LEDGER: Scaling Agentic Document Editing with Dependency-aware Graph Retrieval

Anonymous ACL submission

Abstract

We introduce LEDGER to tackle the novel context engineering challenge of agentic document editing, where localized edits to *long, structured documents* must be applied efficiently without breaking cross-references or semantic consistency. LEDGER constructs a lightweight dependency graph that explicitly models document structure, including hierarchical organization, explicit references, implicit dependencies, and semantic relationships. For each edit, graph-guided retrieval selects only the necessary context, avoiding full-document processing while preserving consistency. We evaluate LEDGER on a curated benchmark of 1.9k test cases with various document types and lengths, spanning six state-of-the-art models: LEDGER improves consistency from 56% to 76% across all six models and test scenarios while reducing token usage. Notably, LEDGER with low reasoning effort matches baseline performance at high reasoning effort using fewer tokens, showing that explicit dependency representations can partially substitute for expensive internal reasoning in agentic document editing.

1 Introduction

Agentic document editing (e.g., ChatGPT Canvas and Agent Mode in Word) represents a new interaction paradigm for working with long, structured documents. Instead of manually editing content, users issue high-level natural language instructions such as "clarify the experiment description," "update terminology," or "revise the tone of Section 4", while an AI agent plans and executes the concrete document modifications. This paradigm mirrors recent code agents, where users specify intent and the agent handles low-level edits, refactoring, and consistency checks. However, unlike code, documents lack formal semantics, executable structure, or compiler-level guarantees, making agentic document editing fundamentally more challenging. For example, a simple terminology change such as renaming "Microsoft" to "MS" in one section—requires identifying and updating multiple surface forms (Microsoft, MS, MSFT) across the document without a syntactic anchor, turning consistency into a pattern recognition problem rather than a deterministic replacement. Stylistic edits introduce even subtler dependencies. Revising a paragraph from passive to active voice requires careful alignment with its adjacent paragraphs to avoid abrupt stylistic breaks. Moreover, once a stylistic modification is introduced, its downstream impact must be considered: related sections elsewhere in the document may now violate the intended global style and require corresponding updates, even if they are not directly adjacent or explicitly referenced. Cross-references further complicate this process: revising a figure caption may invalidate descriptions in distant paragraphs that refer to "the linear scaling shown in the figure" without explicit citations.

These loose dependencies create a consistency challenge that is fundamentally a context engineering problem in agentic document editing. In particular, preserving consistency in *long documents* requires the agent to reason over structural and semantic dependencies across the document, yet naively providing the full document as context for every edit is both inefficient and counterproductive: beyond incurring prohibitive token cost and latency as document length grows, full-document processing can distract the model from the most relevant regions, diluting attention and increasing the risk of unfocused or incorrect edits. Retrieval-based editing improves efficiency by selecting a subset of context, but existing approaches primarily rely on topical similarity and frequently miss structural dependencies critical for consistency, such as explicit cross-references, implicit mentions, or downstream sections that depend on the edited content. As a

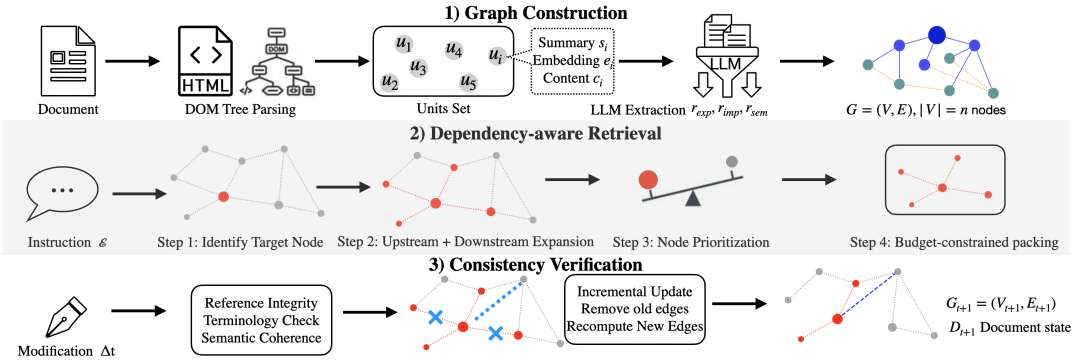

Figure 1: Overview of LEDGER. (1) Graph construction converts documents to DOM trees, extracts units, and applies LLM-based extraction protocols to create a dependency graph with four edge types. (2) Dependency-aware retrieval identifies target nodes, expands to dependencies through graph traversal, prioritizes by edge type, and packs within token budget. (3) Consistency verification checks reference integrity, terminology consistency, and semantic coherence after modifications, then updates the graph incrementally.

result, agentic document editing faces a persistent tension: efficient partial editing risks inconsistency, while comprehensive context provision is expensive and can impair the agent's ability to focus on the most consequential dependencies.

We introduce LEDGER (see illustration in Figure 1), a framework that addresses this context engineering bottleneck through explicit dependency-aware graph retrieval. The core insight is that structured documents exhibit dependency patterns that can be extracted once and queried efficiently. LEDGER constructs a lightweight semantic graph where nodes represent document elements and typed edges encode explicit references ("Theorem 2"), implicit connections ("this result"), and semantic relationships (conceptually related content). For each edit, graph traversal identifies relevant dependencies: both upstream context (what the target relies on) and downstream impacts (what depends on it). This retrieval is selective, gathering only necessary elements within token budgets. After modifications, consistency verification validates structural invariants, ensuring that references resolve correctly, terminology remains consistent, and semantic relationships are preserved. The graph is then updated incrementally to reflect the changes.

We evaluate LEDGER across the curated 1.9k test cases spanning six state-of-the-art models (Claude 4.5 Haiku/Sonnet/Opus, GPT-4.1/5/5.2). LEDGER achieves 76% consistency versus 56% baseline while maintaining constant token usage (1,500 tokens per edit regardless of document size), achieving up to 85% reduction versus full-document regeneration in scalability tests. Critically, LEDGER with low reasoning effort matches baseline performance at high reasoning effort while using fewer tokens, suggesting explicit dependency representations can substitute for expensive internal reasoning. This has broad implications for designing efficient agentic systems operating on structured data such as coding agent.

Our contributions are: **(1)** A dependency-aware framework that applies explicit semantic graphs to document editing, resolving the context engineering bottleneck through typed dependency extraction and graph-guided retrieval; **(2)** A curated benchmark for evaluating consistency in agentic document editing, comprising 1.9k test cases across diverse document types, edit scenarios, and complexity levels; **(3)** Evidence that explicit structural guidance substitutes for reasoning effort, establishing a design principle for efficient agentic systems operating on structured artifacts such as natural language documents.

