. Spans are derived by rendering: $s(n)$ is the interval occupied by $\text{render}_R(n)$ inside $\text{render}(R)$. Rendered content can change when children move even if a node's local payload does not, which is why non-interference is stated in terms of $p(n)$ below.

Well-Formedness Constraints.

- **Nesting**: Parent spans strictly contain child spans.
- **Acyclicity**: (N, E) contains no directed cycles.
- **Non-overlap**: Nodes not in an ancestor–descendant relation have disjoint spans.

Identity and Reconciliation. Each node $n \in N$ carries a stable entity identifier $e(n)$ that persists across edits and moves, and a content identifier derived from its local payload. When a refresh re-extracts a hierarchy from text, the executor computes a reconciliation map χ from prior nodes to refreshed nodes. The reconciliation is *structure-first*: parent/position anchors and spans are primary signals, and payload similarity is used only as a tie-breaker. Outside the allowed payload footprint, χ must be total and unambiguous; if any node outside the footprint cannot be matched uniquely, execution fails (fail-closed). Inside the footprint, unmatched nodes are permitted and receive fresh entity identifiers. Merge and split events create new entity

identifiers and record lineage edges so continuity is preserved without reusing prior identities. The formal model assumes strict payload identity outside the payload footprint, with canonical paragraph separators represented by C as gap nodes; optional executor policies that allow whitespace-normalized matching for soft adjacency nodes are implementation variants and are not part of the core theorem unless explicitly specified.

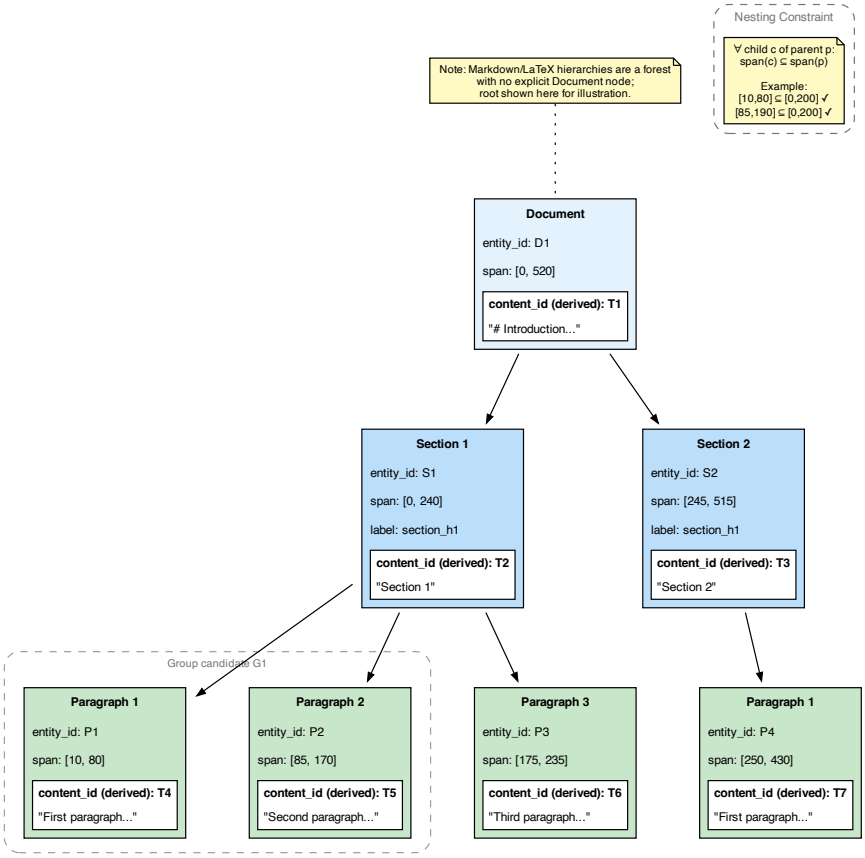

Figure 2: Example hierarchical representation R showing a document with two sections and three paragraphs. Each node stores a span indicating its position in the source text. The nesting constraint assumes child spans are contained within parent spans (e.g., $[10, 80] \subseteq [0, 200]$).

2.3 Edit Algebra Formalization

Let $S = (s_1, \ldots, s_m)$ be the top-level nodes (e.g., sections). The edit algebra is:

$$\mathcal{O} = \{\textsc{Move}, \textsc{Edit}, \textsc{Insert}, \textsc{Delete}, \textsc{Annotate}\}.$$

We use $\textsc{Move}$ for deterministic section permutations ($\textsc{Move}(\pi)$) and subtree relocation ($\textsc{Move}(n, \text{pos})$); "reorder" tasks are instances of $\textsc{Move}(\pi)$.

Operator Semantics.

- $\textsc{Move}(\pi)$: π is a permutation of $(s_1, \ldots, s_m)$. The executor deterministically permutes section buffers; no worker is invoked.

- $\textsc{Move}(n, \text{pos})$: relocate the subtree rooted at node n to position pos (parent/index or anchor before/after). The executor performs a deterministic structural move and preserves payload content byte-for-byte; paragraph separators are handled by the separator policy.
- $\textsc{Edit}(s_i, r, \theta)$: s_i is a node identifier, r is a natural-language instruction, and θ is an optional crop (α, β). If θ is provided, the worker receives the cropped substring of $\text{render}_R(s_i)$ and r; the executor splices the result back into s_i. If the plan supplies replacement content directly, the worker can be bypassed and the executor splices the provided content. If θ is empty, the worker (or the supplied content) replaces the rendered content of that node; if s_i has children, the executor re-extracts the subtree rooted at s_i after the replacement to rebuild its internal structure and payloads.
- $\textsc{Insert}(\text{content}, \ell, \text{pos})$: creates a new node with payload content and label $\ell \in \mathcal{L}$ at position pos. The position specifier pos can indicate: (i) a parent node and child index, (ii) an anchor sibling with before/after placement, or (iii) a root-level index. The new node is assigned a fresh identifier $n_{\text{new}} \notin N$.
- $\textsc{Delete}(n)$: removes node $n \in N$ and all of its descendants from the hierarchy. Parent–child links are updated to excise the deleted subtree.
- $\textsc{Annotate}(n, \ell)$: update node n's label/metadata to ℓ without changing its payload (e.g., relabel a heading as a sub-heading in the hierarchy). If rendering in a format depends on the label token itself, this operation is paired with a corresponding Edit.

Insert and Delete. The Insert and Delete operators are structural primitives that modify the node set N. Insert adds a new node at a specified position within the hierarchy—either as a child of an existing node, as a sibling relative to an anchor node, or at the root level. The content of the inserted node may be provided directly by the plan or authored by a worker LLM when the plan specifies a reason and read-only context; in either case, the structural placement is deterministic. Delete removes a node and its entire subtree, updating parent references to maintain well-formedness. Together, these operators enable the planner to express additive and subtractive structural edits, complementing the content-modifying Edit and the purely structural Move.

2.4 Plan Validity and Typing Rules

A plan is a finite sequence $P = (o_1, \ldots, o_k)$ with $o_i \in \mathcal{O}$. The special directive DelegatePermutation is also permitted. Here $\mathcal{O} = \{\textsc{Move}, \textsc{Edit}, \textsc{Insert}, \textsc{Delete}, \textsc{Annotate}\}$. Let $\mathcal{P}$ denote the set of well-typed symbolic plans over $\mathcal{O}$, and let $\mathcal{R}$ denote the set of well-formed hierarchies. Typing judgments take the form

$$R \vdash o_i : R \to R_i,$$

with $R_0 = R$, and extend compositionally to $R \vdash P : R \to R_k$.

A symbolic plan $P = (o_1, \ldots, o_k)$ is *well-typed* for hierarchy $R = (N, E, \ell, s, p)$ with top-level nodes $S = (s_1, \ldots, s_m)$ iff each operator o_i satisfies the corresponding typing rule:

(a) $\textsc{Move}(\pi)$ operator is well-typed for $S = (s_1, \ldots, s_m)$ iff the multiset of elements in π equals the multiset of elements in S:

$$\{\{\pi_1, \ldots, \pi_m\}\} = \{\{s_1, \ldots, s_m\}\}.$$

Equivalently, the map $i \mapsto \pi_i$ defines a bijection from $\{1, \ldots, m\}$ onto S.

(b) $\textsc{Move}(n, \text{pos})$ is well-typed iff

- $n \in N$ is a valid node,
- pos specifies a valid position (parent exists, anchor exists, or root-level index is valid),
- if pos is parent-based, then the parent is not in the subtree of n,
- if pos is anchor-based, then the anchor is not in the subtree of n and the parent inferred from the anchor is valid.

(c) $\textsc{Edit}(s_i, r, \theta)$ is well-typed iff

- s_i is a valid node (i.e. $s_i \in N$),
- r is a natural-language instruction when worker generation is used; if the plan supplies concrete replacement content, r may be empty and the worker is bypassed, and
- if a crop $\theta = (\alpha, \beta)$ is supplied, then

$$(\alpha, \beta) \subseteq s(s_i),$$

where $s(s_i)$ denotes the token-span interval associated with node s_i.

(d) $\textsc{Insert}(\text{content}, \ell, \text{pos})$ is well-typed iff

- $\ell \in \mathcal{L}$ is a valid semantic label,
- content is a (possibly empty) string, and
- pos specifies a valid position:
  - if $\text{pos} = (\text{parent_id}, \text{index})$, then $\text{parent_id} \in N$ and $0 \leq \text{index} \leq |\text{children}(\text{parent_id})|$,
  - if $\text{pos} = (\text{anchor_id}, \text{rel})$ with $\text{rel} \in \{\texttt{before}, \texttt{after}\}$, then $\text{anchor_id} \in N$,
  - if $\text{pos} = (\texttt{root}, \text{index})$, then $0 \leq \text{index} \leq m$.

(e) $\textsc{Delete}(n)$ is well-typed iff $n \in N$.

(f) $\textsc{Annotate}(n, \ell)$ is well-typed iff $n \in N$ and $\ell \in \mathcal{L}$.

(g) DelegatePermutation is well-typed iff it expands uniquely to a valid Move. Formally, the plan supplies a permutation π of S, yielding the operator $\textsc{Move}(\pi)$ which satisfies rule (a).

2.5 Executor Semantics

The executor applies the operator semantics defined in Section 2.3 deterministically and in sequence, invoking workers only where the plan requires generation and enforcing the structural constraints described below.

In the router–hybrid pipeline, $\textsc{Move}(n, \text{pos})$ is used for section and paragraph relocation; paragraph moves require paragraph anchors or a parent section, while $\textsc{Move}(\pi)$ remains the dedicated section permutation operator.

The executor checks nesting, acyclicity, non-overlap, and payload non-interference after each operation. When span alignment is required, it refreshes the hierarchy deterministically, and when an Edit can change structure, it re-extracts the affected subtree.

Unified Position Specifiers. The executor uses a unified positioning model for both Insert and internal move operations. A position specifier pos can take three forms:

1. *Index-based*: $(\text{parent_id}, k)$ places the node as the k-th child of node parent_id.

2. *Anchor-based*: $(\text{anchor_id}, \text{rel})$ with $\text{rel} \in \{\texttt{before}, \texttt{after}\}$ places the node immediately before or after the anchor sibling. The target parent is inferred from the anchor's parent.

3. *Root-level*: $(\texttt{root}, k)$ places the node as the k-th top-level node.

This unified model allows planners to express both absolute positioning ("make this the third paragraph of section 2") and relative positioning ("insert this after the introduction paragraph") using the same underlying mechanism.

2.6 Stochastic Planning Function

The planning stage is stochastic; we can model it as a function

$$P = f_\theta(X, R, \omega),$$

where ω captures all randomness. In this formalism, operations refer to symbolic node identifiers. In the router–hybrid implementation, the router emits a plan P directly or a delegate directive; when delegation occurs, a symbolic planner provides a permutation that is expanded into a MOVE and spliced into the plan.

The planning stage is responsible for emitting a well-formed symbolic plan in the edit algebra. In practice, this entails the underlying model producing syntactically valid operations, but questions of empirical reliability are addressed in the experimental section.

2.7 Deterministic Executor

Given a hierarchical representation R and a symbolic edit plan $P = (o_1, \ldots, o_k)$, the executor applies the operators in sequence to produce a new hierarchy,

$$R' = E(R, P),$$

where E is a total deterministic function

$$E : \mathcal{R} \times \mathcal{P} \to \mathcal{R}.$$

That is, once the initial hierarchy R and the plan P are fixed, the executor's behavior is fully determined: there is exactly one resulting hierarchy R'. Although individual worker modules may be stochastic—returning different candidate outputs when invoked—their effects are confined to the specific spans referenced in the plan. In this executor, worker output is not allowed to alter any other portion of the document's payload; structural metadata (spans and ordering) may be recomputed deterministically as required by structural edits.

Thus, for fixed R and P, the global structure of R' and all payload outside the planner-selected regions are uniquely determined, independent of any stochasticity in worker calls. The executor therefore functions as a pure, structure-constrained interpreter for symbolic plans, ensuring that all content side effects remain local, bounded, and auditable.

3 Hierarchy-Aware Functional Safety

Functional Safety separates planning from execution. The host first builds an explicit hierarchy over the input, a router emits typed edit operations over that hierarchy, and a deterministic executor applies those operations while enforcing locality and structural validity. This section summarizes the operational flow corresponding to the formal model in Section 2.

3.1 Pass I: Host-Grounded Hierarchy Extraction

The host extracts a hierarchy before planning or worker calls. For structured formats (e.g., Markdown, LaTeX, and code), extraction is deterministic; for weakly structured inputs, hierarchy induction can be learned or heuristic. The extracted hierarchy is the authoritative representation for subsequent planning and execution: nodes carry local payloads and ordered children, and rendering the hierarchy recovers the document under lossless canonicalization.

To keep planning context compact, the host provides router-facing descriptors (IDs, titles, and short previews) rather than full node payloads. For tasks that require local structural detail (for example, edits inside a table or list), the executor may construct temporary local views and apply the edit within that scoped view before reintegrating it into the main hierarchy.

3.2 Pass II: Router-Based Symbolic Planning

The router receives the user instruction plus a compact hierarchy outline and emits a typed plan over five operators:

- MOVE: reorder or relocate existing nodes.
- EDIT: modify content in a selected node (optionally on a cropped span).
- INSERT: add a new node at a validated position.
- DELETE: remove a node and its descendants.
- ANNOTATE: relabel a node (for example, heading-to-subheading) without expanding write scope.

The router may also delegate pure permutations to a symbolic permutation routine. Context units are read-only: they can inform generation but do not expand the writable scope of an operation.

Router/Planner Decomposition. Figure 1 presents a unified view for readability. Conceptually, routing and delegated symbolic planning are distinct interfaces: routing chooses typed operations, while delegated planning solves permutation-only structure edits.

3.3 Pass III: Deterministic Execution

The executor validates each operation, computes step-wise footprints, and applies edits deterministically. MOVE, INSERT, DELETE, and ANNOTATE are interpreted directly by the executor. EDIT may invoke a worker model, but the resulting write is confined to the targeted scope and then reconciled against the hierarchy.

After each step, the executor checks structural well-formedness and payload non-interference outside the allowed footprint. If an edit changes internal structure, the affected subtree is refreshed before subsequent steps. When remaining operations are no longer valid under the refreshed hierarchy, execution fails closed or replans from the updated state, depending on configuration.

Worker outputs are validated before application. Invalid outputs are rejected, optionally with bounded retry, and failures are surfaced as explicit execution errors rather than silently applied edits. This deterministic enforcement layer is the mechanism behind the safety claims in Section 4.

4 Functional Safety Properties

A central objective of the architecture is to reduce unintended changes to the input. By constraining edits to a small, symbolic edit algebra and executing plans through a deterministic executor, the system is designed to enforce structural and semantic properties that token-level generation does not typically provide.

Planner Capability Assumption. The formal results in this section assume that the planning stage (router and/or delegated planner) emits a well-formed symbolic plan. The executor computes a structural footprint *per operation* (denoted $F_{\text{struct}}(o_i)$) that is closed under every *structural* change a step may induce: moved subtrees, sibling-order changes (root order for roots), and any structural insert/delete effects produced by subtree refresh. If the plan supplies an intended structural footprint T, the executor checks $F_{\text{struct}}(o_i) \subseteq T$ for every step and rejects the plan otherwise. If no T is provided, the executor may set $T \coloneqq \bigcup_i F_{\text{struct}}(o_i)$ by default. When a plan passes these checks, the executor enforces payload preservation outside each step's payload footprint (defined below) in the formal model under these assumptions (Theorem 2). Worker modules that implement EDIT operations may be stochastic, but in this formulation their effects are confined to the step's payload footprint; payload outside that footprint is byte-identical for that step. When re-extraction is required, it is scoped to edited subtrees when possible and escalates to parent or full-document scope only when required for well-formedness or lossless rendering. If the router outputs malformed JSON,

an invalid operator, or a reference to a nonexistent node, the executor rejects the plan and performs no edit. Full-document rewrites are permitted by setting $T{=}N$, in which case the side-effect claim becomes vacuous but the same execution model applies. The router-hybrid implementation used in experiments enforces equivalent locality via protected-section invariants and paragraph-level footprint checks rather than an explicit T. These formal guarantees apply to the executor defined here and to any implementation that satisfies the same footprint-closure and refresh assumptions. Plan validation simulates each operation and checks invariants per step; during execution, invariants are enforced step-by-step. The executor may refresh the hierarchy for span alignment, but refresh scopes induced by EDIT are handled per step. For a fixed plan, this implies that mixed plans are equivalent to executing each step in isolation; when the system replans between steps (e.g., after a hierarchy refresh), the equivalence applies to each plan segment, not to the full replan loop.