2 Related Work

Context Engineering for LLM Agents. Managing context for large language models on extensive documents presents fundamental scalability challenges (Zhang et al., 2025). Full-context methods maintain comprehensive awareness (An et al., 2024) but face prohibitive scaling costs. Retrieval-augmented generation (Lewis et al., 2020; Laban et al., 2024) and efficient attention mechanisms (Lai et al., 2025; Martins et al., 2020; Song et al., 2024; Hooper et al., 2025; Liu et al., 2024a) reduce costs through semantic search and computational optimization, but struggle with structural dependencies beyond topical similarity. A section referencing "Figure 3" exhibits structural connections

that embeddings cannot reliably capture. Memory systems (Maharana et al., 2024; Wang et al., 2023; Xiao et al., 2024) maintain interaction history rather than artifact structure, while long-context models (Liu et al., 2024b; Jin et al., 2024) lack explicit dependency tracking. Code editing systems (Chen et al., 2023; Wang et al., 2024; Shinn et al., 2023) employ syntax-aware retrieval but target narrow domains. Our work differs by explicitly representing structural dependencies through typed graph edges, enabling constant time retrieval with comprehensive dependency awareness.

Graph-Based Representations for Structured Tasks. Graph structures enable dependency tracking across domains: abstract syntax trees for program analysis (Allamanis et al., 2018; Guo et al., 2021), knowledge graphs for semantic coherence (Yasunaga et al., 2021; Zhang et al., 2022), and directed acyclic graphs for workflow dependencies (Huang et al., 2022; Chen et al., 2021). Graph neural networks (Kipf and Welling, 2017; Hamilton et al., 2017; Xu et al., 2019) provide learning architectures for such data. We synthesize these insights for agentic editing, demonstrating that explicit dependency graphs enable efficient context selection and automated consistency verification. Critically, our experiments show that structural guidance substitutes for expensive internal reasoning: cheaper inference modes match or exceed quality achievable through costly extended reasoning alone. This reasoning substitution effect, validated across six models and multiple domains, establishes a design principle for efficient agentic systems operating on structured data.

3 LEDGER

We formalize document editing as an iterative process where an AI agent receives edit requests and applies modifications while maintaining structural consistency. A document D consists of units $U = \{u_1, u_2, \ldots, u_n\}$ organized hierarchically, where each unit u_i (section, paragraph, figure, equation) has content c_i, type τ_i, position p_i, and unique identifier id_i. Units connect through three relationship types beyond the hierarchical tree: *explicit references* (direct citations like "Figure 2"), *implicit dependencies* (pronouns and contextual references requiring coreference resolution), and *semantic relationships* (topically related content). At time t, the agent receives edit request $\mathcal{E}_t$ specifying an instruction, target, and scope, and must retrieve relevant context $\mathcal{C}_{\text{context}} \subseteq U$ to generate modifications Δ_t while preserving structural consistency.

The core challenge is selecting context that satisfies three objectives simultaneously: *consistency* (preserving structural invariants), *efficiency* (achieving $|\mathcal{C}_{\text{context}}| = O(1)$ rather than $O(|D|)$), and *quality* (correctly addressing instructions). Full-document methods achieve consistency but incur $O(|D|)$ costs; retrieval-based methods achieve efficiency but miss structural dependencies. For example, editing a theorem in Section 3 requires retrieving Section 7's proof that explicitly references it, despite low topical similarity that embedding-based retrieval cannot capture. As illustrated in Figure 1, LEDGER addresses this through three integrated components: graph construction, dependency-aware retrieval, and consistency verification. The detailed prompt is in Section A.

3.1 Semantic Graph Construction

Given document D with units $U = \{u_1, u_2, \ldots, u_n\}$, LEDGER constructs a graph $G = (V, E)$ where each node $v_i \in V$ corresponds to unit $u_i \in U$. We convert the document to HTML and parse it into a DOM tree, where each structural element (section, paragraph, figure, table, equation) becomes a unit with unique identifier id_i, content c_i, type τ_i, and position p_i.

Each unit u_i maps to graph node $v_i = (\mathrm{id}_i, s_i, e_i, \tau_i, p_i, t_i)$ where s_i is a summary of c_i, $e_i \in \mathbb{R}^d$ is the embedding vector, and t_i is the modification timestamp. The edge set $E \subseteq V \times V \times T$ encodes four dependency types: $E_{\mathsf{REF}} = \{(v_i, v_j, \mathsf{REFERENCES}) \mid r_{\text{exp}}(u_i, u_j)\}$ for explicit references, $E_{\mathsf{DEP}} = \{(v_i, v_j, \mathsf{DEPENDS}) \mid r_{\text{imp}}(u_i, u_j)\}$ for implicit dependencies, $E_{\mathsf{CON}} = \{(v_i, v_j, \mathsf{CONTAINS}) \mid u_j \in u_i\}$ for hierarchical containment, and $E_{\mathsf{REL}} = \{(v_i, v_j, \mathsf{RELATED}) \mid r_{\text{sem}}(u_i, u_j)\}$ for semantic relationships.

Explicit references. For edges where $r_{\text{exp}}(u_i, u_j)$ holds, content c_i contains a substring m that explicitly refers to identifier id_j. The computational challenge is ensuring three critical properties. First, *minimality*: the substring m must be minimal such that removing any token breaks the reference. Second, *unambiguity*: the reference must explicitly name id_j rather than relying on semantic similarity. Third, *validation*: the identified id_j must exist in $\{\mathrm{id}_1, \ldots, \mathrm{id}_n\}$. We implement these requirements through a structured extraction protocol (Appendix A) that queries a language model with formal constraints and parses

the structured output, returning validated pairs $\{(m_1, \text{id}_{j_1}), (m_2, \text{id}_{j_2}), \ldots\}$ where each m_k is a minimal span and $\text{id}_{j_k} \in \{\text{id}_1, \ldots, \text{id}_n\}$, enabling edge creation $(v_i, v_j, \texttt{REFERENCES})$.

Implicit dependencies. For edges where $r_{\text{imp}}(u_i, u_j)$ holds, unit u_j provides semantic prerequisites necessary to interpret u_i. The computational challenge is determining whether removing u_j would make u_i semantically incomplete, ambiguous, or ill-defined. This requires evaluating *necessity* rather than mere relatedness: u_j is a prerequisite only if u_i relies on definitions, assumptions, variables, or conceptual constructs introduced in u_j that cannot be resolved from generic background knowledge. We must also verify *directionality*: the dependency must be $u_j \to u_i$, not bidirectional or circular. We implement this through a counterfactual testing protocol (Appendix A) that presents both units to the language model with formal testing procedures, returning a directional dependency judgment enabling edge creation $(v_i, v_j, \texttt{DEPENDS})$ when $r_{\text{imp}}(u_i, u_j) = 1$.