4.1 Edit Algebra

The edit algebra specifies the finite set of symbolic operations that the planning stage may emit. These operators form the primary interface between the stochastic planning stage and the deterministic executor in this formulation, and thereby define the space of permissible transformations.

Let $\mathcal{O}$ denote the operator set defined in Section 2.3. It consists of five typed operators—MOVE, EDIT, INSERT, DELETE, and ANNOTATE—plus the special directive DELEGATEPERMUTATION, which expands deterministically to a MOVE after validation.

The executor interprets these operators symbolically in this formulation. Token-level generation occurs inside worker calls attached to EDIT, and unspecified spans remain untouched. This separation—symbolic structure at the top level, bounded generation at the leaves—is the foundation of functional safety.

Implementation note. Concrete plan schemas map to the abstract EDIT/MOVE/INSERT/DELETE/ANNOTATE operators; the executor validates structural preconditions and span consistency before applying a plan.

Whitespace handling (appendix pointer). Detailed whitespace handling, canonicalization, and separator policies are documented in Appendix D.1.

4.2 Safety Invariants

We now formalize the correctness properties enforced by the deterministic executor. Let R be a well-formed hierarchy, let $P \in \mathcal{P}$ be a symbolic plan, and let

$$R' = E(R, P)$$

be the result of executing the plan. For per-step reasoning, let $R_0 = R$ and $R_i = E(R_{i-1}, (o_i))$ denote the intermediate hierarchy after step i. For a single operator o, let Targets_o denote the nodes it directly references, and $\text{Targets}_o(\textsc{Edit})$ the subset referenced by EDIT. Unless an edit is constrained to a strict subset of a leaf node, it may change internal structure within its scope; therefore the executor re-extracts the affected subtree after each step. If heading structure changes, the executor may lift the refresh to a parent scope. For a single operator o, let

$$\text{Refresh}(o) := \{\, n \in N : n \text{ is the root of a refresh scope induced by some } \textsc{Edit}(\cdot,\cdot,\emptyset) \text{ in that step} \,\},$$

and write $\text{Subtree}(\text{Refresh}(o))$ for all nodes in those refreshed scopes in the *pre-step* hierarchy.

Payload footprint. For a single operator o, define

$$\begin{aligned} F_{\text{payload}}(o) := {} & \text{Targets}_o(\textsc{Edit}) \cup \text{Subtree}(\text{Refresh}(o)) \cup \text{Subtree}(\text{Targets}_o(\textsc{Move})) \\ & \cup \text{Parents}_{\text{payload}}(o) \cup \text{Inserted}(o) \cup \text{Deleted}(o) \cup \text{Adjacent}(o), \end{aligned}$$

where $\text{Adjacent}(o)$ tracks adjacency primarily through the boundary gap nodes (inter-paragraph whitespace) at the source and destination boundaries of each MOVE/INSERT/DELETE in that step. Because whitespace

is modeled as payload, those gap nodes are payload-touching by construction. The boundary leaf nodes on each side are included only as a conservative halo because separator insertion can touch their local payload. These nodes may be handled as *soft* under optional normalization policies. Here Inserted(o) includes nodes created by explicit INSERT *or* by refresh-based re-extraction, and Deleted(o) includes nodes removed by DELETE *or* by refresh-based re-extraction. We define

$$\mathrm{Parents}_{\mathrm{payload}}(o) := \mathrm{Parents}_o(\textsc{Move}\cup\textsc{Insert}\cup\textsc{Delete}) \cup \mathrm{Ancestors}(\mathrm{Parents}_o(\textsc{Move} \cup \textsc{Insert} \cup \textsc{Delete}))\,,$$

where $\mathrm{Parents}_o(\cdot)$ denotes parents whose child ordering or local delimiters may change because of MOVE /INSERT/DELETE in that step, and $\mathrm{Ancestors}(S)$ denotes all ancestors of nodes in S. This expansion accounts for local payload owned by parents (inter-child whitespace and delimiters): reordering, inserting, or deleting children can change that local payload even when child content is unchanged. This makes the payload region explicit: payload side effects (creation, modification, deletion) are confined to $F_{\mathrm{payload}}(o)$ for each step.

Structural footprint. Define the structural footprint for a single step as

$$F_{\mathrm{struct}}(o) := \{\, n \in N \mid \text{node } n\text{'s label/parent/children/sibling position or existence changes under } R \xrightarrow{o} R' \,\},$$

which is computed by simulating the step and collecting nodes whose *structural signature* changes (plus insertions/deletions). For intuition, a conservative superset of $F_{\mathrm{struct}}(o)$ is given by the union of $\mathrm{Targets}_o$, $\mathrm{Parents}_o$, Subtree(Refresh(o)), Inserted(o), and Deleted(o); this superset may also include $\mathrm{Anchors}_o$ when anchors are treated as "touched" for access control. $F_{\mathrm{payload}}(o)$ overlaps $F_{\mathrm{struct}}(o)$ but may include nodes whose structure is unchanged (boundary siblings via the separator halo or parents with local delimiters). Structure-only operations (e.g., MOVE) live in the set difference $F_{\mathrm{struct}} \setminus F_{\mathrm{payload}}$. Cross-parent moves change parent/child links for source and destination ancestors; therefore we assume the intended target set T is *closed* under these structural dependencies (i.e., if a node is in $F_{\mathrm{struct}}(o_i)$ then all ancestors whose child lists must change are also in T). The executor computes $F_{\mathrm{struct}}(o_i)$ per step and either checks $F_{\mathrm{struct}}(o_i) \subseteq T$ when a planner-provided T is available, or sets $T := \bigcup_i F_{\mathrm{struct}}(o_i)$ when absent.

Footprint callout. The executor enforces $F_{\mathrm{struct}}(o_i)$ as the minimal region where *structure* may change for each step, and the *payload* side-effect claim is scoped to $F_{\mathrm{payload}}(o_i)$. Spans may be reindexed deterministically due to payload-length shifts or refreshes. Local payload outside $F_{\mathrm{payload}}(o_i)$ is unchanged for that step. Label changes are structural and therefore included in F_{struct}. Structure-only operators preserve the rendered content of moved subtrees (Invariant 4.2). The rendered content of a parent can change when children move even if the local payload does not. The executor checks payload preservation per step by reconciling entity identifiers across pre/post hierarchies and comparing a *local payload signature* (text outside child spans) for entities outside $F_{\mathrm{payload}}(o_i)$, so span reindexing is permitted while payload changes are rejected. This local payload corresponds to $p(n)$; the full rendered content can change when children move and is therefore not used for non-interference checks. The executor supports a configurable non-interference policy. Policy A (default) enforces strict byte-identical payloads outside $F_{\mathrm{payload}}(o_i)$ on the canonicalized representation $C(X)$ that preserves paragraph separators. Optional policies may relax this check for whitespace-only changes in soft adjacency nodes or prose labels, and a strict-raw mode can disable separator preservation (at the cost of possible paragraph collapse after structural moves). These variants are not assumed by Theorem 2 unless explicitly stated.

Whitespace halo. Because separator canonicalization can adjust the boundary gap between adjacent paragraphs, $F_{\mathrm{payload}}(o_i)$ includes a minimal whitespace halo: the gap nodes between neighboring semantic siblings at the edit boundary and the boundary leaf nodes of those neighbors. This keeps the non-interference guarantee precise while allowing the expected local edits (e.g., inserting a new root before a leading blank-line preamble or appending a paragraph after a nested subsection). Appendix F gives examples and the concrete footprint procedure.

Entity reconciliation and scoped refresh. When refresh-based execution re-extracts a hierarchy, the executor computes a reconciliation map between pre- and post-refresh nodes. Reconciliation is structure-first (parent/position anchors and spans are primary signals) with payload similarity used only as a tie-breaker. Outside the payload footprint, reconciliation must be total and unambiguous; otherwise execution fails (fail-closed). Inside the footprint, unmatched nodes are permitted and receive fresh entity identifiers. Merge and

split events create new entity identifiers and record lineage edges, so continuity is preserved without reusing prior identities. When a plan allows a full-document edit ($T{=}N$), the reconciliation requirement becomes vacuous because every node lies inside the footprint.

Remark (Why closure is necessary). The closure condition on T is not an extra "weakening" of the claim; it captures all *structural* dependencies induced by an operation. For example, a cross-parent MOVE that relocates a paragraph entails (i) updating the source parent's child list and (ii) updating the destination parent's child list. Those parents and their ancestors are therefore structurally touched even though their local payloads remain identical (their rendered spans may change). Declaring them in T isolates the structural changes while preserving the core claim: every node outside $\bigcup_i F_{\text{payload}}(o_i)$ remains payload-identical under the model assumptions.

Invariant 1 (Payload Non-Interference). For a single step $R_{i-1} \xrightarrow{o_i} R_i$, if a node $n \notin F_{\text{payload}}(o_i)$, then its payload is unchanged in that step:

$$n \notin F_{\text{payload}}(o_i) \;\Rightarrow\; p_{R_i}(n) = p_{R_{i-1}}(n).$$

Invariant 1a (Rendered-Span Note). Rendered content is derived from payload and children, so span indices can shift deterministically when payload lengths change; nodes may be outside F_{struct} even if their spans shift. Payload preservation claims are limited to nodes outside $F_{\text{payload}}(o_i)$ for each step and to moved subtrees (Invariant 4.2). Rendered spans may change under edits or moves without violating payload non-interference.

Invariant 2 (Subtree Preservation for Structural Edits). If an operator rearranges nodes but does not specify new payload, then the payload of any moved subtree is preserved exactly:

$$o_i \in \{\textsc{Move}(\pi), \textsc{Move}(\cdot)\} \;\Rightarrow\; p_{R'}(n) = p_R(n) \quad \text{for all } n \text{ in the moved subtree of } o_i.$$

Equivalently, the rendered content of the moved subtree is identical and only its position in the document changes.

Invariant 3 (Structural Soundness). For every execution, the resulting hierarchy remains well-formed: nesting, acyclicity, and non-overlap constraints hold under the stated assumptions. The executor is designed to reject or repair any operation that would violate structural integrity.

Discussion. The executor enforces these invariants independently of planner behavior: even if the planner is stochastic or sampled at high temperature, each plan is validated independently and each execution is safe under the model assumptions.

Bounded model-checking complement. In addition to the theorem-level proofs in this section, we model-check a bounded TLA+ abstraction of the orchestration loop and worker-output guardrails. The checked properties are FS-INV-001 (NoSideEffects), FS-INV-002 (NoIntroducedArtifacts), FS-INV-003 (RetryBoundedness), and FS-INV-004 (TerminationUnderBounds). Appendix G summarizes the checked bounds and explicit out-of-scope realism assumptions.

Lemma 1 (Preservation)**.** *Let R be a well-formed hierarchy, and let o be a single symbolic operator satisfying the typing rules of Section 2.4. If*

$$R \xrightarrow{o} R',$$

then R' is also a well-formed hierarchy (nesting, acyclicity, and non-overlap).

Proof. By typing, o belongs to the edit algebra $\mathcal{O}$ and satisfies the structural preconditions for its operator type:

- $\textsc{Move}(\pi)$ implements a pure permutation of section buffers; moved subtree bytes are preserved, and spans may be recomputed deterministically, but nesting and non-overlap are preserved by construction.

- $\textsc{Move}(n, \text{pos})$ relocates the subtree rooted at n. The executor updates parent/child links and re-renders the document from leaf payloads before spans are reindexed. The executor additionally rejects cycles (e.g., moving a node into its own subtree) and rejects any move that would violate non-overlap. Hence well-formedness is preserved.
- $\textsc{Edit}(s_i, r, \theta)$ modifies the content of a crop bracketed entirely within $s(s_i)$; when the edit can change internal structure (e.g., θ is empty and the replacement may split or merge leaves), the affected subtree (or a lifted parent scope) is re-extracted deterministically and therefore satisfies well-formedness by construction. Otherwise, the executor does not change any spans or structural relations, so the hierarchy remains well-formed.
- $\textsc{Insert}(\text{content}, \ell, \text{pos})$ adds a new node with a fresh identifier whose span lies within its parent span (when known) and does not overlap siblings; no existing node is modified, so well-formedness is preserved.
- $\textsc{Delete}(n)$ removes a subtree; the executor updates parent–child links to maintain structural validity.
- $\textsc{Annotate}(n, \ell)$ updates a node's label or metadata without changing its payload or child relations; this changes the structural signature but preserves well-formedness.

Thus no operator in $\mathcal{O}$ can introduce overlapping spans, cyclic parentage, or violations of nesting. Hence R' is well-formed. □

Lemma 2 (Progress)**.** *Let R be a well-formed hierarchy, and let o be a well-typed operator. Then either:*

1. *the plan is empty (execution halts), or*
2. *$R \xrightarrow{o} R'$ for some hierarchy R' (or the executor returns an error).*

Furthermore, when worker calls succeed and their outputs are parseable by the deterministic extractors, the step $R \xrightarrow{o} R'$ is total and deterministic; otherwise the executor reports a worker-output error.

Proof. Well-typed operators fall into one of five cases.

Case 1: $o = \textsc{Move}(\pi)$. A valid permutation can be applied by reassembling section buffers. No stochasticity is introduced by this operator in the formal model, so R' exists uniquely.

Case 2: $o = \textsc{Edit}(s_i, r, \theta)$. Typing stipulates s_i exists and θ (if provided) is within its span. Thus the executor can: (i) extract the needed content, (ii) call the worker LLM (or use provided replacement content), (iii) splice the result, yielding a unique R' when the worker succeeds and its output is parseable. If the edit can change internal structure (e.g., θ is empty and the replacement may split or merge leaves), subtree (or lifted parent-scope) re-extraction is deterministic under the same parseability assumption.

Case 3: $o = \textsc{Insert}(\text{content}, \ell, \text{pos})$. Typing stipulates pos is valid. The executor can create the new node and insert it, yielding a unique R'.

Case 4: $o = \textsc{Delete}(n)$. Typing stipulates $n \in N$. The executor can remove the subtree rooted at n, yielding a unique R'.

Case 5: $o = \textsc{Annotate}(n, \ell)$. Typing stipulates $n \in N$ and ℓ is a valid label. The executor updates the node's label/metadata without modifying payload or parent/child relations; the structural signature changes only in the label, yielding a unique R'.

In all cases, the executor defines a unique next state. Hence execution can proceed unless the plan is empty. □

Corollary 1 (Type Soundness for Symbolic Plans)**.** *Repeated execution of a well-typed symbolic plan:*

$$R_0 \xrightarrow{o_1} R_1 \xrightarrow{o_2} \dots \xrightarrow{o_k} R_k$$

preserves well-formedness at every step and does not get stuck under the typing assumptions.

4.3 Deterministic Safety Theorem

The invariants above yield our primary correctness result: if a plan is well-typed and footprint-contained, then execution is payload-side-effect-free and structurally well-formed in the formal model.

Theorem 2 (Deterministic Safety)**.** *Let $R = (N, E, \ell, s, p)$ be a well-formed hierarchy, let $T \subseteq N$ be the intended structural target set, and let P be a symbolic plan such that* $\mathrm{Good}(P, R)$ *holds, where* $\mathrm{Good}(P, R)$ *is defined by:*

(i) P is well-typed for R (Section 2.4),

(ii) $F_{\text{struct}}(o_i) \subseteq T$ for every step (equivalently, $\bigcup_i F_{\text{struct}}(o_i) \subseteq T$), and

(iii) every operation belongs to $\mathcal{O}$ and satisfies its structural preconditions (including span containment and non-overlap for INSERT*).*

Then the executor output $R' = E(R, P)$ is payload-side-effect-free in the formal model: for every step $R_{i-1} \xrightarrow{o_i} R_i$, all nodes $n \notin F_{\text{payload}}(o_i)$ retain identical payload in that step, structure-only moves preserve the rendered content of moved subtrees (Invariant 4.2), and every R_i is structurally well-formed. Consequently, all nodes $n \notin \bigcup_i F_{\text{payload}}(o_i)$ retain identical payload in R'. For full-document rewrites, choose T=N, in which case the side-effect claim becomes vacuous but the same execution semantics apply.