Semantic relationships. For edges where $r_{\text{sem}}(u_i, u_j)$ holds, units discuss related topics. We compute cosine similarity $\text{sim}(e_i, e_j) = \frac{e_i \cdot e_j}{\|e_i\|\|e_j\|}$ and create edge $(v_i, v_j, \texttt{RELATED})$ when $\text{sim}(e_i, e_j) > \theta$, provided $(v_i, v_j) \notin E_{\texttt{REF}} \cup E_{\texttt{DEP}}$. The threshold θ balances precision and recall; we use $\theta = 0.7$ based on standard practice for high-confidence semantic similarity (Kandola et al., 2002). The embeddings e_i require summaries s_i that capture semantic content, which we extract through type-specific protocols (Appendix A). Hierarchical edges $E_{\texttt{CON}}$ come directly from the DOM tree structure.

3.2 Dependency-Aware Context Retrieval

For edit request $\mathcal{E}_t = (\text{instruction}, \text{target}, \text{scope})$, LEDGER retrieves context $\mathcal{C}_{\text{context}} \subseteq U$ through graph traversal in four steps.

Step 1: Target identification. We identify target nodes $\mathcal{N}_{\text{target}} \subseteq V$. If the instruction specifies an explicit target (e.g., "Figure 3"), we resolve it directly: $\mathcal{N}_{\text{target}} = \{v_j \mid \text{id}_j = \text{target}\}$. Otherwise, we compute the instruction embedding $e_{\mathcal{E}} = \text{Embed}(\text{instruction})$ and select the most similar nodes based on cosine similarity $\text{sim}(e_{\mathcal{E}}, e_i)$.

Step 2: Dependency expansion. We expand to include all connected nodes: $\mathcal{N}_{\text{deps}} = \mathcal{N}_{\text{target}} \cup \mathcal{N}_{\text{upstream}} \cup \mathcal{N}_{\text{downstream}}$ where upstream dependencies (what targets depend on) are $\mathcal{N}_{\text{upstream}} = \{v_j \mid \exists v_i \in \mathcal{N}_{\text{target}}, (v_i, v_j, t) \in E\}$ and downstream impacts (what depends on targets) are $\mathcal{N}_{\text{downstream}} = \{v_j \mid \exists v_i \in \mathcal{N}_{\text{target}}, (v_j, v_i, t) \in E\}$.

Step 3: Priority scoring. We prioritize nodes in $\mathcal{N}_{\text{deps}}$ based on their relationship to target nodes. Target nodes receive highest priority. Among connected nodes, we prioritize explicit references (`REFERENCES`) as critical for structural integrity, followed by implicit dependencies (`DEPENDS`) as necessary for semantic completeness, then containment relationships (`CONTAINS`) for hierarchical context, and finally semantic relationships (`RELATED`) as optional background. This ordering reflects each edge type's importance for maintaining document consistency.

Step 4: Budget-constrained packing. Given token budget B, we construct context by selecting nodes in descending score order. We define $\mathcal{C}_{\text{context}} = \{u_i \mid v_i \in \text{Pack}(\mathcal{N}_{\text{deps}}, B)\}$ where $\text{Pack}(\mathcal{N}, B)$ greedily selects nodes sorted by score until the cumulative token count $\sum_{v_i \in \text{selected}} |c_i|$ exceeds budget B. This achieves $|\mathcal{C}_{\text{context}}| = O(1)$ tokens regardless of $|D|$, typically retrieving 10 to 15% of total content while ensuring all critical dependencies are included.

3.3 Consistency Verification

After applying modifications Δ_t to produce D_{t+1}, LEDGER validates structural invariants on the set of modified nodes $\mathcal{N}_{\text{mod}} \subseteq V$ through three checks.

Reference integrity. For all outgoing reference edges from modified nodes, we verify targets exist in the updated graph: $\mathcal{V}_{\text{ref}} = \{(v_i, v_j) \mid (v_i, v_j, \texttt{REFERENCES}) \in E, v_i \in \mathcal{N}_{\text{mod}}, v_j \notin V_{t+1}\}$. Any pair in $\mathcal{V}_{\text{ref}}$ represents a broken reference requiring correction.

Terminology consistency. For modified nodes, we extract key terms $\mathcal{T}(c_i)$ from content and compare with dependent nodes: $\mathcal{V}_{\text{term}} = \{(v_i, v_j) \mid v_i \in \mathcal{N}_{\text{mod}}, (v_j, v_i, t) \in E, \mathcal{T}(c_i^{\text{new}}) \neq \mathcal{T}(c_i^{\text{old}})\}$. These indicate terminology changes requiring review of dependent content.

Semantic coherence. We recompute embeddings $e_i^{\text{new}} = \text{Embed}(c_i^{\text{new}})$ for modified nodes and identify cases where semantic relationships have weakened significantly: $\mathcal{V}_{\text{sem}} = \{(v_i, v_j) \mid v_i \in \mathcal{N}_{\text{mod}}, (v_i, v_j, \texttt{RELATED}) \in E, \text{sim}(e_i^{\text{new}}, e_j) < \text{sim}(e_i^{\text{old}}, e_j)\}$. These indicate potential semantic drift requiring review.

The graph updates incrementally by removing old edges incident to $\mathcal{N}_{\text{mod}}$ and recomputing new

Model	Full Doc.					LEDGER				
	Consistency	Ref. Valid.	Edit Quality	Pass	Tokens	Consistency↑	Ref. Valid.↑	Edit Quality ↑	Pass ↑	Tokens ↓
Claude Haiku 4.5	53.23	63.41	63.18	35.72	1,759	**80.12**	**80.36**	**83.05**	**84.27**	**1,479**
Claude Sonnet 4.5	55.48	58.62	67.19	36.54	1,725	**74.31**	**72.44**	**70.26**	**73.58**	**1,575**
Claude Opus 4.5	61.17	69.53	70.28	43.61	1,736	**75.42**	**77.18**	**78.34**	**77.66**	**1,508**
GPT-4.1	54.69	60.71	62.44	43.22	1,974	**74.53**	**81.29**	**79.16**	**74.84**	**1,478**
GPT-5 Chat	57.36	66.24	68.57	41.63	1,825	**77.08**	**74.19**	**76.42**	**82.11**	**1,489**
GPT-5.2-High	57.59	67.33	71.14	45.26	1,910	**77.44**	**74.61**	**75.38**	**75.92**	**1,683**
Average	56.59	64.31	67.13	41.00	1,822	**76.48**	**76.68**	**77.10**	**78.06**	**1,535**

Table 1: Comparison of LEDGER and baseline approaches across six language models. For each method, the first four columns report percentage (%) metrics (Consistency, Reference Validity, Edit Quality, and Overall Pass), while the last column reports the average number of tokens per edit. Results are the average value over testing cases.

edges using the construction algorithm, maintaining G synchronized with document state D_{t+1}.

4 Experiments

Evaluation Benchmark. We curated 1,900 test cases across six editing scenarios: *Local refinement* (120 cases) targets specific paragraphs without structural changes (e.g., "make this paragraph more concise"). *Cross-reference updates* (200 cases) modify content referenced elsewhere, requiring the system to identify all citing sections (e.g., editing Figure 3's caption affects all paragraphs referencing it). *Structural reorganization* (160 cases) moves or renumbers sections while maintaining cross-reference validity (e.g., "move Section 4.2 before Section 4.1"). *Content addition* (120 cases) inserts new material while preserving coherence. *Iterative editing* (160 cases) applies 3 to 5 dependent edits testing multi-step consistency. *Scalability tests* (120 cases) apply identical operations to documents from 5K to 100K tokens, testing whether token usage remains constant as document size increases.

Models. We evaluate six models: Claude 4.5 Haiku/Sonnet/Opus and GPT-4.1/5/5.2, with GPT-5.2 tested across four reasoning effort levels. Our primary baseline is the full-document approach, where the entire document is provided as input context to the LLM for each edit operation, but only the targeted section is modified. Note that token usage in this case reflects the input context provided to generate modifications, not full document regeneration output. For scalability analysis (Figure 7), we additionally compare against theoretical full-document regeneration, which would require regenerating all 5K-100K tokens as output and thus scales linearly with document size. We measure consistency (structural validity and cross-reference integrity), reference validity (percentage of correct cross-references), edit quality (correctness of modifications), overall pass rate (percentage meeting all criteria), and token usage per edit.

Baselines. We compare LEDGER against full-document regeneration (processes entire document for each edit, our primary baseline), sliding window (fixed 4K token context), RAG-based retrieval (semantic search without structural awareness), and Claude+Memory (built-in memory without explicit graph structure).

Metrics. We measure *consistency* (reference validity, structural coherence, and overall preservation of structural constraints), *efficiency* (token usage per edit and percentage reduction versus baseline), and *quality* (correctness of edits and overall pass rate combining all criteria). Metrics are computed automatically through programmatic validation with LLM as judge, with quality assessed against gold-standard human edits for 200 test cases. We also include the examples for Metrics evaluation in Appendix Section C.

4.1 Main Results

Table 1 presents results comparing LEDGER against full-document baseline across all six models. LEDGER achieves 76% average consistency versus 56% baseline, a 20 percentage point improvement that remains consistent across model families. This translates to 78% overall pass rate versus 41% baseline. Token usage averages 1,535 tokens per edit for LEDGER versus 1,822 for baseline. The benefits prove particularly pronounced for smaller models: Claude Haiku 4.5 with LEDGER achieves 80% consistency versus 53% baseline (27 point gain), while larger models show 14 to 20 point gains. This suggests explicit structural memory provides larger relative benefits when model capacity is constrained.

Performance over Editing Types. Figure 2 breaks down LEDGER performance across editing scenarios. Cross-reference updates (86% consistency),

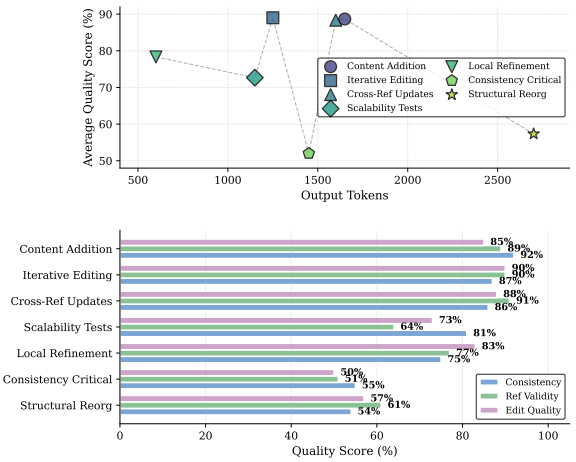

Figure 2: Performance breakdown by edit operations. Top chart shows token usage and quality metrics across seven edit categories. Bottom bar chart shows the performance of LEDGER on different editing categories.

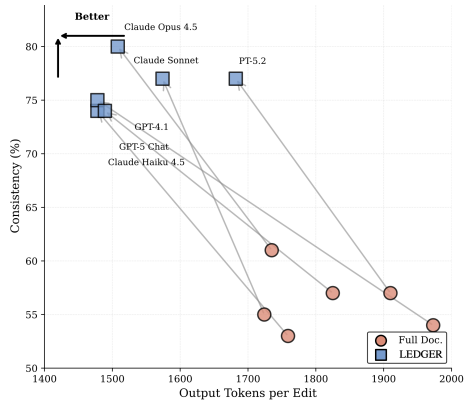

Figure 3: Efficiency-quality tradeoff analysis. Scatter plot with Pareto frontier showing the relationship between token efficiency (x-axis, lower is better) and consistency preservation (y-axis, higher is better) across all experimental conditions. Each point represents one (model, edit type) combination.

content addition (88%), and iterative editing (86%) achieve the strongest results because these scenarios heavily exercise dependency tracking. Local refinement achieves 85% consistency with minimal token usage (560 tokens), reflecting efficient targeted retrieval. Structural reorganization achieves only 54% consistency despite higher token usage (2,749 tokens), representing the most challenging category where cascading updates across structural levels strain graph construction capabilities.

Consistency Evaluation. Figure 3 visualizes the efficiency-quality frontier across all models and configurations. LEDGER configurations (filled markers) cluster in the upper-left quadrant with consistency exceeding 70% and token usage between 1,200 to 2,000 tokens. Baseline points (hollow markers) scatter across lower consistency regions (40-65%) with widely varying token usage. Pareto frontier analysis reveals most LEDGER configurations lie on or near the optimal frontier, meaning no alternative configuration achieves substantially better quality without increasing tokens, or better efficiency without decreasing quality.

Token Efficiency. Figure 4 compares average token usage between baseline and LEDGER across six models for documents ranging from 5K to 100K tokens. LEDGER consistently achieves 85 to 92% token reduction across all models when compared to full-document regeneration at scale, with mean reduction of 88.5%. The consistency demonstrates model-independent efficiency gains.

Editing Quality Analysis. Figure 5 visualizes quality preservation across models and evaluation dimensions. LEDGER maintains or improves

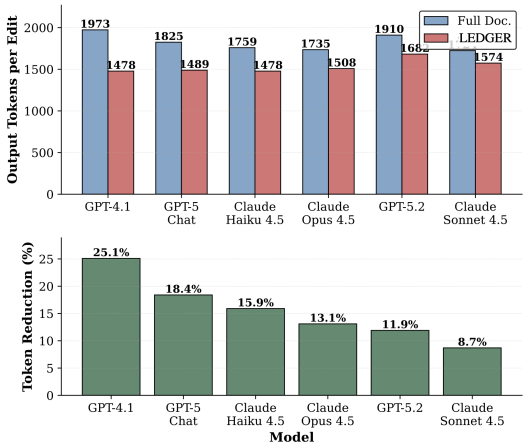

Figure 4: Token efficiency comparison across models in scalability tests. Bar chart comparing average token usage between baseline and LEDGER across six models for documents ranging from 5K to 100K tokens.