Composition. Because execution is deterministic for a fixed plan, sequential composition agrees with batching when the plan is executed as a single fixed sequence: for any two plans P_1, P_2 that are each well-typed and footprint-contained, $E(R, P_1 \parallel P_2) = E(E(R, P_1), P_2)$ provided the combined structural footprint respects closure and the plan remains well-typed on the intermediate hierarchy. This is the case when the executor validates the full plan and does not trigger replanning. This justifies emitting compound edits either as a single plan or as a series of calls without changing observable outcomes. When the runtime replans between steps, the planning stage emits a new plan conditioned on the updated hierarchy, so the composition statement applies per plan segment rather than to the entire replan loop.

Proof. We view execution as a small-step relation on hierarchies: for a single operator o, write

$$R \xrightarrow{o} R' \quad \text{iff} \quad R' = E(R, (o)).$$

By definition of E, multi-step execution of a plan $P = (o_1, \ldots, o_k)$ is the reflexive, transitive closure of this relation:

$$R \xrightarrow{o_1} R_1 \xrightarrow{o_2} \ldots \xrightarrow{o_k} R_k,$$

with $R_k = E(R, P)$.

We prove that, under $\mathrm{Good}(P, R)$, every step preserves (i) structural well-formedness and (ii) payload identity of all nodes outside the payload footprint; the theorem then follows by composition of steps.

Step-wise preservation. Fix an intermediate hierarchy $R_j = (N, E, \ell, s, p)$ and a single operator o_j satisfying the clauses of $\mathrm{Good}(P, R)$:

(a) o_j is well-typed for R_j,

(b) $F_{\text{struct}}(o_j) \subseteq T$, and

(c) $o_j \in \mathcal{O}$ and satisfies its structural preconditions.

Let R_{j+1} be the result of applying o_j, i.e. $R_j \xrightarrow{o_j} R_{j+1}$.

By (a), the typing rules of Section 2.4 imply that the executor applies o_j without violating any local structural constraints. Invariant 4.2 therefore implies that R_{j+1} is well-formed whenever R_j is well-formed.

By Invariant 4.2, for all $n \notin F_{\text{payload}}(o_j)$,

$$p_{R_{j+1}}(n) = p_{R_j}(n).$$

Clause (b) ensures the structural changes for o_j are confined to T, while Clause (c) restricts o_j to the edit algebra $\mathcal{O}$. For MOVE operators, Invariant 4.2 shows that each moved subtree preserves its rendered content; for a root-level permutation this implies $p_{R_{j+1}}(n) = p_{R_j}(n)$ for all $n \in N$.

Global preservation. We now compose these step-wise properties along the unique execution trace induced by P. Write

$$R_0 = R, \quad R_{j+1} = E(R_j, (o_{j+1})), \quad R_k = E(R, P).$$

Since R_0 is well-formed by assumption, repeated application of Invariant 4.2 shows that every R_j is well-formed, in particular $R_k = R'$. Likewise, repeated application of Invariant 4.2 shows that for all $n \notin \bigcup_i F_{\text{payload}}(o_i)$ and all j, the payload of n is identical in R_j and R_0.

Therefore, in the final hierarchy R', every node $n \notin \bigcup_i F_{\text{payload}}(o_i)$ is payload-identical relative to R, and the hierarchy remains well-formed. This is precisely the payload-side-effect-free property claimed in the theorem. □

Worker conformance. Let $\text{WorkerOK}(P, R)$ denote the event that all worker outputs produced during execution of plan P conform to the executor interface (i.e., pass validation/guardrails). Define the combined event

$$\text{GoodExec}(P, R) \coloneqq \text{Good}(P, R) \wedge \text{WorkerOK}(P, R).$$

Lemma 3 (Conditional Functional Safety)**.** *Fix a model, temperature τ, input X, and hierarchy R. Let $R = (N, E, \ell, s, p)$. Let $T \subseteq N$ be the intended structural target set, and draw $P \sim P_\tau(\cdot \mid X, R)$. Suppose*

$$\mathbb{P}\big(\text{GoodExec}(P, R)\big) \geq 1 - \epsilon$$

for some $0 \leq \epsilon \leq 1$, where $\text{Good}(P, R)$ is as defined in Theorem 2 and $\text{WorkerOK}(P, R)$ requires worker outputs to conform to the executor interface. In particular, $\text{Good}(P, R)$ includes the per-step footprint-closure condition $F_{\text{struct}}(o_i) \subseteq T$ for all i. Let $R' = E(R, P)$ be the executor output. Then the probability that execution produces any side effect—specifically, modifies payload outside $\bigcup_i F_{\text{payload}}(o_i)$ or violates the structural invariants—is at most ϵ.

Proof. Theorem 2 shows that whenever $\text{GoodExec}(P, R)$ holds, execution is payload-side-effect-free with probability 1 in the formal model under the stated assumptions. By the law of total probability,

$$\begin{aligned}\mathbb{P}(\text{side-effect}) = {} & \mathbb{P}(\text{side-effect} \mid \text{GoodExec}(P, R)) \cdot \mathbb{P}(\text{GoodExec}(P, R)) \\ & + \mathbb{P}(\text{side-effect} \mid \neg\text{GoodExec}(P, R)) \cdot \mathbb{P}(\neg\text{GoodExec}(P, R)).\end{aligned}$$

The first term vanishes by Theorem 2. The second term is bounded by $\mathbb{P}(\neg\text{GoodExec}(P, R)) \leq \epsilon$, yielding the claim. □

Empirical support and practical implications. Lemma 3 makes explicit that functional safety claims are *conditional* on the planning stage emitting a valid symbolic plan and on worker outputs conforming to the executor interface. Temperature τ influences the probability of these events, not the correctness of execution itself. The experiments in Section 5 validate this behavior: for the primary setting ($\tau = 0.3$) and pilot sweeps at nearby temperatures, router failures manifest as malformed JSON rather than incorrect symbolic reasoning. When a plan is malformed, the system detects and rejects it, performing no edit; safety is preserved by construction.

4.4 Robustness to Planner Stochasticity

The deterministic safety theorem applies pointwise to each plan individually. Thus, variability in the planner induced by temperature or decoding strategy does not affect correctness of execution: each sampled plan is validated independently.

Formally, let $P_\tau(\cdot \mid X, R)$ denote the planner's distribution over symbolic plans at temperature τ. For any two plans P_1, P_2 such that $\mathrm{Good}(P_j, R)$ holds and worker outputs conform to the executor interface, the deterministic safety theorem implies that

$$E(R, P_1) \text{ and } E(R, P_2)$$

are both payload-side-effect-free. Safety therefore depends on whether a sampled plan is valid and worker outputs conform, not on how plans vary under temperature.

5 Experiments

We evaluate Functional Safety (FS) across targeted paragraph editing, contextual insertion, code visibility refactors, long-form section relocation, paragraph consolidation, LaTeX micro-edits, and multi-file refactors. Single-document experiments compare three tiers of editing approach: (i) a *flat autoregressive baseline* that rewrites full document spans as a naive lower bound, (ii) a *ReAct-style tool agent* reflecting current agentic editor architectures (Section 5.9), and (iii) the *hierarchy-aware Functional Safety planner–executor*. Large multi-file code refactors instead compare CLI frontier baselines against the hierarchical planner–executor. Unless noted otherwise, remote runs use provider-default temperatures and local runs use $T = 0.3$. Stress tests (pure permutation and compound multi-operation edits) are summarized in Appendix E. Additional method comparisons (guardrail+regeneration and constrained JSON span edits) are reported in Section 5.10.

The ReAct baselines use the same model roster (`gpt-4.1` and `grok-4-1-fast-reasoning`) as the flat and FS conditions. Each variant runs a standard observe–think–act loop with 12 editing tools (list headings, search, view, and span-editing or patching primitives). We evaluate two tool modes: *basic* (raw span editing with an 8-step budget) and *apply_patch* (context-matching unified diffs with a 12-step budget). The *apply_patch* variant mirrors how production editing agents (e.g., Cursor, GitHub Copilot) apply changes via context-matching patches rather than raw character offsets; it represents a strong baseline for current agentic editing practice.

Table 1: Experimental overview. Each row represents a main experiment with its content type, scale, task, and corresponding results.

Experiment	Content	Scale	Task	Tables
Targeted Paragraph Editing	Policy documents	≈2k chars	Single paragraph fix	3
Contextual Insert	Policy documents	≈2k chars	Insert summary with context	6
Code Visibility Refactor	Python module	≈1.7k–3.1k chars	Group private methods + fix typo	7, 8
Long-Form Section Relocation	Policy briefs	10k, 20k chars	Move section + fix typo	9, 10
Paragraph Consolidation	Policy briefs	10k, 20k chars	Merge two paragraphs	11, 12
LaTeX Micro-Edits	LaTeX (book)	10k, 20k chars	Fix math / `\ref` / `\cite` typos	13, 14, 15, 16, 17, 18
Code Multi-File Refactor (Compute)	Python (package)	13 files	Split monolith into package tree	20
Code Multi-File Refactor (Vector)	Python (package)	49 files	Split monolith into package tree	21
Code Multi-File Refactor (Imports-top)	Python (package)	13, 49 files	Split monolith with consolidated imports	22, 23
Code Multi-File Refactor (Open, relaxed)	Python (package)	13, 49 files	Open-ended split into package tree	24, 26
Code Multi-File Refactor (Open, compat)	Python (package)	13, 49 files	Open-ended split with import-path compatibility	25, 27
LaTeX Multi-File Refactor	LaTeX (book)	5 files	Split inline chapters into `chapters/`	28
LaTeX Multi-File Refactor (Appendix)	LaTeX (book)	4 files	Split inline appendix with cross-refs	29

Experimental Overview. Table 1 summarizes the experimental progression. Each experiment isolates a different dimension of difficulty: document scale (2k to 20k characters), content type (prose vs. code), and task complexity (single-target edits vs. compound structural-plus-semantic operations). The long-form policy-brief tasks share a single base template with fixed section titles and paragraph markers; we apply relocation and consolidation edits to the same underlying document, padding to 10k or 20k characters to control scale.

Metrics include *Success Rate*, *Side-Effect Rate*, and wall-clock *Time*. We report strict (byte-level) and whitespace-normalized variants; side effects are measured against the task-defined target region (not the planner footprint). For rate estimates we report 95% Wilson confidence intervals. Per-task tables show all three tiers (Flat, ReAct+`apply_patch`, Functional Safety) side by side with both WS and strict metrics, enabling direct comparison across approaches.

Default remote runs use provider-default temperatures on `gpt-4.1` and `grok-4-1-fast-reasoning`; some remote providers do not expose temperature control. Local models use $T = 0.3$. We run scope-depth and sparse-hints ablations, and report stress-test diagnostics in Appendix E. For paragraph consolidation, success is defined as reducing the target section's paragraph count by exactly one. Multi-file refactor experiments compare CLI-style baselines to the hierarchical planner/executor (rather than flat full-span rewrites) because the task is reconstructing a file tree from a monolith.

Table 2: Summary of key results aggregated across task families. Flat and FS columns use strict (byte-level) metrics; ReAct+patch columns use whitespace-normalized (WS) metrics for this summary. Full per-task tables report both WS and strict metrics for all three tiers. All rates are proportions with 95% Wilson confidence intervals. Detailed per-condition tables are reported inline; local-model tables appear in Appendix B.

Task Family	Flat Succ.	Flat SE	ReAct+patch Succ. (WS)	ReAct+patch SE (WS)	FS Succ.	FS SE	N
Targeted Paragraph Editing	0.00[†]	1.00	0.97 [0.83, 0.99]	0.03 [0.01, 0.17]	1.00 [0.89, 1.00]	0.00 [0.00, 0.11]	30
Contextual Insert	0.00	1.00	0.77 [0.59, 0.88]	0.23 [0.12, 0.41]	1.00 [0.89, 1.00]	0.00 [0.00, 0.11]	30
Code Visibility Refactor	0.00	1.00	0.00 [0.00, 0.06]	0.35 [0.24, 0.48]	1.00 [0.94, 1.00]	0.00 [0.00, 0.06]	60
Long-Form Section Relocation	0.00	1.00	0.10 [0.05, 0.20]	0.73 [0.61, 0.83]	1.00 [0.94, 1.00]	0.00 [0.00, 0.06]	60
Paragraph Consolidation	0.00	1.00	0.92 [0.82, 0.96]	0.02 [0.00, 0.09]	1.00 [0.94, 1.00]	0.00 [0.00, 0.06]	60
LaTeX Micro-Edits	0.00	1.00	0.74 [0.67, 0.80]	0.26 [0.20, 0.33]	1.00 [0.98, 1.00]	0.00 [0.00, 0.02]	180

[†] Flat baseline CIs omitted: all conditions are exactly 0.00 strict success and 1.00 strict side-effect.

All methods aggregate two models (gpt-4.1, grok-4-1-fast-reasoning) × 15 runs per condition.

Table 2 reveals a three-tier pattern. The flat baseline always fails strict success and always introduces side effects—it serves as a naive lower bound. The ReAct+`apply_patch` agent nearly matches Functional Safety on targeted paragraph editing (97% success, 3% SE) and paragraph consolidation (92% success, 2% SE), but struggles on structural tasks: code visibility refactors (0% success, 35% side effects) and section relocation (10% success, 73% side effects). On LaTeX micro-edits, ReAct+`apply_patch` achieves 74% aggregate success with 26% side effects—reliable on simple symbol fixes but degrading on multi-location edits (cross-references, citations), where incorrect edits within the target section are counted as side effects. Functional Safety succeeds across the board by confining payload changes to hierarchy-aware per-step footprints, though planning-stage mis-scoping remains a source of residual failures on complex tasks. Detailed per-task tables are reported inline below; the ReAct baselines are described in Section 5.9.

5.1 Targeted Paragraph Editing

The Targeted Paragraph Editing task evaluates models on three multi-section documents per condition (each ≈2,000 characters). Each trial requests a single targeted edit—fixing deliberate spelling errors in a specific paragraph—while requiring all untouched material to remain byte-identical. Local model results appear in Appendix B.

Table 3 shows all three tiers side by side. Flat baselines fix the typo but reflow whitespace across the document, yielding strict success = 0.00 (strict side-effect rate = 1.00) while achieving whitespace-normalized success = 1.00. The ReAct+`apply_patch` agent nearly matches Functional Safety here (aggregate 97% success, 3% SE)—the edit is localized enough that a context-matching patch suffices. Functional Safety preserves byte identity and reaches 15/15 strict success for both models.

Table 3: Targeted Paragraph Editing (remote models, default temperature). Functional Safety confines edits to the target paragraph with no strict collateral edits observed in these runs.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.70
gpt-4.1 (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.97
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.87
grok-4-1-fast-reasoning (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.67
grok-4-1-fast-reasoning (ReAct+patch)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	50.11
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	5.44

We evaluate two alternative ways to help the router target edits under tight context budgets:

- **Scope-aware depth expansion.** When the instruction explicitly references a paragraph, the host can expand the router context to include paragraph IDs and previews; disabling this auto-expansion can prevent paragraph-level plans under the same constraints. Table 4 reports prompt size and latency for this ablation (gpt-4.1, default temperature), showing the depth heuristic is an optional implementation choice rather than a core requirement.
- **Sparse, query-driven content hints.** For content-cue edits (e.g., "the paragraph about elephants") with paragraph scope disabled, we replace dense per-node attribute lists with a compact list of only the nodes that match the cue; because most nodes are omitted, the net prompt shrinks despite adding these hints. Table 5 shows that sparse hints preserve locality while reducing router prompt size by more than 2× and improving success rate on the content-cue task.

Table 4: Scope-aware depth ablation (gpt-4.1, default temperature). Auto-expanding paragraph context enables paragraph-scoped plans when explicit scope metadata is omitted.

Task	Router Prompt (chars)	Success Rate (WS)	Success Rate (Strict)	Time (s)
Paragraph Edit (explicit scope) (2k)	4181	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.87
Paragraph Edit (auto depth) (2k)	4313	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.73
Paragraph Edit (sections-only) (2k)	4410	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.55

Table 5: Sparse content-hints ablation (gpt-4.1, default temperature). Lightweight cues preserve locality while reducing router prompt size for content-cue edits.

Condition	Success Rate (WS)	Side-Effect Rate (WS)	Time (s)	Router Prompt (chars)
Sparse hints ON	1.00 [0.72, 1.00]	0.00 [0.00, 0.28]	1.23	3824
Sparse hints OFF	0.80 [0.49, 0.94]	0.00 [0.00, 0.28]	2.24	10480

Together, these two ablations target different reference styles: explicit paragraph mentions benefit from scope expansion (IDs/previews), while content-cue mentions benefit from sparse hints that list only cue-matching nodes.