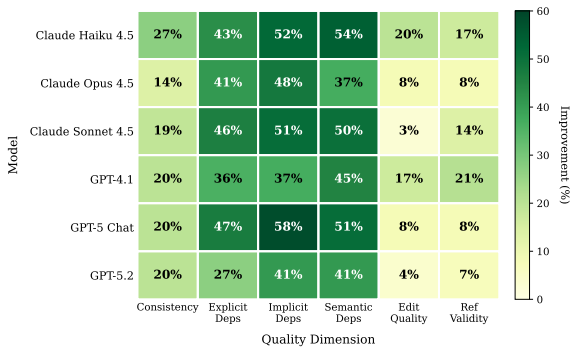

Figure 5: Quality metrics preservation across models and evaluation dimensions. Heatmap showing consistency preservation, reference validity, and edit quality percentages for both baseline and LEDGER.

quality across all dimensions: consistency preservation (74-80%), reference validity (72-81%), and

Reasoning Level	Method	Cons. (%)	Ref. Val. (%)	Pass (%)	Tok./Edit ↓
None	Full Doc.	39.28	46.61	28.43	1,521
	LEDGER	**67.14**	**69.37**	**65.82**	**1,497**
Low	Full Doc.	48.52	55.18	34.64	**1,494**
	LEDGER	**80.12**	**82.41**	**79.33**	1,508
Medium	Full Doc.	42.67	50.39	30.58	1,837
	LEDGER	**77.26**	**79.14**	**75.49**	**1,522**
High	Full Doc.	57.59	67.33	45.26	1,910
	LEDGER	**78.08**	**79.46**	**77.19**	**1,653**

Table 2: GPT-5.2 performance across reasoning efforts.

edit quality (70-83%). The heatmap shows consistent performance across model families with no systematic degradation, suggesting benefits stem from framework architecture rather than model-specific capabilities.

4.2 Reasoning Effort Analysis

Table 2 presents GPT-5.2 performance across reasoning effort levels. LEDGER with low reasoning effort achieves 80% consistency substantially outperforming baseline with high reasoning effort (57% consistency). This demonstrates that explicit structural representations can substitute for approximately one level of internal reasoning while using 70% fewer tokens. LEDGER provides consistent 20-32 percentage point improvements across all reasoning levels. The baseline exhibits non-monotonic performance where medium reasoning (43% consistency) underperforms both low (49%) and high (57%), suggesting increased computation without structural guidance can introduce complexity rather than resolve it.

Figure 6 visualizes this reasoning substitution effect. The left panel shows LEDGER maintains stable consistency (71-80%) across all reasoning levels, while baseline exhibits a U-shaped pattern with medium reasoning (51%) underperforming both low (58%) and high (57%). The right panel shows both systems experience increasing token usage with higher reasoning effort, but LEDGER consistently requires 11-15% fewer tokens at equivalent levels. The stability of LEDGER across reasoning modes indicates that explicit dependency information provides structural guidance that compensates for limited reasoning capacity, reducing the burden of inferring relationships through extended internal deliberation.

4.3 Scalability Analysis

Figure 7 demonstrates that LEDGER maintains constant token usage averaging 1.5k tokens per edit across all document lengths from 5K to 100K tokens, while full-document regeneration scales

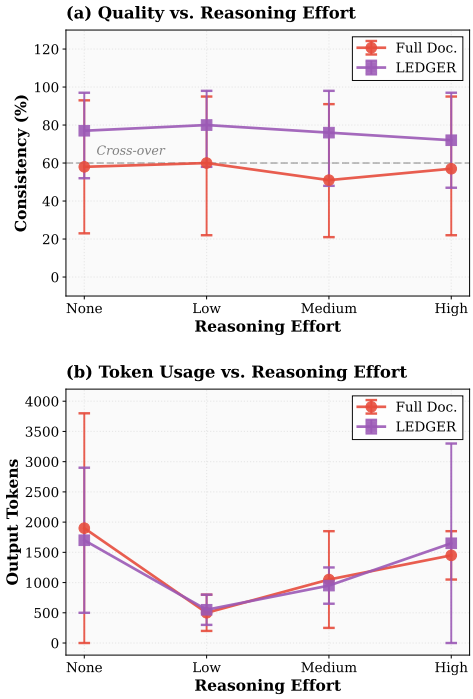

Figure 6: Impact of reasoning effort on performance (GPT-5.2). Top panel: Consistency preservation versus reasoning effort level. Bottom panel: Token usage versus reasoning efforts.

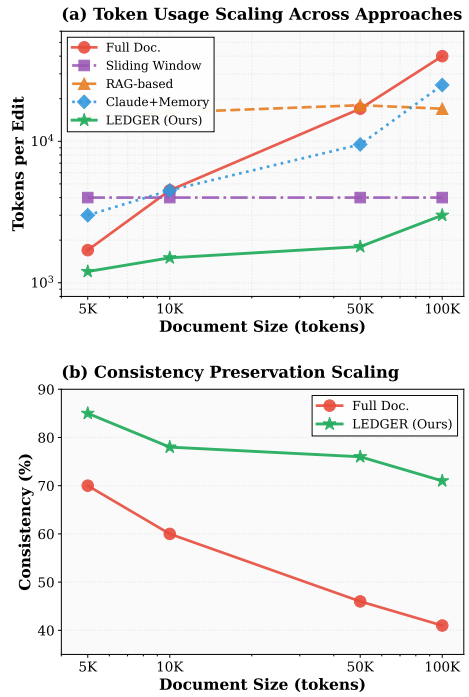

Figure 7: Token usage scaling across document editing approaches.

linearly. At 100K tokens, LEDGER maintains approximately 1.5k tokens per edit. This constant time behavior emerges from dependency-aware graph retrieval's architectural properties. For each edit, graph traversal identifies a locally relevant subset of nodes and retrieves only their content.

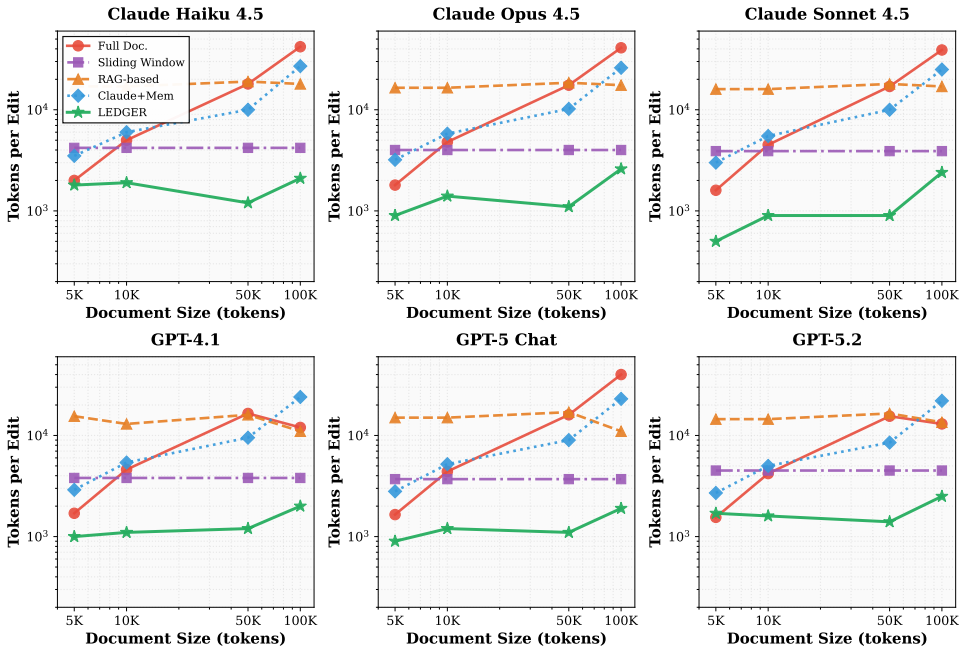

Figure 8: Per-model scalability analysis across six LLM models. All models demonstrate consistent patterns: LEDGER maintains constant efficiency across document sizes.

Configuration	Consistency (%) ↑	Ref Valid (%) ↑	Overall Pass (%) ↑
LEDGER (Full System)	**76.18**	**76.01**	**78.06**
w/o Dependency Tracking	62.47	65.12	58.69
w/o Graph-based Retrieval	68.33	71.58	65.41
w/o Consistency Verification	71.06	72.44	70.27
GPT-5.2-High (Full Doc)	57.59	67.33	45.26

Table 3: Ablation study removing individual components. All three components contribute to overall performance, with dependency tracking providing the largest single contribution.

Crucially, whether the full document contains 50 nodes or 5,000 nodes, the retrieved context size remains comparable because dependencies are inherently local rather than global. A paragraph edit requires awareness of its containing section, referenced definitions, and downstream citations, but not the entire document structure. Additional analysis is provided in Figure 8, where per-model scalability demonstrates architecture-independent constant token usage across all six models.

4.4 Ablation Study

Table 3 isolates the contribution of individual components. Removing dependency tracking (relying only on semantic similarity) reduces consistency to 62%, a 14 point drop, validating that explicit tracking of structural edges is crucial. Removing graph-based retrieval (using full-document context instead) achieves 68% consistency, indicating that targeted context selection improves consistency beyond simply providing more context. Removing consistency verification achieves 71% consistency, showing that post-edit validation prevents error propagation. All three components contribute independently to overall performance.

5 Conclusion

We introduce LEDGER, a framework that address the context engineering bottleneck in agentic document editing through explicit dependency graphs. By extracting and representing structural and semantic dependencies as typed edges, LEDGER achieves 76% consistency versus 56% baseline while reducing token usage. We contribute a curated benchmark of 1,900 test cases across diverse document types and edit scenarios, enabling rigorous evaluation of consistency maintenance. Critically, LEDGER with minimal internal reasoning substantially outperforms baselines with expensive extended reasoning while using fewer tokens, demonstrating that explicit structural representations can substitute for costly computation. This design principle generalizes to code refactoring, knowledge bases, and multi-step workflows. Future directions include richer dependency detection incorporating coreference resolution and discourse structure, extending to collaborative and cross-document editing with conflict resolution mechanisms, and multimodal extensions to presentations and rich media documents.

Limitations

While LEDGER demonstrates consistent improvements across diverse test cases, several limitations warrant discussion. First, the framework is optimized for structured documents with clear hierarchical organization and explicit identifiers (sections, figures, equations), and may perform suboptimally on unstructured content such as creative writing, conversational text, or documents lacking consistent formatting conventions. Second, our evaluation focuses primarily on English-language documents; the dependency extraction protocols may require adaptation for languages with different syntactic structures, particularly for implicit dependency detection that relies on coreference resolution. Third, although we demonstrate generalization across six models, the quality of dependency extraction and editing operations remains bounded by the underlying LLM's capabilities—weaker models may produce less accurate dependency graphs, partially offsetting efficiency gains. Fourth, the framework assumes a relatively stable document structure during editing sessions; operations that fundamentally reorganize document architecture (e.g., merging chapters, restructuring entire sections) may require full graph reconstruction rather than incremental updates. Fifth, our experiments focus on single-agent editing scenarios, and the framework's behavior under concurrent multi-agent editing with potential conflicts requires further investigation. Sixth, while we evaluate consistency through reference validity and structural coherence metrics, these may not fully capture subtle semantic inconsistencies that human readers would detect. Finally, the constant token usage achieved by LEDGER represents an average across test cases; pathological documents with extremely dense interconnections might require retrieving larger context subgraphs, though this remains substantially more efficient than full-document processing.

Potential Risk

The proposed method's effectiveness depends critically on the accuracy of LLM-based dependency extraction protocols. Errors in identifying explicit references, implicit dependencies, or semantic relationships during graph construction propagate through subsequent editing operations, potentially causing the system to miss critical dependencies or retrieve irrelevant context. The extraction protocols require careful prompt engineering and may exhibit inconsistent performance across document domains, with technical specifications having clearer structural patterns than creative or conversational content. Additionally, the graph construction and incremental update processes introduce computational overhead: while amortized across multiple edits, the initial graph construction for very large documents (>100,000 tokens) or documents with dense interconnections can be time-consuming. The semantic similarity threshold for RELATED edges represents a design choice that balances precision and recall but may require domain-specific tuning. Furthermore, the method assumes document structure can be reliably parsed into DOM trees, which may fail for poorly formatted documents, legacy formats, or content with inconsistent markup. Finally, the framework's reliance on external graph memory introduces a synchronization challenge: if the graph becomes desynchronized from the document state due to external edits or system failures, consistency guarantees no longer hold until the graph is reconstructed.

Ethical Considerations

The deployment of AI agents for automated document editing raises several ethical concerns that warrant careful consideration. First, the system's ability to maintain cross-document consistency and propagate changes efficiently could facilitate unauthorized modifications or manipulation of collaborative documents if access controls are inadequate, particularly in settings involving multiple stakeholders or version-controlled content. Second, while the framework assists human editors, over-reliance on automated editing agents may reduce critical human oversight, potentially allowing errors or inappropriate modifications to propagate undetected through dependency chains. The LLM-based dependency extraction and editing components may also inherit biases present in their training data, potentially affecting how relationships are identified or how edits are formulated across different document types, domains, or writing styles. Third, the system processes document content to construct semantic graphs and generate embeddings, raising privacy concerns when applied to sensitive or confidential documents—organizations must ensure appropriate data handling practices and consider on-premises deployment for sensitive applications. Fourth, the framework's effec-

tiveness in maintaining consistency might create false confidence in fully automated editing workflows, whereas human judgment remains essential for evaluating semantic appropriateness, stylistic coherence, and domain-specific correctness that automated systems cannot fully capture. We emphasize that LEDGER is designed as an assistive tool augmenting human editors rather than replacing them, and recommend that deployed systems include human-in-the-loop verification for critical editing operations, maintain comprehensive audit trails, and provide transparency about which modifications were AI-generated versus human-authored.