5.2 Contextual Insert

This task inserts a new summary paragraph at the start of the Executive Summary, conditioned on the Methodology section. To keep prompts compact, the router is given only the Executive Summary target paragraph IDs plus short previews of those targets; full paragraph previews are optional. The Methodology

section, which supplies the content to summarize, is provided separately via read-only CONTEXT_UNITS. The router emits a paragraph-level INSERT operation that creates a new paragraph node before the existing first paragraph; the worker drafts the inserted text using the Methodology section as read-only context.

The task specifies that the summary include the keywords *indexing*, *deterministic*, and *latency*. We additionally request one Methodology-specific detail (e.g., *automated checks*, *manual review*, or *unit of work*) and report its inclusion as a diagnostic, but success is defined by correct insertion with the specified keywords and a side-effect rate of 0 under the corresponding metric. If the model returns the original document unchanged, we count the trial as a failure (missing insert) but not a side-effect event. In this evaluation, all existing paragraphs are expected to remain byte-identical. Local model results appear in Appendix B.

Table 6 shows all three tiers. Flat baselines satisfy the content constraints but introduce whitespace drift outside the inserted paragraph, so strict success remains at 0.00 while whitespace-normalized success reaches 1.00. The ReAct+`apply_patch` agent achieves 77% aggregate success (87% for `gpt-4.1`, 67% for `grok`) with 23% aggregate side effects, confirming that localized insertions are mostly within reach of tool-based editing agents, though occasional patch-matching failures introduce collateral changes. Functional Safety succeeds in 15/15 for both models. Some models occasionally prefix paragraph-insert outputs with an operation header (e.g., `INSERT S1.P0:`); we strip such headers deterministically before validation. Remaining failures in this setting are safe no-ops caused by worker output-format violations for paragraph inserts (e.g., emitting multiple paragraphs separated by blank lines), which we reject to preserve paragraph boundaries. We therefore enable worker-output guardrails (retry=1) by default; Appendix E reports an ablation with guardrails disabled where `gpt-4.1` drops to 14/15 without introducing side effects (Table 48).

Table 6: Contextual Insert (remote models, default temperature). Functional Safety preserves locality and succeeds in all runs for both models.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.96
gpt-4.1 (ReAct+patch)	0.87 [0.62, 0.96]	0.87 [0.62, 0.96]	0.13 [0.04, 0.38]	0.13 [0.04, 0.38]	6.47
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.78
grok-4-1-fast-reasoning (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.12
grok-4-1-fast-reasoning (ReAct+patch)	0.67 [0.42, 0.85]	0.67 [0.42, 0.85]	0.33 [0.15, 0.58]	0.33 [0.15, 0.58]	58.71
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	7.40

5.3 Code Visibility Refactor

The Code Visibility Refactor task assesses symbolic planning on Python classes containing 5–7 non-`__init__` methods (6–8 methods including `__init__`). We treat `__init__` as fixed (not reordered) and include it for visibility. Each trial asks the model to group private methods (those beginning with `_`) at the bottom of the class while fixing a single docstring typo. The resulting classes span approximately 1.7k–3.1k characters without artificial padding. In this evaluation, method bodies and all non-target content are expected to remain byte-identical. Functional Safety applies the router's permutation to pre-extracted method blocks (including indentation and trailing whitespace), and recovers plan JSON from raw provider payloads (e.g., tool-call arguments) when the text response is empty. It enforces the public-then-private constraint (and preserves `__init__` at the top) when the router's order violates the grouping requirement, and falls back to a deterministic single-typo replacement if the worker output is not byte-identical. Tables 7 and 8 show all three tiers. Both the flat baseline and the ReAct+`apply_patch` agent fail completely (0% success at both sizes); the ReAct agent additionally produces aggregate side-effect rates of 43% (small) and 27% (medium)—structural method reordering exceeds the capabilities of context-matching patches. Functional Safety succeeds in all runs for both small and medium classes.

Table 7: Code Visibility Refactor (small classes, default temperature). Functional Safety enforces grouping while preserving all non-target code.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	3.88
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.60 [0.36, 0.80]	0.60 [0.36, 0.80]	13.82
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.48
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	9.02
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.27 [0.11, 0.52]	0.27 [0.11, 0.52]	91.54
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	22.75

Table 8: Code Visibility Refactor (medium classes, default temperature). Functional Safety enforces grouping while preserving all non-target code.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	4.85
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.40 [0.20, 0.64]	0.40 [0.20, 0.64]	11.15
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.58
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.96
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.13 [0.04, 0.38]	0.13 [0.04, 0.38]	110.34
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	22.18

5.4 Long-Form Section Relocation

We stress-test the system on 10k- and 20k-character policy briefs drawn from the shared long-form template. Tables 9 and 10 show all three tiers. The flat baseline regenerates the full document, yielding 0/15 success at both sizes. The ReAct+`apply_patch` agent largely fails: `gpt-4.1` achieves 0% success at both sizes with 80–100% side-effect rates, while `grok` reaches 40% at 10k but 0% at 20k—structural section moves are mostly beyond diff-based editing, and most runs terminate at the parse-error limit. Functional Safety delegates the section permutation to the deterministic executor and issues a single localized edit for the typo, achieving 15/15 strict success with no side effects for both models at both sizes.

Table 9: Long-Form Section Relocation (10k briefs, default temperature). Functional Safety confines edits to the target section.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.96
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.80 [0.55, 0.93]	1.00 [0.80, 1.00]	19.22
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.20
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.18
grok-4-1-fast-reasoning (ReAct+patch)	0.40 [0.20, 0.64]	0.40 [0.20, 0.64]	0.60 [0.36, 0.80]	0.60 [0.36, 0.80]	227.73
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	10.61

Table 10: Long-Form Section Relocation (20k briefs, default temperature). Functional Safety confines edits to the target section.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.80
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	19.57
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.06
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.67
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.53 [0.30, 0.75]	0.53 [0.30, 0.75]	416.54
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	9.05

Each model/size condition is evaluated over fifteen repetitions in these tables; larger repetition sweeps are reported separately when available.

5.5 Paragraph Consolidation

This task merges two adjacent paragraphs inside a designated section of a long policy brief. The change is local, but the document context is long (10k–20k characters), making it a strong test of side-effect control. The router is instructed to use a section-level EDIT (local_edit) rather than a paragraph-level edit so the executor can replace the two-paragraph span in a single operation. Tables 11 and 12 show all three tiers. Flat baselines fail in all runs (0/15 strict success, strict side-effect rate = 1.00). The ReAct+`apply_patch` agent shows nuanced performance: 83% aggregate success at 10k and 100% at 20k, with low side-effect rates ($\leq$3%). The merge is localized enough that context-matching patches usually succeed, but occasional matching failures reduce reliability. Functional Safety completes the merge with no side effects in all runs for both models.

Table 11: Paragraph Consolidation (10k briefs, default temperature). Functional Safety confines edits to the targeted paragraphs.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	11.28
gpt-4.1 (ReAct+patch)	0.80 [0.55, 0.93]	0.80 [0.55, 0.93]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	8.29
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.94
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	25.96
grok-4-1-fast-reasoning (ReAct+patch)	0.87 [0.62, 0.96]	0.87 [0.62, 0.96]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	148.38
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	19.43

Table 12: Paragraph Consolidation (20k briefs, default temperature). Functional Safety confines edits to the targeted paragraphs.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	9.30
gpt-4.1 (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	6.22
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.17
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	25.15
grok-4-1-fast-reasoning (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	129.55
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	19.32

5.6 LaTeX Micro-Edits

We evaluate localized edits on book-style LaTeX documents with chapter-level hierarchy. Each task targets a single line-level fix—an equation symbol, a cross-reference label plus its matching `\ref`, or a citation key inside `\cite`—and requires all other content to remain byte-identical. Functional Safety constrains worker edits to the specific LaTeX span while preserving the surrounding chapter text. Tables 13–18 show all three tiers for each LaTeX sub-task. Flat baselines fail in all conditions (0/15 strict success, strict side-effect rate = 1.00). The ReAct+`apply_patch` agent shows a task-dependent gradient: simple math-symbol fixes succeed reliably (97% at both sizes) with near-zero side effects, but cross-reference updates requiring coordinated changes at two locations (`\label` and matching `\ref`) show strongly model-dependent results (`grok` 93–100% vs. `gpt-4.1` 33–47%), and citation-key fixes diverge similarly (`grok` 87–93% vs. `gpt-4.1` 20–27%). Failures are not clean: the agent frequently edits the wrong location within the target chapter (e.g., replacing the wrong math symbol or label), which is detected as an intra-target side effect. Functional Safety completes all three micro-edits without observed side effects for both models. This illustrates that even localized LaTeX edits become unreliable for context-matching patches when the fix spans multiple locations or requires precise navigation of LaTeX markup.

Table 13: LaTeX Math Edit (10k, default temperature). Functional Safety updates the math symbol while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.26
gpt-4.1 (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.94
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.45
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	2.84
grok-4-1-fast-reasoning (ReAct+patch)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	32.15
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.60

Table 14: LaTeX Math Edit (20k, default temperature). Functional Safety updates the math symbol while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	9.45
gpt-4.1 (ReAct+patch)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	4.86
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	6.20
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.86
grok-4-1-fast-reasoning (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	28.47
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	6.56

Table 15: LaTeX Cross-Reference Update (10k, default temperature). Functional Safety updates the label and matching `\ref` while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	9.10
gpt-4.1 (ReAct+patch)	0.33 [0.15, 0.58]	0.33 [0.15, 0.58]	0.67 [0.42, 0.85]	0.67 [0.42, 0.85]	13.76
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.47
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	4.80
grok-4-1-fast-reasoning (ReAct+patch)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	103.03
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	6.62

Table 16: LaTeX Cross-Reference Update (20k, default temperature). Functional Safety updates the label and matching `\ref` while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	9.59
gpt-4.1 (ReAct+patch)	0.47 [0.25, 0.70]	0.47 [0.25, 0.70]	0.53 [0.30, 0.75]	0.53 [0.30, 0.75]	11.47
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.92
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	3.32
grok-4-1-fast-reasoning (ReAct+patch)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	86.53
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	5.89

Table 17: LaTeX Citation Update (10k, default temperature). Functional Safety updates the citation key while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.97
gpt-4.1 (ReAct+patch)	0.27 [0.11, 0.52]	0.27 [0.11, 0.52]	0.73 [0.48, 0.89]	0.73 [0.48, 0.89]	8.40
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.74
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	2.98
grok-4-1-fast-reasoning (ReAct+patch)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	92.24
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	8.22

Table 18: LaTeX Citation Update (20k, default temperature). Functional Safety updates the citation key while preserving surrounding content.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	10.01
gpt-4.1 (ReAct+patch)	0.20 [0.07, 0.45]	0.20 [0.07, 0.45]	0.80 [0.55, 0.93]	0.80 [0.55, 0.93]	9.89
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	2.95
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	2.65
grok-4-1-fast-reasoning (ReAct+patch)	0.87 [0.62, 0.96]	0.87 [0.62, 0.96]	0.13 [0.04, 0.38]	0.13 [0.04, 0.38]	93.21
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	8.77

Structural LaTeX permutation and relocation results are reported in Appendix E.

5.7 Code Multi-File Refactor

We evaluate two long-running, multi-file Python refactors that split a monolithic file back into its original package layout: `compute` (13 files) and `vector` (49 files). Each monolith is created by concatenating the original package files; the task asks the model to reconstruct the original file tree and preserve public exports. These experiments compare CLI baselines to the hierarchical planner/executor (there is no flat full-span baseline for multi-file reconstruction).

We report *Byte Id* (exact match), *Whitespace-normalized Id* (whitespace-normalized using the line-preserving code policy described above), *AST Id* (AST-equivalence ignoring docstrings), *Sig* Δ (function signature diffs), *Import* Δ (missing/extra imports), *Repair Rounds* (mean number of planner repair loops per run), *Test Pass* (fraction of unit tests passing), and mean wall-clock time per run. Plans are validated; invalid plans produce no output files and are scored as missing. Runs were accumulated over multiple days on a fixed code snapshot; Table 19 summarizes the experimental variants and test suites used in this section, and Tables 20–27 report the per-variant results.

Table 19: Multi-file refactor setup variants. "Exact" uses the original package layout; "open" asks the planner to propose a new layout.

Case	Monolith	Layout target	Prompt constraint	Tests
compute_refactor	concat	exact	reconstruct original tree	compute suite (149 tests)
vector_refactor	concat	exact	reconstruct original tree	vector suite (45 tests, 1 skipped)
compute_refactor_imports_top	imports_top	exact	reconstruct original tree	compute suite (149 tests)
vector_refactor_imports_top	imports_top	exact	reconstruct original tree	vector suite (45 tests, 1 skipped)
compute_refactor_open_relaxed	concat	open	behavior-only layout suggestion	compute suite (149 tests)
compute_refactor_open_compat	concat	open	add import-path compatibility	compute suite (149 tests)
vector_refactor_open_relaxed	concat	open	behavior-only layout suggestion	vector suite (45 tests, 1 skipped)
vector_refactor_open_compat	concat	open	add import-path compatibility	vector suite (45 tests, 1 skipped)

Table 20: Multi-file Python compute refactor (13 files). Metrics include structural equivalence, import/signature diffs, test pass rate, and timing.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	5	0.92+/-0.05	0.92+/-0.05	–	0.92+/-0.05	2.4+/-3.4	1.4+/-1.7/1.4+/-3.1	–	1123.53+/-80.04	1.00+/-0.00
codex_cli_default_medium	5	0	5	0.08+/-0.00	0.08+/-0.00	–	1.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	119.00+/-10.31	1.00+/-0.00
hier_gpt-4o	5	0	5	0.85+/-0.00	0.85+/-0.00	–	0.85+/-0.00	0.0+/-0.0	53.0+/-0.0/49.0+/-0.0	1.2+/-0.4	4.87+/-1.20	1.00+/-0.00
hier_gpt-5	5	0	5	0.94+/-0.08	0.94+/-0.08	–	0.94+/-0.08	0.0+/-0.0	21.2+/-29.0/19.6+/-26.8	1.0+/-0.0	29.23+/-3.61	1.00+/-0.00
hier_grok-4-1-fast-reasoning	5	0	5	0.85+/-0.00	0.85+/-0.00	–	0.85+/-0.00	0.0+/-0.0	53.0+/-0.0/49.0+/-0.0	1.0+/-0.0	25.57+/-1.75	1.00+/-0.00

Table 21: Multi-file Python vector refactor (49 files). Metrics include structural equivalence, import/signature diffs, test pass rate, and timing.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	3	0.53+/-0.24	0.53+/-0.24	–	0.65+/-0.21	1.2+/-2.7	50.8+/-37.8/35.4+/-40.1	–	1256.93+/-730.61	0.97+/-0.01
codex_cli_default_medium	5	0	5	0.02+/-0.00	0.02+/-0.00	–	1.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	563.60+/-103.20	1.00+/-0.00
hier_gpt-4o	5	0	5	0.78+/-0.19	0.78+/-0.19	–	0.78+/-0.19	0.0+/-0.0	24.8+/-25.8/142.2+/-149.1	14.0+/-3.5	23.16+/-2.62	0.98+/-0.01
hier_gpt-5	5	0	5	0.58+/-0.39	0.58+/-0.39	–	0.58+/-0.39	0.0+/-0.0	8.6+/-14.3/793.8+/-776.9	6.6+/-5.1	421.63+/-307.95	0.99+/-0.01
hier_grok-4-1-fast-reasoning	5	0	5	0.87+/-0.28	0.87+/-0.28	–	0.87+/-0.28	0.0+/-0.0	2.6+/-5.8/237.6+/-531.3	3.6+/-3.6	174.46+/-169.68	1.00+/-0.01

We also evaluate a variant where the monolith consolidates all imports into a single block at the top of the file. This stresses the ability to reassign or duplicate imports across the reconstructed modules, and the resulting tree may intentionally differ from the original import placement; identity metrics are therefore reported for completeness alongside import deltas and (when available) tests.