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Make your llm fully utilize the context. *Advances in Neural Information Processing Systems*, 37:62160–62188.

Jiawen Chen, Yajun Yang, Chenyang Wang, Heng Zhang, Chao Qiu, and Xiaofei Wang. 2021. Multitask offloading strategy optimization based on directed acyclic graphs for edge computing. *IEEE Internet of Things Journal*, 9(12):9367–9378.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *International Conference on Learning Representations*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, volume 30.

Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Monishwaran Maheswaran, Sebastian Zhao, June Paik, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2025. Squeezed attention: Accelerating long context length llm inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32631–32652.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*.

Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. 2024. LLM maybe LongLM: Self-extend LLM context window without tuning. In *International Conference on Machine Learning*.

Jaz Kandola, Nello Cristianini, and John Shawe-taylor. 2002. Learning semantic similarity. *Advances in neural information processing systems*, 15.

Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

Philippe Laban, Alexander Richard Fabbri, Caiming Xiong, and Chien-Sheng Wu. 2024. Summary of a haystack: A challenge to long-context llms and rag systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9885–9903.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. *arXiv preprint arXiv:2502.20766*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, and 1 others. 2024a. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516*.

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

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*.

André Martins, António Farinhas, Marcos Treviso, Vlad Niculae, Pedro Aguiar, and Mario Figueiredo. 2020. Sparse and continuous attention mechanisms. *Advances in Neural Information Processing Systems*, 33:20989–21001.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Lin Song, Yukang Chen, Shuai Yang, Xiaohan Ding, Yixiao Ge, Ying-Cong Chen, and Ying Shan. 2024. Low-rank approximation for sparse attention in multimodal llms. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13763–13773.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023. Augmenting language models with long-term memory. *Advances in Neural Information Processing Systems*, 36:74530–74543.

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

Chaojun Xiao, Pengle Zhang, Xu Hu, Guangxuan Liu, Yichuan Xu, Chenyang Lin, Chengqiang Wang, Yishi Dong, Furu Wei, and Yeyun Cao. 2024. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. In *Advances in Neural Information Processing Systems*, volume 37.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks? In *International Conference on Learning Representations*.

Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. 2021. QA-GNN: Reasoning with language models and knowledge graphs for question answering. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 535–546.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, and 1 others. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*.

Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. 2022. GreaseLM: Graph REASoning enhanced language models for question answering. In *International Conference on Learning Representations*.

Appendix

Section A Dependency Extraction Protocols

- Explicit Reference Relation (r_{exp})
- Implicit Semantic Dependency Relation (r_{imp})
- Semantic Relatedness Relation (r_{sem})
- Implementation Notes

Section B LEDGER Aided Document Editing Examples

- Example Task: Revising a Theorem Statement with Cascading Updates

Section C Evaluation Metrics and Examples

- Metric Definitions
- Evaluation Examples (3 cases)
- Metric Computation

Section D Benchmark Construction

- Benchmark Overview
- Test Scenario Taxonomy (7 categories)
- Document Generation Process
- Benchmark Statistics
- Evaluation Protocol

Section E Pseudocode for LEDGER Algorithms

- Semantic Graph Construction
- Dependency-Aware Context Retrieval
- Consistency Verification

A Dependency Extraction Protocols

This section provides the complete extraction protocols for each dependency type. Each protocol is designed to operationalize the formal relation definitions from Section 3 through structured constraints that ensure mathematical rigor.

Remark on the token cost for Graph Construction. Graph construction incurs a one-time cost of $O(n)$ for document parsing and embedding computation, plus $O(n \cdot k)$ for dependency extraction, where k is the average number of candidate dependencies per unit (typically 5-10 nearby units for implicit dependencies, avoiding all-pairs $O(n^2)$ computation). For a typical 10K token document with 100 units, this requires approximately 500 pairwise LLM calls for implicit dependency extraction, completing in 2-3 minutes. This cost is amortized across multiple editing operations on the same document, with incremental updates (Algorithm 2) adding minimal overhead per edit.

A.1 Explicit Reference Relation (r_{exp})

Operator definition reminder: $r_{\text{exp}}(u_i, u_j) = 1$ if and only if there exists a minimal span $m \subset u_i$ that explicitly refers to identifier id_j.

Extraction Protocol for r_{exp}:

You are given a text unit u_i from a technical document with known identifiers $\{\text{id}_1, \ldots, \text{id}_n\}$.

Task: Identify all substrings $m \subset u_i$ that establish explicit references to other units.

Constraints:

1. **Explicit naming:** The substring m must explicitly and unambiguously refer to another document unit by identifier (e.g., section number "Section 2.1", equation label "Equation 3", figure caption "Figure 5", named definition "Theorem 1").

2. **Minimality:** The span m must be minimal – removing any token from m must break the reference. For example, in "as shown in Figure 3 above", the minimal span is "Figure 3", not "Figure 3 above".

3. **Surface-level only:** Do not infer references that rely on semantic similarity, paraphrasing, or topic relatedness. Only surface-level, identifier-based references are allowed. For example, "the matrix operation discussed earlier" is NOT an explicit reference unless it includes an identifier.

4. **LaTeX references:** Include LaTeX cross-references (e.g., `\ref{sec:method}`, `\eqref{eq:loss}`, `\cite{smith2020}`) as explicit references when the label can be resolved to id_j.

Output format: Return a set of pairs $\{(m_1, \text{id}_{j_1}), (m_2, \text{id}_{j_2}), \ldots\}$ where each m_k is the minimal substring and id_{j_k} is the referenced identifier. If no such spans exist, return an empty set.

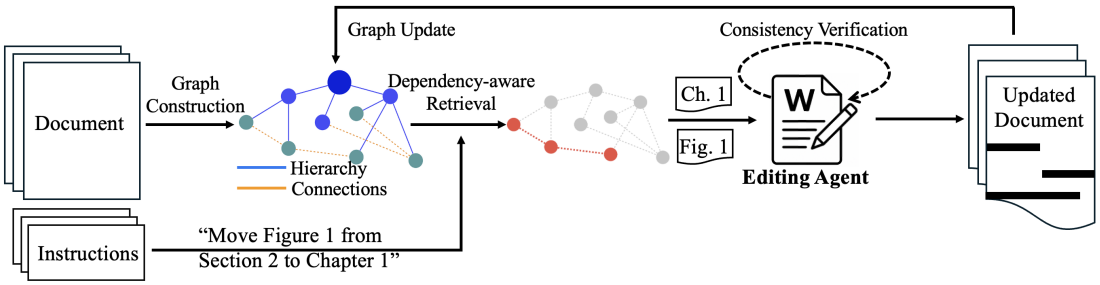

Figure 9: LEDGER workflow for document editing with dependency awareness. The system constructs a semantic graph from the input document, capturing hierarchical relationships (solid blue lines) and implict, explict and semantic connections (dotted orange lines). When an edit instruction is received, the graph guides context retrieval to identify relevant dependencies (highlighted in red). The editing agent applies edits with consistency verification, and the semantic graph is updated incrementally. Example operations include updating cross-references (e.g., figure references), reorganizing sections (e.g., update Chapter 1 $\rightarrow$ 2), and adding examples while maintaining consistency across all dependent elements.