Table 22: Multi-file Python compute refactor with consolidated imports at the top of the monolith.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.12+/-0.18	3.2+/-3.3	15.4+/-21.3/11.8+/-6.2	–	1188.39+/-115.97	1.00+/-0.00
codex_cli_default_medium	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	2.8+/-0.4/3.0+/-0.0	–	538.44+/-156.49	1.00+/-0.00
hier_gpt-4o	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	146.8+/-23.4/57.8+/-23.4	1.6+/-0.5	4.75+/-1.18	1.00+/-0.00
hier_gpt-5	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	178.0+/-0.0/89.0+/-0.0	1.0+/-0.0	54.97+/-2.82	1.00+/-0.00
hier_grok-4-1-fast-reasoning	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	149.2+/-26.3/60.2+/-26.3	1.0+/-0.0	60.86+/-7.66	1.00+/-0.00

Table 23: Multi-file Python vector refactor with consolidated imports at the top of the monolith.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	4	0.03+/-0.05	0.03+/-0.05	–	0.07+/-0.05	16.2+/-24.6	42.6+/-42.6/55.0+/-36.0	–	1861.38+/-1559.05	0.97+/-0.01
codex_cli_default_medium	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.01	0.2+/-0.4	109.8+/-56.7/115.8+/-91.8	–	1888.04+/-349.87	0.99+/-0.01
hier_gpt-4o	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	526.8+/-25.8/187.8+/-25.8	11.0+/-1.9	30.06+/-15.38	0.98+/-0.00
hier_gpt-5	4	0	4	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	554.0+/-0.0/215.0+/-0.0	10.8+/-2.9	660.84+/-82.91	0.98+/-0.01
hier_grok-4-1-fast-reasoning	5	0	5	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	517.0+/-20.1/178.0+/-20.1	9.0+/-5.6	311.74+/-167.36	0.98+/-0.00

Finally, we report open-layout variants where the planner proposes a file layout rather than reconstructing the original tree. The relaxed prompt asks for a cohesive split that preserves behavior, while the compat prompt adds a request to keep existing import paths and public exports working (e.g., via re-exports). In these runs, the *Layout Match* column reflects agreement with the original file layout and is not a success criterion; we therefore emphasize tests and missing/extra diagnostics.

Table 24: Open-layout compute refactor (relaxed). The planner proposes a layout; identity metrics are reported against the original tree for reference.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	0	0.85+/-0.05	0.85+/-0.05	–	0.92+/-0.06	1.0+/-2.2	3.0+/-5.6/5.8+/-11.9	–	1296.86+/-57.99	0.00+/-0.00
codex_cli_default_medium	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	271.57+/-45.50	0.20+/-0.45
hier_gpt-4o	5	0	0	0.15+/-0.09	0.15+/-0.09	–	0.62+/-0.37	0.0+/-0.0	2.4+/-3.3/6.6+/-12.6	0.6+/-0.5	3.42+/-0.96	0.00+/-0.00
hier_gpt-5	5	0	0	0.14+/-0.19	0.14+/-0.19	–	0.33+/-0.45	0.0+/-0.0	0.0+/-0.0/22.8+/-31.2	0.2+/-0.4	34.51+/-13.77	0.20+/-0.45
hier_grok-4-1-fast-reasoning	5	0	0	0.05+/-0.10	0.05+/-0.10	–	0.20+/-0.45	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	0.0+/-0.0	22.34+/-7.07	0.00+/-0.00

Table 25: Open-layout compute refactor (compat). The prompt requests compatibility with prior import paths and public exports.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	0	0.83+/-0.13	0.83+/-0.13	–	0.90+/-0.14	5.2+/-11.6	5.6+/-8.0/6.2+/-11.1	–	1320.36+/-212.39	0.00+/-0.00
codex_cli_default_medium	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	412.92+/-142.06	0.00+/-0.00
hier_gpt-4o	5	0	0	0.08+/-0.05	0.08+/-0.05	–	0.33+/-0.20	0.0+/-0.0	3.6+/-3.3/37.0+/-33.7	0.2+/-0.4	3.05+/-0.92	0.00+/-0.00
hier_grok-4-1-fast-reasoning	4	0	0	0.27+/-0.18	0.27+/-0.18	–	0.75+/-0.50	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	0.0+/-0.0	30.45+/-12.62	0.00+/-0.00

Table 26: Open-layout vector refactor (relaxed). The planner proposes a layout; identity metrics are reported against the original tree for reference.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	0	0.04+/-0.04	0.04+/-0.04	–	0.64+/-0.22	0.0+/-0.0	30.8+/-29.2/66.6+/-68.9	–	1568.02+/-1259.75	0.96+/-0.00
codex_cli_default_medium	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	733.87+/-56.03	0.96+/-0.00
hier_gpt-4o	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	2.6+/-1.5	18.68+/-6.28	0.96+/-0.00
hier_gpt-5	2	0	0	0.33+/-0.14	0.33+/-0.14	–	0.96+/-0.06	0.0+/-0.0	1.0+/-1.4/0.5+/-0.7	1.5+/-2.1	254.63+/-255.50	0.96+/-0.00
hier_grok-4-1-fast-reasoning	5	0	0	0.05+/-0.04	0.05+/-0.04	–	0.70+/-0.41	0.0+/-0.0	0.0+/-0.0/3.8+/-6.1	0.2+/-0.4	67.79+/-9.19	0.96+/-0.00

Table 27: Open-layout vector refactor (compat). The prompt requests compatibility with prior import paths and public exports.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	AST Id	Sig Δ	Import Δ	Repair Rounds	Time (s)	Test Pass
claude_cli_reference	5	0	0	0.05+/-0.04	0.05+/-0.04	–	0.54+/-0.20	0.0+/-0.0	66.6+/-32.5/56.4+/-30.7	–	1303.92+/-1378.88	0.96+/-0.00
codex_cli_default_medium	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	–	883.09+/-175.52	0.96+/-0.00
hier_gpt-4o	5	0	0	0.00+/-0.00	0.00+/-0.00	–	0.00+/-0.00	0.0+/-0.0	0.0+/-0.0/0.0+/-0.0	1.0+/-0.7	14.25+/-2.48	0.96+/-0.00
hier_grok-4-1-fast-reasoning	5	0	0	0.04+/-0.03	0.04+/-0.03	–	0.61+/-0.46	0.0+/-0.0	0.0+/-0.0/118.4+/-184.6	0.0+/-0.0	77.29+/-14.26	0.96+/-0.00

5.8 LaTeX Multi-File Refactor

We evaluate a multi-file refactor in which a book-style LaTeX document has chapter bodies inlined directly into `main.tex`, with each chapter preceded by its `\include` line. The task asks the model to reconstruct the original file tree under `chapters/`, restoring a standalone `main.tex` that contains only the preamble and the `\include` directives. We report three identity metrics: *Byte Id* (exact file match), *Whitespace-normalized Id* (whitespace-normalized match using the line-preserving LaTeX policy described above), and *TeX Id* (whitespace-normalized match plus normalization of `\include` and `\input` paths such as `\include{foo}` versus `\include{foo.tex}`), plus mean wall-clock time per run. We run five repetitions per approach and aggregate the means and standard deviations in Table 28.

Table 28: Multi-file LaTeX chapter refactor (5 runs). The TeX Id metric treats `\include` path suffixes as equivalent; timing is reported as mean wall-clock seconds.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	Time (s)
claude_cli_reference	5	0	5	0.93+/-0.09	0.93+/-0.09	0.93+/-0.09	73.50+/-8.63
codex_cli_default_medium	5	0	5	0.83+/-0.00	0.83+/-0.00	0.83+/-0.00	75.60+/-11.16
hier_gpt-4o	5	0	5	1.00+/-0.00	1.00+/-0.00	1.00+/-0.00	1.12+/-0.19
hier_grok-4-1-fast-reasoning	5	0	5	1.00+/-0.00	1.00+/-0.00	1.00+/-0.00	4.26+/-0.83

We additionally evaluate an appendix refactor where cross-file `\ref` links point from chapters to the appendix and back. This variant stresses whether refactoring preserves labels and references across file boundaries; the same identity metrics plus timing are reported in Table 29.

Table 29: Multi-file LaTeX appendix refactor with cross-file references (5 runs), reporting both correctness and timing.

Approach	Runs	Failed	Layout Match	Byte Id	WS Id	TeX Id	Time (s)
claude_cli_reference	5	0	5	1.00+/-0.00	1.00+/-0.00	1.00+/-0.00	57.53+/-9.93
codex_cli_default_medium	5	0	5	0.70+/-0.27	0.70+/-0.27	0.70+/-0.27	58.61+/-26.42
hier_gpt-4o	5	0	5	1.00+/-0.00	1.00+/-0.00	1.00+/-0.00	1.03+/-0.25
hier_grok-4-1-fast-reasoning	5	0	5	1.00+/-0.00	1.00+/-0.00	1.00+/-0.00	3.69+/-0.39

5.9 ReAct Tool-Agent Baselines

We evaluate a ReAct-style tool agent as a realistic baseline representing current agentic editor architectures. The agent runs a standard observe–think–act loop: at each step it receives the task instruction, the current document state, and tool schemas, then emits a JSON action envelope selecting one tool and its arguments. Invalid JSON triggers lightweight auto-repair (strip prose, fix trailing commas, normalize booleans, quote bare keys) before counting a parse error. Tool failures and parse errors each have bounded budgets; exceeding either terminates the episode.

We test two tool configurations:

- **ReAct basic.** Eight raw span-editing tools (list_headings, search, search_lines, view, view_lines, replace_span, insert_span, delete_span, cut, paste, finish) with an 8-step budget. Edits operate on character offsets directly.

- **ReAct+`apply_patch`.** Read-only tools (list_headings, search, search_lines, view, view_lines) plus a `apply_patch` tool that accepts unified diffs and applies them by matching context lines (`git apply`-style). This uses a 12-step budget and represents how production editing agents apply changes via context-matching patches rather than raw offsets.

Both variants use `gpt-4.1` and `grok-4-1-fast-reasoning` with 15 runs per model per condition and a 60 s per-request timeout. Tables 30 and 31 report per-condition results for each variant.

Table 30: ReAct baseline with raw span-editing tools (15 runs per model per condition). Common failure modes include malformed span arguments, raw-index edits that corrupt headers, and repeated invalid tool calls. `tool_error_limit` reflects repeated invalid tool calls; `parse_error_limit` indicates repeated JSON violations.

Task	Size	Model	Success (WS)	Side-Effect (WS)	Steps	Time (s)	Finish
code_visibility	small	gpt-4.1	0.00 [0.00, 0.20]	0.47 [0.25, 0.70]	7.1	7.4	max_steps (8/15)
code_visibility	small	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.60 [0.36, 0.80]	8.0	211.1	max_steps (15/15)
code_visibility	medium	gpt-4.1	0.00 [0.00, 0.20]	0.60 [0.36, 0.80]	7.9	8.6	max_steps (12/15)
code_visibility	medium	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.53 [0.30, 0.75]	8.0	192.9	max_steps (15/15)
contextual_insert	2k	gpt-4.1	0.20 [0.07, 0.45]	0.80 [0.55, 0.93]	5.0	5.0	finished (15/15)
contextual_insert	2k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.9	46.6	finished (15/15)
doc_relocate	10k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	7.6	6.3	max_steps (11/15)
doc_relocate	10k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	7.5	79.6	finished (13/15)
doc_relocate	20k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	7.5	6.1	finished (9/15)
doc_relocate	20k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	7.3	81.0	finished (14/15)
doc_reorder	10k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	5.1	3.8	finished (15/15)
doc_reorder	10k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.00 [0.00, 0.20]	5.2	88.0	finished (14/15)
doc_reorder	20k	gpt-4.1	0.00 [0.00, 0.20]	0.73 [0.48, 0.89]	4.5	3.3	finished (15/15)
doc_reorder	20k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.0	84.9	finished (15/15)
latex_cite_update	10k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.8	3.4	finished (15/15)
latex_cite_update	10k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	4.5	23.2	finished (15/15)
latex_cite_update	20k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	5.1	4.0	finished (15/15)
latex_cite_update	20k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	4.6	26.0	finished (15/15)
latex_math_edit	10k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.0	2.9	finished (15/15)
latex_math_edit	10k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	5.1	31.6	finished (15/15)
latex_math_edit	20k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.1	3.2	finished (15/15)
latex_math_edit	20k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.0	37.2	finished (14/15)
latex_xref_update	10k	gpt-4.1	0.13 [0.04, 0.38]	0.87 [0.62, 0.96]	6.0	5.6	finished (15/15)
latex_xref_update	10k	grok-4-1-fast-reasoning	0.53 [0.30, 0.75]	0.47 [0.25, 0.70]	6.3	70.1	finished (14/15)
latex_xref_update	20k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	6.0	5.5	finished (15/15)
latex_xref_update	20k	grok-4-1-fast-reasoning	0.60 [0.36, 0.80]	0.40 [0.20, 0.64]	6.9	88.6	finished (13/15)
para_consolidate	10k	gpt-4.1	0.00 [0.00, 0.20]	0.67 [0.42, 0.85]	5.0	5.5	finished (15/15)
para_consolidate	10k	grok-4-1-fast-reasoning	0.87 [0.62, 0.96]	0.00 [0.00, 0.20]	7.6	115.6	finished (13/15)
para_consolidate	20k	gpt-4.1	0.00 [0.00, 0.20]	0.73 [0.48, 0.89]	5.1	5.8	finished (15/15)
para_consolidate	20k	grok-4-1-fast-reasoning	0.87 [0.62, 0.96]	0.00 [0.00, 0.20]	7.9	116.0	finished (12/15)
para_edit	2k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	4.5	3.4	finished (15/15)
para_edit	2k	grok-4-1-fast-reasoning	0.73 [0.48, 0.89]	0.27 [0.11, 0.52]	7.7	62.0	finished (11/15)

ReAct Basic.

Table 31: ReAct+`apply_patch` baseline (15 runs per model per condition). The patcher follows a `git apply`-style strategy: line numbers are hints and hunks are placed by matching context. Paragraph-level edits succeed reliably, but structure-heavy tasks fail; `parse_error_limit` reflects repeated JSON violations, `tool_error_limit` captures malformed patches.

Task	Size	Model	Success (WS)	Side-Effect (WS)	Steps	Time (s)	Finish
code_visibility	small	gpt-4.1	0.00 [0.00, 0.20]	0.60 [0.36, 0.80]	7.4	13.8	finished (8/15)
code_visibility	small	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.27 [0.11, 0.52]	4.7	91.5	tool_error_limit (11/15)
code_visibility	medium	gpt-4.1	0.00 [0.00, 0.20]	0.40 [0.20, 0.64]	6.7	11.1	tool_error_limit (10/15)
code_visibility	medium	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.13 [0.04, 0.38]	5.3	110.3	tool_error_limit (12/15)
contextual_insert	2k	gpt-4.1	0.87 [0.62, 0.96]	0.13 [0.04, 0.38]	5.1	6.5	finished (15/15)
contextual_insert	2k	grok-4-1-fast-reasoning	0.67 [0.42, 0.85]	0.33 [0.15, 0.58]	4.7	58.7	finished (15/15)
doc_relocate	10k	gpt-4.1	0.00 [0.00, 0.20]	0.80 [0.55, 0.93]	9.6	19.2	parse_error_limit (7/15)
doc_relocate	10k	grok-4-1-fast-reasoning	0.40 [0.20, 0.64]	0.60 [0.36, 0.80]	7.1	227.7	parse_error_limit (9/15)
doc_relocate	20k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	8.1	19.6	parse_error_limit (12/15)
doc_relocate	20k	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.53 [0.30, 0.75]	10.3	416.5	parse_error_limit (10/15)
doc_reorder	10k	gpt-4.1	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	9.3	23.7	parse_error_limit (15/15)
doc_reorder	10k	grok-4-1-fast-reasoning	0.20 [0.07, 0.45]	0.07 [0.01, 0.30]	7.9	378.0	parse_error_limit (10/15)
doc_reorder	20k	gpt-4.1	0.00 [0.00, 0.20]	0.07 [0.01, 0.30]	9.1	14.7	parse_error_limit (14/15)
doc_reorder	20k	grok-4-1-fast-reasoning	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	10.8	614.4	parse_error_limit (13/15)
latex_cite_update	10k	gpt-4.1	0.27 [0.11, 0.52]	0.73 [0.48, 0.89]	6.7	8.4	finished (10/15)
latex_cite_update	10k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	5.9	92.2	finished (14/15)
latex_cite_update	20k	gpt-4.1	0.20 [0.07, 0.45]	0.80 [0.55, 0.93]	7.4	9.9	finished (7/15)
latex_cite_update	20k	grok-4-1-fast-reasoning	0.87 [0.62, 0.96]	0.13 [0.04, 0.38]	6.0	93.2	finished (15/15)
latex_math_edit	10k	gpt-4.1	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.7	4.9	finished (15/15)
latex_math_edit	10k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	6.0	32.1	finished (15/15)
latex_math_edit	20k	gpt-4.1	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	5.9	4.9	finished (15/15)
latex_math_edit	20k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.2	28.5	finished (15/15)
latex_xref_update	10k	gpt-4.1	0.33 [0.15, 0.58]	0.67 [0.42, 0.85]	9.9	13.8	finished (6/15)
latex_xref_update	10k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	7.0	103.0	finished (15/15)
latex_xref_update	20k	gpt-4.1	0.47 [0.25, 0.70]	0.53 [0.30, 0.75]	9.6	11.5	finished (11/15)
latex_xref_update	20k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	6.7	86.5	finished (14/15)
para_consolidate	10k	gpt-4.1	0.80 [0.55, 0.93]	0.07 [0.01, 0.30]	5.5	8.3	finished (14/15)
para_consolidate	10k	grok-4-1-fast-reasoning	0.87 [0.62, 0.96]	0.00 [0.00, 0.20]	5.3	148.4	finished (13/15)
para_consolidate	20k	gpt-4.1	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	4.8	6.2	finished (15/15)
para_consolidate	20k	grok-4-1-fast-reasoning	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	5.5	129.6	finished (15/15)
para_edit	2k	gpt-4.1	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	4.0	4.0	finished (15/15)
para_edit	2k	grok-4-1-fast-reasoning	0.93 [0.70, 0.99]	0.07 [0.01, 0.30]	5.2	50.1	finished (15/15)

ReAct + Apply-Patch. Per-task pattern. The ReAct+`apply_patch` variant nearly matches Functional Safety on targeted paragraph editing (97% success, 3% side effects) and paragraph consolidation (83–100% success depending on document size, ≤3% side effects). However, it struggles on structural tasks: code visibility refactors (0% success, 13–60% side effects), section relocation (0–40% success, 53–100% side effects), and section reordering (0–20% success). On LaTeX micro-edits, the apply_patch variant achieves 74% aggregate success with 26% aggregate side effects: simple math-symbol fixes succeed reliably (93–100% with near-zero SE), but multi-location edits (cross-references requiring coordinated `\label`/`\ref` changes: 33–100% success; citation keys: 20–93% success) show high variance across models and produce frequent intra-target side effects where the agent edits the wrong location within the target chapter. The basic variant performs worse across the board, with higher side-effect rates from raw-offset corruption—and the gap is strongly model-dependent: `gpt-4.1` achieves near-0% success on basic tools while `grok` reaches 60–100% on most tasks, suggesting that extended reasoning can compensate for raw-offset arithmetic. The `apply_patch` tool narrows this gap via context matching, converging on easy tasks but leaving a residual divergence on LaTeX multi-location edits where precise document navigation still matters (cross-references: `grok` 93–100% vs. `gpt-4.1` 33–47%; citations: 87–93% vs. 20–27%). This gradient—success on simple localized edits, degradation on multi-location edits, failure on structural operations—demonstrates that tool-based editing without hierarchy awareness is insufficient for tasks that require coordinated changes or structural reorganization.

5.10 Additional Method Comparisons

We include two additional supplementary baselines: guardrail+regeneration and constrained JSON span edits. Both are run for 15 trials per condition with `gpt-4.1`. Rate columns report proportions with 95% Wilson confidence intervals.

Table 32: Guardrail+regeneration baseline (15 runs per condition; `gpt-4.1`).

Task	Size	Model	Success (WS)	Side-Effect (WS)	Attempts	Time (s)	Finish
code_visibility	small	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.0	18.7	guardrail_limit (15/15)
code_visibility	medium	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.0	19.9	guardrail_limit (15/15)
doc_reorder	10k	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.0	31.3	guardrail_limit (15/15)

Guardrail + Regeneration Baseline.

Table 33: Constrained JSON span-edit baseline (15 runs per condition; `gpt-4.1`).

Task	Size	Model	Success (WS)	Side-Effect (WS)	Attempts	Time (s)	Finish
code_visibility	small	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.0	6.1	applied (15/15)
code_visibility	medium	gpt-4.1	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.0	6.8	applied (15/15)
doc_reorder	10k	gpt-4.1	0.00 [0.00, 0.20]	0.13 [0.04, 0.38]	2.8	33.8	parse_error_limit (13/15)

Constrained Generation Baseline. Guardrail and constrained-generation baselines can reduce some formatting failures, but they still operate over whole-document rewrites or raw offsets and therefore cannot enforce hierarchical locality during execution. This is the main source of their residual side effects relative to Functional Safety.

5.11 Cross-Task Synthesis

To unify the empirical findings across domains, we include a narrative synthesis of success rates, latency, and failure modes:

Functional Safety decomposes edits into typed plans over explicit hierarchy units and executes them with deterministic locality checks. Flat baselines rewrite larger spans directly, while ReAct tool agents apply edits through an observe–think–act loop with span-editing or patch-based tools (Section 5.9). We also compare guardrail and constrained-generation baselines (Section 5.10).

Table 2 summarizes strict success and strict side-effect rates across the six main task families: targeted paragraph editing, contextual insert, code visibility refactor, long-form relocation, paragraph consolidation, and LaTeX micro-edits.

Main trend. Results follow a three-tier pattern. Flat baselines always fail strict success and always introduce side effects—they serve as a naive lower bound showing that full-span regeneration is incompatible with byte-level non-interference. ReAct tool agents succeed on simple localized edits but fail on structural tasks, demonstrating that current agentic editing architectures are insufficient when hierarchy-level operations are required. Functional Safety succeeds across the board by confining edits to hierarchy-aware footprints.

Localized prose edits (ReAct succeeds). For targeted paragraph edits, both Functional Safety and the ReAct+`apply_patch` agent achieve near-perfect results (97% success, 3% SE for ReAct), with both models performing similarly. Contextual insertion is somewhat weaker: ReAct+`apply_patch` reaches 77% aggregate success with 23% side effects, and here `gpt-4.1` is moderately better than `grok` (87% vs. 67% success). These tasks involve single-target, localized changes where a context-matching patch can locate and modify the correct region without disturbing surrounding content. The flat baseline fails only because it rewrites the entire document, introducing whitespace drift.

Moderate-complexity edits (ReAct degrades). For paragraph consolidation in 10k–20k policy briefs, the ReAct+`apply_patch` agent achieves 83–100% success with low side effects ($\leq$3%)—the merge is localized enough that context-matching patches usually work, but occasional matching failures reduce reliability. Functional Safety maintains 100% success by operating on hierarchy-identified paragraph spans rather than context-matching heuristics.

Structural operations (ReAct fails). For code visibility refactors, section relocation, and section reordering, the ReAct+`apply_patch` agent largely fails (0% success on code visibility; 10% on section relocation),

often with significant side effects (35% on code visibility, 73% on section relocation). The aggregate relocation number is driven entirely by `grok`, which reaches 40% success on 10k documents while `gpt-4.1` achieves 0% at both sizes. These tasks require moving, reordering, or reorganizing document structure—operations that exceed the capabilities of context-matching patches operating on linear text. Functional Safety succeeds by decomposing structural operations into typed plan steps (permutations, relocations) executed deterministically over the hierarchy.

LaTeX edits (ReAct shows task-dependent gradient). For localized LaTeX tasks, Functional Safety preserves surrounding content with high reliability in all conditions. The ReAct+`apply_patch` agent shows a fine-grained gradient: simple math-symbol fixes succeed reliably (93–100% with near-zero SE), but cross-reference updates requiring coordinated changes at two document locations (`\label` and matching `\ref`) achieve 33–100% success, and citation-key fixes reach 20–93% success. This variance is strongly model-dependent: on multi-location edits, `grok` reaches 93–100% on cross-references and 87–93% on citations, while `gpt-4.1` achieves only 33–47% and 20–27% respectively. Failures are not clean: the agent frequently edits the wrong location within the target chapter (e.g., replacing the wrong symbol or label), which is detected as an intra-target side effect. In aggregate, ReAct+`apply_patch` achieves 74% success with 26% side effects on LaTeX micro-edits. This confirms that even "localized" edits can exceed tool-agent capabilities when the fix spans multiple locations or requires precise navigation of structured markup.

Model choice × tool design. The interaction between model capability and tool mode is striking. With basic (raw-offset) tools, `gpt-4.1` achieves near-0% success on nearly every task, while `grok` reaches 60–100% on most—suggesting that extended reasoning helps compute correct character offsets. The `apply_patch` tool largely neutralizes this gap by replacing offset arithmetic with context matching: both models converge on easy tasks (math edits ∼97%, paragraph edit ∼97%, consolidation ∼90%) and impossible tasks (code visibility 0%). The residual gap on LaTeX multi-location edits (cross-references: `grok` 93–100% vs. `gpt-4.1` 33–47%; citations: 87–93% vs. 20–27%) indicates that precise document navigation still benefits from extended reasoning even when context matching handles the mechanics of patch application. Functional Safety eliminates model-dependent variance entirely: both models achieve 100% success across all single-document tasks.

Worker-format failures and guardrails. Some models occasionally return structurally problematic worker outputs (e.g., wrapper artifacts or multi-paragraph output when one paragraph is required). We treat these as failures rather than silently rewriting outputs. Bounded worker-output guardrails (retry=1, enabled by default) reduce these failures; ablations are reported in Appendix E.

Multi-file refactors. For Python and LaTeX multi-file reconstruction, we report identity metrics, import and signature deltas, tests (when available), and latency. Exact byte identity is hardest in open-layout and imports-top variants, but test-oriented behavior preservation remains high in several settings. The multi-file tables should therefore be read jointly because identity, deltas, and tests capture different aspects of refactor quality.

6 Related Work

Research at the intersection of hierarchical modeling, structured editing, and LLM safety spans several partially overlapping communities. We focus on five strands of work that illuminate different facets of the problem: long-context and hierarchical Transformers, structure-aware code representations, planner–executor and tool-using LLMs, safety and guardrail frameworks, and efficient Transformer variants. Our aim is not to survey each area exhaustively, but to clarify why existing approaches do not provide the symbolic, functionally constrained editing architecture developed in this paper.

6.1 Long-Context and Hierarchical Transformers

A natural response to the structural limitations of LLMs is to extend context length or incorporate multi-scale attention. Models such as Longformer and BigBird introduce sparse attention patterns that allow tens of thousands of tokens to be processed efficiently while maintaining strong document-level performance Beltagy et al. (2020); Zaheer et al. (2020). Hierarchical attention variants further segment inputs into sentences or

paragraphs and perform cross-segment attention to capture multi-level structure Pappagari et al. (2019); Chalkidis et al. (2022); He et al. (2024). These architectures can be viewed as neural analogues to the two-level representation discussed in our introduction: one mechanism attends within segments, another across them.

However, these models treat hierarchy as a latent internal structure rather than an explicit symbolic object. They do not explicitly manipulate named units—e.g., "the third paragraph in section 2" or "lemma 4"—which makes it difficult to provide formal assurances that edits remain localized or that protected regions' content remains unchanged. By contrast, the approach in this paper exposes hierarchy explicitly and can route modifications through a symbolic plan interpreted by an executor that enforces structural invariants. Our method is therefore complementary: long-context or hierarchical Transformers may serve as powerful planners, and embedding them in a structure-aware execution framework can provide editing assurances in this setting.

6.2 Structure-Aware Code Models and Tree Encoders

A second line of work incorporates syntax and control-flow information to improve code understanding. Models such as CodeBERT and GraphCodeBERT augment token sequences with data-flow or semantic graphs for code search and summarization Feng et al. (2020); Guo et al. (2020). Tree-based models such as TreeBERT and AST-Transformer go further by encoding abstract syntax trees directly Jiang et al. (2021); Tang et al. (2021). These approaches demonstrate that explicit structure can significantly improve robustness on program-analysis tasks.

Yet even in tree-aware models, editing remains predominantly autoregressive: the model regenerates AST fragments or textual spans, and there is no formal mechanism preventing unintended changes outside the intended region. Moving a function or reordering a block may inadvertently modify formatting, comments, or semantically adjacent code. Our work differs in kind: edits are expressed as symbolic operators over hierarchical nodes, and application is carried out by a deterministic executor that isolates stochasticity to planner-selected regions, preserves byte-for-byte payload outside each step's payload footprint, and confines structural changes to each step's structural footprint under the stated assumptions. This yields stronger refactoring assurances in this setting than are typically available in prior code-model architectures.

6.3 Planner–Executor Architectures and Tool-Using LLMs

Planner–executor decompositions have become central to extending LLM capabilities. ReAct couples chain-of-thought reasoning with tool invocation Yao et al. (2023), while Toolformer shows that LLMs can self-supervise their own use of external APIs Schick et al. (2023). Industrial systems further generalize these ideas through orchestration layers that manage retrieval, function calls, or multi-step workflows.

These systems share a core intuition with our work: language models should propose actions, and a separate component should execute them. However, tool invocations in existing frameworks typically operate at the level of free-form text manipulation or broad task descriptions ("edit this paragraph," "summarize this section"). They do not act over discrete hierarchical units, nor do they provide symbolic edit operators with formally specified invariants. In contrast, our methodology treats the planner's outputs as typed edit plans over a structured hierarchy and constrains worker modules—potentially stochastic themselves—to operate within the regions selected by the planner. A deterministic executor then verifies structure, enforces locality, and applies the permissible changes specified by the plan. We implement ReAct-style baselines—including a `apply_patch` variant mirroring production editor agents—as concrete comparators (Section 5.9). The empirical results show that ReAct agents match Functional Safety on simple localized edits but fail on structural tasks (reordering, relocation), precisely where hierarchy-aware planning and execution become necessary.

6.4 Safety, Guardrails, and Model Editing

A growing literature examines safety vulnerabilities in unconstrained LLM outputs and model updates. Surveys highlight risks associated with jailbreaks, prompt injection, and training-set poisoning, motivating

stronger guardrails and programmable layers between users and foundation models Dong et al. (2024); Zhang et al. (2025); Liu et al. (2025). Parallel work in knowledge editing studies techniques for modifying specific model facts or behaviors, but often uncovers unintended global side effects on generalization and downstream tasks Wang et al. (2024); Hsueh et al. (2024); Youssef et al. (2025).

These approaches generally treat safety as a property of either the model's internal parameters or a wrapper around its input/output channels. They do not typically provide fine-grained structural assurances for document or code transformations. Our approach is orthogonal: by reifying structure into explicit hierarchies and restricting edits to a deterministic executor that applies a typed edit algebra, we obtain functional-style invariants—non-interference, structural preservation, and bounded stochasticity—that operate at the level of text and code transformations themselves. This shifts a portion of safety reasoning from opaque parameter space into a discrete, auditable space of symbolic operations. We also implement guardrail-style regeneration and constrained span-edit baselines to benchmark these tradeoffs (Section 5.10).

6.5 Relation to Efficient Transformers

A rich body of work on *efficient Transformers* explores sparse, local, or hierarchical attention mechanisms to reduce computational cost for long sequences Beltagy et al. (2020); Zaheer et al. (2020); Kitaev et al. (2020); Wang et al. (2020); Gu & Dao (2023). These models compress attention patterns or introduce state-space abstractions to effectively skim long inputs while attending selectively to relevant regions.

Our methodology is complementary. Rather than modifying attention patterns inside the model, we construct an explicit symbolic hierarchy outside the model and can restrict the planner and workers to operate on trimmed or summarized node views. This constitutes an application-level analogue of sparse attention: the model need not process entire documents unless the task calls for it, and execution is hardened by structural constraints.

6.6 Cognitive and Functional Programming Perspectives

Finally, our work draws on two bodies of ideas rarely combined in LLM research: hierarchical models of human cognition and the design principles of functional programming. Cognitive theories emphasize chunking, multi-level representation, and compositional reasoning when humans manipulate complex artifacts such as proofs or codebases. Functional programming emphasizes purity, composability, and explicit control of side effects.

The architecture developed in this paper synthesizes these perspectives by (i) representing documents and programs as explicit hierarchies of semantic units and (ii) enforcing that edits occur through a deterministic executor that isolates model stochasticity—both from the planner and from worker modules—within carefully bounded regions. To our knowledge, no existing LLM framework integrates symbolic hierarchical representations with a formally specified, structure-constrained execution layer. Filling this gap is the central motivation of the present work.

7 Discussion

The proposed architecture unifies insights from hierarchical cognition, functional programming, and contemporary LLM design into a structured framework for controlled editing and reasoning. By introducing explicit symbolic units and a deterministic executor, the system aims to address limitations of token-level generation—particularly structural drift, collateral modifications outside the intended scope, and the lack of formal assurances about edit locality in many settings.

At a conceptual level, the architecture bridges human-like hierarchical reasoning with machine representations. Humans naturally operate over multi-level abstractions—paragraphs, sections, arguments, lemmas—rather than individual symbols. By mirroring this structure explicitly, the system enables models to reason and act over units that are meaningful both semantically and structurally. This alignment improves coherence in transformations and provides a substrate for richer forms of reasoning.

From a systems perspective, the separation between planning and execution introduces a principled notion of constrained editing. The planning stage performs high-level symbolic reasoning, while the executor enforces safety properties through purity and determinism in this formulation. This mirrors design principles in functional programming, where side effects are controlled by isolating mutation behind explicit interfaces. Such modularity improves reliability and facilitates auditing, verification, and interpretability.

The empirical comparison with ReAct-style tool agents clarifies where current agentic editing architectures fall short. ReAct agents succeed on simple, localized edits where context-matching patches can identify the correct region, but degrade on multi-location changes and fail on structural operations that require coordinated moves or reordering. This gradient—from success on easy edits to failure on structural tasks—is not a limitation of any particular model but of the flat, linear representation over which the agent operates. Hierarchy-aware planning addresses this directly: by decomposing edits into typed operations over explicit structure, the architecture handles structural tasks that are fundamentally out of reach for context-matching patches.

The non-interference property has implications beyond single-agent editing. Just as functional purity enables safe parallelism in languages like Haskell and Erlang, the locality properties proven here can enable multiple agents to operate concurrently on disjoint regions of a shared document without coordination overhead. If agents A and B target non-overlapping node sets $T_A \cap T_B = \emptyset$, their executions are independent in the formal model: neither is modeled as able to observe or corrupt the other's work. This opens possibilities for collaborative multi-agent editing systems where safety is enforced by construction rather than through locking or serialization.

The framework is agnostic to how hierarchies are constructed. Documents with explicit syntactic structure—Markdown, LaTeX, code in any language with a parser—support deterministic hierarchy extraction in well-formed cases. For unstructured prose or scanned documents, an LLM may infer logical boundaries, effectively adding a stochastic hierarchy-induction step before planning begins. Importantly, the safety theorems apply once a well-formed hierarchy is obtained: errors in hierarchy construction affect task success but not the executor properties in the formal model. This separation allows the same formal framework to accommodate both syntactic parsing and learned structure induction, with the proviso that hierarchy quality becomes an additional dependency in the latter case.

The architecture also has implications for AI safety. Existing methods for constraining model behavior often operate post hoc—inspecting or evaluating outputs after generation. In contrast, the symbolic executor enforces constraints during the transformation process itself, so that safety properties such as locality, payload non-interference, and structural soundness hold by construction in the formal model rather than by inspection. When a well-formed plan is executed, payload side effects outside each step's payload footprint (and thus outside the plan-level union) are shown to be absent in the formal model under the stated assumptions; when a malformed plan is submitted, the executor rejects it and performs no modification. This shift from reactive to proactive safety design provides a rigorous, conditional foundation for high-stakes applications, where the distinction between "usually safe" and formally bounded under stated assumptions is material.

Finally, the framework raises new research questions. How can hierarchical representations be induced automatically and optimally? How expressive should the edit algebra be across domains? Can symbolic plans be optimized or verified beyond local invariants? And how does this architecture interact with emerging long-context and retrieval-based systems? Exploring these questions may lead to LLMs that not only generate text but also manipulate structured knowledge with the precision and reliability found in human reasoning.

8 Limitations

Despite its advantages, the proposed architecture also has several limitations and open challenges. First, it assumes access to a reliable mechanism for constructing hierarchical representations. Many structures—such as sections, paragraphs, or code functions—can be identified through heuristics or existing parsers, but more abstract or noisy domains may call for sophisticated analysis pipelines whose correctness becomes an additional dependency.

Second, the executor operates over a predefined edit algebra whose expressiveness constrains what the model can modify. This restriction is intentional for safety reasons, but it may limit applicability in domains that call for very fine-grained edits or whose structure does not decompose cleanly into discrete nodes.

Third, the multi-stage pipeline introduces computational overhead. The hierarchy extraction stage and symbolic planning step involve additional computation relative to direct generation, and scaling the approach to extremely long inputs may therefore call for careful engineering.

Fourth, the architecture depends on the planning stage's ability to generate coherent and correct symbolic operations. While the executor enforces local safety invariants, it does not attempt to correct high-level conceptual errors in the plan itself. Empirically, the hardest cases are multi-operation plans and structural LaTeX relocation: in stress-test settings, we observe reduced success and occasional side effects when the planning stage selects an incorrect scope or omits an operation. We also see safe no-ops (missing inserts) in contextual insertion when the worker violates a narrow output contract (e.g., emitting multiple paragraphs for a single-paragraph insert), causing the executor to reject and fail closed; bounded worker-output guardrails (enabled by default; retry=1) can mitigate such format-induced failures. These results underscore a central boundary condition: locality is formally bounded with respect to the plan, so functional safety with respect to the user's intent is contingent on planning-stage accuracy.

Finally, the architecture does not eliminate the need for broader safety measures. It is best viewed as complementing, rather than replacing, work on interpretability, robust evaluation, and adversarial testing. Full integration into safety-critical systems would still call for additional safeguards beyond the structural assurances discussed here.

8.1 Future Work

Several promising directions emerge from this framework. First, implementing the full architecture in small- or medium-scale LLMs would allow empirical evaluation of the tradeoffs between symbolic planning and autoregressive generation, particularly for long-context editing tasks. Second, automating hierarchy induction remains an open challenge: many domains provide natural structural boundaries, but others may call for learned or hybrid parsing mechanisms capable of identifying meaningful units without manual heuristics.

Third, enriching the edit algebra with domain-specific operations could further expand the system's applicability while preserving safety assurances. We already demonstrate semantic reorganization, cross-referencing, and multi-document transformations; future work is to generalize these capabilities into broader operator families and richer domain-specific semantics. Fourth, integrating the architecture with retrieval-based or memory-augmented models may enable more sophisticated reasoning over distributed or external knowledge sources.

In our current implementation, the router receives fixed-size previews of each hierarchy node. An appealing extension is to use an adaptive scanning scheme in which the model reads sections incrementally and is periodically asked whether it has seen enough to make a structural decision. If the answer is yes, scanning stops early; otherwise, additional tokens are streamed. This could further reduce context length and latency, especially for long or repetitive sections.

9 Conclusion

This work describes a Functional Safety approach for large language models, separating analysis, planning, and execution into distinct components. We prove a Deterministic Safety theorem (Theorem 2) showing that, under the formal model and stated assumptions, valid symbolic plans are payload-side-effect-free outside each step's payload footprint (and thus outside the plan-level union), and extend this to a Conditional Functional Safety lemma (Lemma 3) that relates system reliability to planner correctness and worker-output conformance. By introducing explicit symbolic units and delegating structural edits to a deterministic executor, the approach addresses limitations of flat token-level generation with formal properties and empirical evidence of side-effect reduction, structural consistency, and execution invariants.

The multi-stage pipeline leverages the strengths of modern LLMs while constraining their weaknesses. The host-built hierarchy provides the data needed for symbolic planning; the router emits typed operations; and the executor enforces invariants as it moves or edits sections. Empirically, Functional Safety achieves 100% strict success with zero side effects across all single-document tasks, improving on a ReAct-style tool agent—representing current agentic editor practice—that succeeds on simple localized edits but degrades on multi-location changes and fails on structural operations. This division improves reliability in tasks such as document editing, code refactoring, and structured reasoning, and it opens a broader path toward functional safety in AI systems.

More broadly, the procedure connects insights from cognitive science and functional programming with ongoing efforts in AI safety, interpretability, and agentic LLMs. By shifting from implicit to explicit hierarchy, and from direct generation to verifiable transformation, it points toward systems that manipulate structured knowledge with greater reliability and control.

References

Yuntao Bai et al. Training a helpful and harmless assistant with RLHF. *arXiv:2204.05862*, 2022.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Ilias Chalkidis, Anders Søgaard, Joachim Bingel, and Steffen Eger. An exploration of hierarchical attention transformers for efficient long document classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

Yuxin Dong et al. Safeguarding large language models: A survey. *Artificial Intelligence Review*, 2024. arXiv:2406.02622.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*, 2020.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *arXiv preprint arXiv:2312.00752*, 2023.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, et al. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Haoyu He, Markus Flicke, Jan Buchmann, Iryna Gurevych, and Andreas Geiger. Hdt: Hierarchical document transformer. In *Conference on Language Modeling (COLM)*, 2024.

Chih-Hao Hsueh et al. An in-depth exploration of pitfalls of knowledge editing in large language models. In *Findings of the Association for Computational Linguistics: EMNLP*, 2024.

Xinyun Jiang, Linjie Li, Shafiq He, et al. TreeBERT: A tree-based pre-trained model for programming language. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2021.

Saurav Kadavath et al. Language models (mostly) know what they know. *arXiv:2207.05221*, 2022.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

Shiyu Liu et al. A comprehensive survey on safety evaluation of large language models. *arXiv preprint arXiv:2506.11094*, 2025.

Long Ouyang et al. Training language models to follow instructions with human feedback. *NeurIPS*, 2022.

Raghavendra Pappagari, Nick Rossenbach, Ramon Sanabria, and Geoffrey Zweig. Hierarchical transformers for long document classification. In *Proceedings of Interspeech*, 2019.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Maryam Komeili, Pedro Vassallo, Mike Yu, Alessandro Sordoni, Thomas Scialom, and Edouard Grave. Toolformer: Language models can teach themselves to use tools. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhiyu Tang, Zhe Li, and et al. AST-Transformer: Encoding abstract syntax trees efficiently for code summarization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. In *Advances in Neural Information Processing Systems*, 2020.

Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. Knowledge editing for large language models: A survey. *ACM Computing Surveys*, 2024.

Jason Wei et al. Emergent abilities of large language models. *arXiv:2206.07682*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, and et al. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Paul Youssef, Zhixue Zhao, Daniel Braun, Jörg Schlötterer, and Christin Seifert. Position: Editing large language models poses serious safety risks. *arXiv preprint arXiv:2502.02958*, 2025.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, 2020.

Rui Zhang et al. On large language models' safety, security, and privacy. *Journal of Information Security and Applications*, 2025.

A Procedure Details

This appendix reports only the core orchestration procedures used by the method: (1) typed plan synthesis/validation and (2) deterministic execution with worker-output guardrails and invariant checks. Auxiliary engineering variants (e.g., grouped planning and open-ended layout selection) are described in prose in Section 3.

Procedure 1 Typed Plan Synthesis and Validation

Require: Hierarchy handle H, user instruction q, optional task metadata m
Ensure: Well-typed symbolic plan P or error

$(S, \text{prefix}, \text{suffix}) \leftarrow$ ExtractSectionInfos(H)
Optionally run lightweight locator to produce disambiguation hints L
Optionally compute sparse content cues H_s from q and section summaries
if H_s is marked ambiguous and ambiguity guard is enabled **then**
  return Error(`AmbiguousTarget`)
end if
Construct router prompt with allowed operators $\mathcal{O}$, summaries, L, optional H_s, and instruction q
$(\text{raw_plan}, \text{meta}) \leftarrow$ LLMComplete$(rllm, \text{prompt})$
Parse raw_plan into candidate plan object
if parse fails **then**
  return Error(`InvalidPlanEncoding`)
end if
if candidate delegates pure permutation **then**
  $\pi \leftarrow$ ExtractPermutation(candidate)
  if π is not a permutation of S **then**
    return Error(`InvalidPermutation`)
  end if
  return $(\text{Move}(\pi), \text{meta})$
end if
$P_{\text{raw}} \leftarrow$ ordered operations extracted from candidate
for each operation $o \in P_{\text{raw}}$ **do**
  if o is $\text{Move}(\pi)$ and π is not a permutation of S **then**
    return Error(`InvalidPermutation`)
  else if o targets nonexistent node IDs or invalid positions **then**
    return Error(`InvalidTarget`)
  else if o contains invalid label/type fields **then**
    return Error(`InvalidOperator`)
  end if
end for
$P \leftarrow$ NormalizePlan(P_{raw})
return (P, meta)

Procedure 2 Deterministic Execution with Guardrails and Invariants

Require: Hierarchy handle H, well-typed plan P, optional intended structural footprint T
Ensure: Updated document text X' or error

```
X' ← H.doc_text
for each operation o in P do
  Compute per-step structural footprint F_struct(o) and payload footprint F_payload(o)
  if T is provided and F_struct(o) ⊈ T then
    return Error(InvalidFootprint)
  end if
  if o is Move then
    Apply deterministic reordering/reparenting update
  else if o is Edit then
    Extract target content (crop if specified)
    if replacement content is not provided in plan then
      Call worker with scoped context
      Normalize wrapper artifacts (fences/labels/quotes)
      if guardrails reject output and retry budget remains then
        Retry worker call
      end if
      if guardrails still reject output then
        return Error(WorkerOutputInvalid)
      end if
    end if
    Splice accepted content into target span
    Re-extract affected subtree when structural refresh is required
  else if o is Insert then
    Materialize inserted content (worker call only if content omitted)
    Insert new node at validated position
  else if o is Delete then
    Remove node and descendants
  else if o is Annotate then
    Update node label/metadata without changing payload
  else
    return Error(UnknownOperation)
  end if
  if payload outside F_payload(o) changed then
    return Error(PayloadViolation)
  end if
  if structural invariants fail (nesting/acyclicity/non-overlap) then
    return Error(InvariantViolation)
  end if
end for
return X'
```

Execution note. The executor operates over normalized typed operations and fails closed on invalid plans, invalid worker outputs (after bounded retries), or invariant violations. This is the mechanism behind the step-wise non-interference claims in Section 4.

B Local Model Experiment Details

This appendix provides detailed experimental results for local models on the smaller tasks. These experiments illustrate that Functional Safety can make local models more usable editors on these tasks by confining edits to the intended regions. Local results are reported in Tables 34, 35, and 36.

B.1 Why Larger Models Produce More Unintended Edits

A consistent empirical finding in baseline settings is that larger models can exhibit *higher* unintended-edit rates than smaller models, even at low temperatures. This counterintuitive result arises from several reinforcing mechanisms. Larger models have sharper, more peaked logit distributions, making greedy decoding amplify any miscalibrated confidence (Wei et al., 2022; Kadavath et al., 2022). They also internalize stronger priors about document coherence from instruction tuning and RLHF, which reward helpful, polished, and detailed responses (Ouyang et al., 2022; Bai et al., 2022). When given full document context, these priors manifest as unsolicited global rewriting—the model confidently "improves" regions that should remain untouched.

Functional Safety can mitigate these failure modes by limiting planner and worker context rather than exposing the full document. In our implementation, the router sees section titles and short previews (plus paragraph IDs and scoped paragraph previews when paragraph-level edits are enabled), and workers can receive the targeted section (or a cropped paragraph when configured) instead of the full text. The system converts a "global edit" problem into a "local decision" problem. In these local-model runs, we did not observe edits outside the task-defined target region, including for larger local models with strong coherence priors.

B.2 Targeted Paragraph Editing

Table 34: Targeted Paragraph Editing (local models, $T = 0.3$). In these runs, no content side effects were observed while preserving task success.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
llama3:8b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.96
llama3:8b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.63
qwen2.5:14b (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	3.46
qwen2.5:14b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.89
gemma3:27b (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	6.40
gemma3:27b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.98

B.3 Contextual Insert

Table 35: Contextual Insert (local models, $T = 0.3$). Functional Safety inserts the summary paragraph while preserving all other content in these runs; `llama3:8b` succeeds in 15/15 runs.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
llama3:8b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.26
llama3:8b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.74
qwen2.5:14b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	3.30
qwen2.5:14b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.03
gemma3:27b (Flat)	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	6.32
gemma3:27b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.15

B.4 Code Visibility Refactor

Table 36: Code Visibility Refactor (local models, small classes, $T = 0.3$). Functional Safety preserves method bodies while enforcing private-method grouping in these runs.

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
llama3:8b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	1.58
llama3:8b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.10
qwen2.5:14b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	3.20
qwen2.5:14b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.65
gemma3:27b (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.28
gemma3:27b (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.73

C Hierarchy Extraction

This appendix reports only the hierarchy-extraction components that are central to the empirical claims in the paper. For structured formats (Markdown, LaTeX, Python, and related code formats), we use deterministic parsers and syntax-driven span assignment. For weakly structured documents, we use the two-pass LLM-assisted extraction procedure below while keeping paragraph spans deterministic.

Whitespace policies for lossless rendering and strict side-effect accounting are described in Appendix D.1.

C.1 LLM extraction diagnostics on structured documents

We compare LLM-extracted hierarchies against deterministic parsers on a small structured fixture set (Markdown and LaTeX, four documents) with 15 repetitions per model. Table 37 reports success counts, span F1 at IoU 0.9 for top-level sections and paragraphs, best-IoU averages, and mean latency. Across models, LLM extraction is 4–5 orders of magnitude slower than host parsing. `grok-4-1-fast-reasoning` yields the strongest section alignment, while both models remain well below deterministic parsers on span F1. These results underscore the latency cost and span fragility of LLM-first hierarchy extraction on structured inputs.

Table 37: LLM hierarchy extraction vs. deterministic parsing on structured fixtures (15 repetitions per model).

Model	Format	Docs	Sec F1@0.9	Sec IoU	Para F1@0.9	Para IoU	LLM mean (s)	Host mean (s)
gpt-4.1	overall	60/60	0.21	0.69	0.38	0.67	2.36	0.00012
gpt-4.1	latex	30/30	0.50	0.89	0.00	0.40	2.60	0.00012
gpt-4.1	markdown	30/30	0.04	0.56	0.67	1.00	2.12	0.00012
grok-4-1-fast-reasoning	overall	60/60	0.46	0.82	0.38	0.67	32.07	0.00012
grok-4-1-fast-reasoning	latex	30/30	0.49	0.89	0.00	0.40	30.93	0.00012
grok-4-1-fast-reasoning	markdown	30/30	0.44	0.78	0.67	1.00	33.21	0.00012

C.2 Unstructured documents

Two-pass LLM extraction with canonical paragraphs. For unstructured documents we use a two-pass LLM pipeline but keep paragraph boundaries deterministic. Pass 1 extracts paragraph anchors from overlapping line windows, and pass 2 groups those paragraphs into sections. To enforce the "no newlines" requirement, we discard LLM paragraph boundaries and instead canonically segment the source text into blank-line paragraph blocks, then snap section spans to those canonical paragraph boundaries. The resulting hierarchy is thus an overlay on the original text: the LLM provides grouping and level metadata, while paragraph spans remain byte-identical to the source.

Procedure 3 Unstructured Two-Pass Hierarchy Extraction with Canonical Paragraphs

Require: Unstructured text X
Ensure: Hierarchy H
Split X into overlapping line windows
Pass 1: For each window, prompt the LLM for paragraph start/end anchors
Resolve anchors to full-text spans; dedupe and drop overlaps
Compute canonical paragraph spans by blank-line segmentation
Replace LLM paragraph spans with canonical paragraph spans
Pass 2: Slide over paragraph windows; prompt LLM to group paragraph IDs into sections
Convert each section group to a span covering its paragraph range
Snap section spans to cover their child paragraphs; drop overlaps
Fill uncovered paragraphs with synthetic sections if needed
Build hierarchy nodes from sections and paragraphs; refresh content from spans
return H

We evaluate localized paragraph edits on nine research papers (one repetition each, `gpt-4.1`, anchors-two-pass, 150/50 line windows). The target paragraph is selected deterministically from the raw text (hash index over blank-line blocks). We then (i) execute a paragraph-level replace using the hierarchy and (ii) run a full-document baseline with the same replacement, configured to return the full document inside tags with an explicit output-token cap sized to the document length (capped at 50k). Success requires an exact match to the expected output with zero outside-target diffs. Table 38 reports per-document results. With canonical paragraphs the hierarchy-based edit succeeds in all 9/9 documents, while the full-document baseline returns no exact matches (two runs exceed the model's 32k completion limit and are rejected, and the remaining runs either truncate or introduce extra edits).

Table 38: Unstructured localized edit pilot with canonical paragraph boundaries (9 papers, 1 repetition, `gpt-4.1`). The full-document baseline is capped at 50k output tokens, but OpenAI enforces a 32k completion limit, so the longest papers cannot return a full document.

Document	Target chars (non-ws)	Chunk fails	Hierarchy edit	Outside diff (hier)	Baseline edit	Outside diff (baseline)
CodeBERT	3,393	0	1	0	0	37,988
GraphCodeBERT	3,182	0	1	0	0	59,578
Hierarchical Transformers	3,729	0	1	0	0	20,656
Knowledge Editing Survey	4,363	4	1	0	0	—
Longformer	3,949	3	1	0	0	62,372
Position Editing Safety Risks	4,311	1	1	0	0	67,687
ReAct	1,559	3	1	0	0	108,035
Safeguarding LLMs Survey	7,206	4	1	0	0	—
Toolformer	3,575	2	1	0	0	64,101

D Whitespace Handling

D.1 Whitespace handling and canonicalization

Whitespace is treated as payload in the executor and is never normalized or trimmed during execution. The only optional modification is paragraph-separator insertion when adjacent paragraph nodes would otherwise merge after structural edits. Whitespace is not directly addressable as a standalone target; extra inter-paragraph whitespace is treated as content and can be removed explicitly (e.g., "remove the extra space between P2 and P3"). This section documents the representation and its tradeoffs.

Canonicalization and payload ownership. We use a canonicalization $C(X)$ that preserves document bytes: render the hierarchy deterministically and require the resulting text to match the input (lossless render). For a node n with ordered children $c_1, \ldots, c_k$, the rendered text is

$$\text{render}(n) = \text{prefix}(n) \parallel \text{render}(c_1) \parallel \Delta_1 \parallel \text{render}(c_2) \parallel \cdots \parallel \Delta_{k-1} \parallel \text{render}(c_k) \parallel \Delta_k,$$

where prefix(n) is the text before the first child, and each Δ_i is the inter-child or trailing whitespace segment drawn from the original text. We represent each Δ_i as an explicit whitespace node with label `gap`. Gap nodes are non-movable and non-anchorable, and they are included in the payload footprint so any change is tracked. This representation is lossless under $C(X)$ and makes whitespace ownership explicit.

Paragraph separators. Paragraph boundaries are detected by blank lines (`\n[ \t]*\n`). After sequential plan execution (structural edits), if two paragraph nodes become adjacent and no blank line remains between them, the default `preserve_paragraph_boundaries` policy inserts the minimal separator (typically `\n\n`, or `\n` if already newline-terminated). This insertion is deterministic and expands the payload footprint to include the affected paragraph nodes. A strict-raw mode disables this insertion when byte identity of the original input must be preserved, at the cost of possible paragraph merges after moves.

Examples and tradeoffs.

- **Missing trailing separator.** Text: `P1\n\nP2` (no trailing blank line after `P2`). Moving `P2` before `P1` creates an adjacent pair with no separator. With `preserve_paragraph_boundaries`, the executor inserts a minimal blank line and returns `P2\n\nP1\n\n`. In strict-raw mode, the boundary may collapse.

- **Extra blank lines between paragraphs.** Text: `P1\n\n\n\nP2\n\n`. The extra spacing is stored in the gap node between the siblings; moving `P2` elsewhere leaves that extra gap behind, while the paragraph-boundary separator remains enforced at the new location.

- **Leading/trailing whitespace.** Any preamble before the first root and trailing whitespace after the last root are represented as gap nodes so that moving the first/last semantic node does not drag file headers, footers, or trailing padding into a different location.

Safety interaction. Whitespace changes are treated like other payload changes: boundary gap nodes are always included in the payload footprint, and the neighboring boundary leaves are included as a conservative halo because separator placement can touch their local payload. Optional normalization policies (B–D) only affect payload comparison during safety checks; they do not modify stored text. This keeps the executor deterministic and ensures that whitespace-sensitive formats (code, LaTeX) remain byte-identical unless an explicit separator insertion is enabled.

Whitespace halo examples.

- **Leading gap + root insert.** Text begins with a blank-line preamble. Inserting a new root at position 0 leaves the leading gap in place, which effectively reattaches that gap to the new root. The gap node must be part of the payload footprint to avoid a false non-interference failure.

- **Nested subsection + trailing insert.** A section contains a nested subsection; inserting a paragraph after the subsection may transfer the boundary newline from the subsection's last paragraph into the new paragraph's boundary. The boundary leaf (last paragraph inside the subsection) must be inside the footprint halo so reconciliation allows the whitespace shift.

E Supplementary Diagnostics

This appendix reports selected diagnostics referenced in the main text. Tables 39 and 40 report compound-operation diagnostics; Tables 41 and 42 report policy-brief structural permutation; Tables 43, 44, 45, and 46 report LaTeX structural diagnostics; and Tables 47 and 48 report worker-output guardrail ablations.

E.1 Compound Operation Diagnostics

Table 39: Compound operations (10k briefs, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.42
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.20 [0.07, 0.45]	0.20 [0.07, 0.45]	20.83
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.05
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	25.78
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	387.86
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	36.74

Table 40: Compound operations (20k briefs, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.85
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	11.96
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.84
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	27.39
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	405.20
grok-4-1-fast-reasoning (Functional Safety)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	39.85

E.2 Policy Brief Structural Permutation

Table 41: Pure section permutation (10k briefs, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.69
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	23.66
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.92
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.98
grok-4-1-fast-reasoning (ReAct+patch)	0.20 [0.07, 0.45]	0.20 [0.07, 0.45]	0.07 [0.01, 0.30]	0.13 [0.04, 0.38]	378.02
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.15

Table 42: Pure section permutation (20k briefs, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.89
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	14.73
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.01
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.06
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	614.39
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.63

E.3 LaTeX Structural Diagnostics

Table 43: LaTeX section permutation (10k, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.87
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	12.54
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.10
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.45
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	383.21
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.21

Table 44: LaTeX section permutation (20k, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	11.52
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	10.92
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.49
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	5.76
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	337.69
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	3.66

Table 45: LaTeX section relocation (10k, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	10.71
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	22.43
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	4.21
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	7.36
grok-4-1-fast-reasoning (ReAct+patch)	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	267.70
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	12.47

Table 46: LaTeX section relocation (20k, default temperature).

Method	Success Rate (WS)	Success Rate (Strict)	Side-Effect Rate (WS)	Side-Effect Rate (Strict)	Time (s)
gpt-4.1 (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	8.82
gpt-4.1 (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	24.65
gpt-4.1 (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	8.64
grok-4-1-fast-reasoning (Flat)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	6.62
grok-4-1-fast-reasoning (ReAct+patch)	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	244.19
grok-4-1-fast-reasoning (Functional Safety)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	12.70

E.4 Worker Output Guardrails

Some models occasionally violate worker output contracts (for example, wrapper artifacts or invalid paragraph boundaries). We report guardrail ablations here.

Table 47: Worker output guardrails ablation (Functional Safety only; `grok-4-1-fast-reasoning`).

Task	Size	FS Strict Success (Guardrails Off)	FS Strict Success (Default)	FS Strict Side-Effect (Guardrails Off)	FS Strict Side-Effect (Default)	N
Paragraph edit (sections-only)	2k	1.00 [0.80, 1.00]	0.93 [0.70, 0.99]	0.00 [0.00, 0.20]	0.07 [0.01, 0.30]	15
LaTeX relocate	10k	0.93 [0.70, 0.99]	0.87 [0.62, 0.96]	0.07 [0.01, 0.30]	0.13 [0.04, 0.38]	15
LaTeX relocate	20k	0.80 [0.55, 0.93]	0.93 [0.70, 0.99]	0.20 [0.07, 0.45]	0.07 [0.01, 0.30]	15
Compound ops	20k	0.93 [0.70, 0.99]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	15
Compound ops (targeted)	20k	0.93 [0.70, 0.99]	0.80 [0.55, 0.93]	0.07 [0.01, 0.30]	0.07 [0.01, 0.30]	15
Compound ops (sequential)	20k	1.00 [0.80, 1.00]	0.87 [0.62, 0.96]	0.00 [0.00, 0.20]	0.13 [0.04, 0.38]	15

Table 48: Contextual Insert guardrails ablation (Functional Safety only; `gpt-4.1`).

Setting	Success (WS)	Success (Strict)	Side-Effect (WS)	Side-Effect (Strict)	N
FS (guardrails=off)	0.93 [0.70, 0.99]	0.93 [0.70, 0.99]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	15
FS (default: retry=1)	1.00 [0.80, 1.00]	1.00 [0.80, 1.00]	0.00 [0.00, 0.20]	0.00 [0.00, 0.20]	15

F Footprint Computation and Payload Preservation Guarantees

Scope of this appendix. Section 4 defines the formal guarantees. This appendix adds only the concrete computation procedures used by the implementation: (1) step-wise structural footprint computation, (2) step-wise payload footprint computation, and (3) a worked multi-step example.

Operational setup. For a hierarchy R and plan $P = (o_1, \ldots, o_k)$, footprints are computed *per operation* on the evolving hierarchy state. We use $F_{\text{struct}}(o_i)$ for nodes whose structural signature changes in step i, and $F_{\text{payload}}(o_i)$ for nodes whose local payload may change in step i. Plan-level sets are unions across steps:

$$F_{\text{struct}}(P) = \bigcup_i F_{\text{struct}}(o_i), \qquad F_{\text{payload}}(P) = \bigcup_i F_{\text{payload}}(o_i).$$

Minimal whitespace halo note. The payload footprint includes boundary gap nodes and adjacent boundary leaves as a conservative halo so separator canonicalization at edit boundaries is treated as in-scope. This allows expected boundary whitespace movement without weakening non-interference outside the step footprint.

Procedure 4 ComputeStructuralFootprints

Require: Hierarchy R, plan $P = (o_1, \ldots, o_k)$
Ensure: Per-op footprints $\{F_{\text{struct}}(o_i)\}$ and union $F_{\text{struct}}(P)$

```
H ← R; F_struct(P) ← ∅
for each op o_i in order do
    Validate schema and referenced nodes (fail closed on error)
    H' ← SimulateOp(H, o_i)
    F_struct(o_i) ← {n : Sig_struct(n) differs between H and H'}
    F_struct(P) ← F_struct(P) ∪ F_struct(o_i)
    H ← H'
end for
return {F_struct(o_i)}, F_struct(P)
```

Procedure 5 ComputePayloadFootprints

Require: Hierarchy R, plan $P = (o_1, \ldots, o_k)$
Ensure: Per-op footprints $\{F_{\text{payload}}(o_i)\}$ and union $F_{\text{payload}}(P)$

```
H ← R; F_payload(P) ← ∅
for each op o_i in order do
    Initialize F_payload(o_i) from edit targets in o_i
    Add refreshed subtrees induced by o_i (if any)
    Add moved subtrees, inserted nodes, and deleted nodes
    Add boundary gap nodes and boundary leaves (minimal whitespace halo)
    Add parents/ancestors whose local payload can change due to child-list or gap updates
    F_payload(P) ← F_payload(P) ∪ F_payload(o_i)
    H ← SimulateOp(H, o_i)
end for
return {F_payload(o_i)}, F_payload(P)
```

Worked multi-step example (Move + Insert). Move paragraph p from section A to section B, then insert a new paragraph under A. In step 1, $F_{\text{payload}}(o_1)$ includes the moved subtree, boundary halo at source/destination, and affected parents/ancestors. In step 2, $F_{\text{payload}}(o_2)$ includes the inserted node, boundary halo at the insertion site, and affected parents/ancestors. Payload outside each step footprint remains unchanged for that step; payload outside $F_{\text{payload}}(P)$ remains unchanged after both steps.

Complexity. Let N be node count and k be operation count. Worst-case structural scanning is $O(kN)$, with optional $O(N)$ refresh/reconciliation passes depending on execution mode. In practice, edits are usually local, so changed sets are sparse even though the conservative scan is linear.

G Formal Methods Checks

G.1 Bounded model checking of the orchestration loop (TLA+)

In addition to the formal proofs in the main text, we include a small bounded TLA+ model of the hierarchical pipeline's orchestration control loop under nondeterministic worker outputs. The model abstracts a document as a mapping from units to a compact structural token (tracking only a few features relevant to worker-output failure modes, such as wrapper artifacts and top-level heading structure). It models deterministic post-processing (wrapper reapplication and simple header restoration) followed by worker-output guardrails (enabled by default) that can reject malformed outputs and retry once before failing closed.

We model-check a small set of orchestration invariants. For stable cross-referencing, we assign the following invariant IDs (and check them in the TLA+ model):

- **FS-INV-001 (NoSideEffects).** Units outside the explicitly touched set remain byte-for-byte unchanged.
- **FS-INV-002 (NoIntroducedArtifacts).** With worker-output guardrails enabled, an accepted `local_edit` output cannot introduce prompt-wrapper artifacts that were not present in the input (e.g., stray fences, delimiter echoes, or prompt labels).
- **FS-INV-003 (RetryBoundedness).** Worker-output guardrail retries are bounded by configuration; the system fails closed after the last allowed attempt rather than looping.
- **FS-INV-004 (TerminationUnderBounds).** For a finite plan and bounded retries, execution terminates in success or a bounded error state.

We check FS-INV-001 through FS-INV-003 as state invariants, and we check FS-INV-004 as a temporal termination property (under weak fairness).

We check these invariants with the TLC model checker under small, explicit bounds to keep the state space finite while still exercising the guardrail retry and failure semantics. Concretely, we evaluate two bounded configurations: (i) a single-unit case and (ii) a two-unit case. In both, the plan length is bounded to a single operation, guardrails run in `retry` mode with at most one retry per operation, and worker outputs are abstracted to a small finite token space (including a bounded heading-structure feature). TLC finds no counterexample for any of the above properties under these bounds. As with any bounded model checking exercise, these results do not constitute a proof of correctness for unbounded inputs, and they do not model string parsing or the full space of format-dependent extraction behavior; rather, they serve as an additional consistency check on the control-flow logic and guardrail semantics.

Limitations and out-of-scope realism. This model is intentionally abstract. In particular, it does not aim to capture several sources of real-world complexity, including: (i) malformed or non-text provider payloads and low-level parsing of raw worker responses (worker outputs are modeled as abstract post-normalization tokens), (ii) format-specific extraction failures and entity-reconciliation details, (iii) planner accuracy or mis-scoping of plans relative to the user's intent (plans are chosen nondeterministically within the configured bounds), and (iv) the full range of operational failure modes (timeouts, rate limits, and other infrastructure errors). We therefore treat the TLC results as a bounded control-flow consistency check, complementary to the proofs and empirical evaluations rather than a complete verification of the deployed system.