Example:

- Input: "The method described in Section 2.1 extends Theorem 4 from (**?**)."
- Output: ("Section 2.1", "sec-2-1"), ("Theorem 4", "thm-4"), ("(**?**)", "cite-jones2019")

A.2 Implicit Semantic Dependency Relation (r_{imp})

Operator definition reminder: $r_{\text{imp}}(u_i, u_j) = 1$ if and only if u_j provides semantic prerequisites necessary to interpret u_i.

Extraction Protocol for r_{imp}:

You are given two text units (u_i, u_j) from the same document, where u_j appears before u_i in document order.

Task: Determine whether u_j is a *necessary semantic prerequisite* for understanding u_i.

Test procedure:

1. **Counterfactual removal:** Consider u_i in isolation, assuming that u_j (and only u_j) is removed from the document. All other units remain available.
2. **Semantic completeness check:** Decide whether u_i becomes semantically incomplete, ambiguous, or ill-defined due to missing content from u_j. Specifically, check if u_i relies on:
   - Definitions or terminology introduced in u_j
   - Mathematical notation or variables defined in u_j
   - Assumptions, constraints, or problem setup from u_j
   - Conceptual constructs or frameworks established in u_j
   - Results, lemmas, or intermediate conclusions from u_j
3. **Specificity requirement:** Identify the specific elements in u_i that rely on content introduced in u_j. The dependency must be concrete, not merely topical similarity.
4. **Generic knowledge exclusion:** The dependency must not be resolvable by generic background knowledge in the domain. For example, if u_i uses the term "gradient descent" and u_j defines it, but gradient descent is common knowledge in machine learning, this does NOT establish $r_{\text{imp}}(u_i, u_j)$.
5. **Directionality:** Verify that the dependency is strictly directional: $u_j \rightarrow u_i$. The relationship cannot be bidirectional or circular.

Output format: Return `True` if all conditions are satisfied, establishing $r_{\text{imp}}(u_i, u_j) = 1$. Otherwise, return `False`. If `True`, also return the specific elements in u_i that depend on u_j.

Example:

- u_j: "We define the loss function $\mathcal{L}(\theta) = \sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$."
- u_i: "To minimize $\mathcal{L}(\theta)$, we compute the gradient with respect to θ."
- Output: `True`, with dependency elements: {"$\mathcal{L}(\theta)$", "loss function definition"}
- Justification: Without u_j, the symbol $\mathcal{L}(\theta)$ in u_i is undefined.

A.3 Semantic Relatedness Relation (r_{sem})

Operator definition reminder: $r_{\text{sem}}(u_i, u_j) = 1$ if and only if units u_i and u_j discuss semantically related topics, quantified as $\text{sim}(e_i, e_j) > \theta$ where e_i, e_j are embedding vectors and θ is a threshold.

Extraction Protocol for r_{sem}:

You are given a text unit u_i of type $\tau_i \in \{\text{section}, \text{paragraph}, \text{figure}, \text{table}, \text{equation}\}$.

Task: Generate a concise summary s_i that captures the semantic content of u_i for embedding-based similarity computation.

Type-specific protocols:

For sections ($\tau_i = \text{section}$):

1. Extract the main claim or thesis of the section (1 sentence)
2. Identify 2-3 key contributions, findings, or concepts introduced
3. Exclude: Structural markers ("In this section..."), transitional phrases, citations
4. Target length: 50-100 tokens

For paragraphs (τ_i = paragraph):

1. Identify the central idea or main point (1-2 sentences)
2. Include key technical terms or concepts
3. Exclude: Supporting details, examples, citations
4. Target length: 30-50 tokens

For figures (τ_i = figure):

1. Use the figure caption directly as s_i
2. If caption is very long (>100 tokens), extract the first sentence and key technical terms
3. Include figure type (e.g., "graph showing", "architecture diagram of")

For tables (τ_i = table):

1. Use the table caption as s_i
2. If caption is uninformative, describe what is being compared or measured (e.g., "Performance comparison of methods on datasets")
3. Include column/row headers if they convey key concepts

For equations (τ_i = equation):

1. Extract any accompanying description or inline explanation
2. If no description exists, describe the equation type (e.g., "optimization objective", "probability distribution")
3. Target length: 20-40 tokens

Output format: Return summary s_i as a coherent text string. The embedding $e_i = \text{Embed}(s_i)$ is computed from this summary.

Semantic edge creation: After computing embeddings for all units, create edge $(v_i, v_j, \texttt{RELATED})$ when:

- $\text{sim}(e_i, e_j) = \frac{e_i \cdot e_j}{\|e_i\|\|e_j\|} > \theta$ where $\theta = 0.7$
- $(v_i, v_j) \notin E_{\texttt{REF}} \cup E_{\texttt{DEP}}$ (not already connected by explicit or implicit dependency)

Example:

- Input (section): "This section introduces our novel attention mechanism. We propose a multi-head cross-attention layer that attends to both encoder and decoder states. Experiments show significant improvements over baseline transformers."
- Output: "Novel attention mechanism using multi-head cross-attention over encoder and decoder states, improving transformer performance." (18 tokens)

A.4 Implementation Notes

Validation and quality control:

- For r_{exp}: Validate that all extracted id_j exist in the known identifier set $\{\text{id}_1, \ldots, \text{id}_n\}$. Discard any references to non-existent identifiers.
- For r_{imp}: Apply the protocol to unit pairs (u_i, u_j) where $j < i$ (i.e., u_j precedes u_i). The test is computationally expensive, so prioritize pairs where u_i is close to u_j in document order (e.g., within 5-10 units).
- For r_{sem}: Compute embeddings using a consistent model (e.g., `text-embedding-ada-002`, `sentence-transformers/all-mpnet-base-v2`). The threshold $\theta = 0.7$ balances precision (avoiding spurious edges) and recall (capturing true semantic relationships).

Edge priority: When multiple edge types could connect the same unit pair, prioritize: $E_{\text{REF}} > E_{\text{DEP}} > E_{\text{REL}}$. This ensures explicit dependencies take precedence over semantic relationships.

B LEDGER aided Document Editing Examples

B.1 Example Task: Revising a Theorem Statement with Cascading Updates

Task Description: Revise Theorem 2 in Section 3.2 by tightening the revenue bound from $O(\log n)$ to $O(1)$, ensuring all dependent sections remain consistent with the updated statement.

Document Context: A 10,000-token academic paper on dynamic pricing for hotel room allocation, containing 7 sections with multiple cross-references between theoretical results, algorithm descriptions, case study validation, and discussion.

B.1.1 Dependency Graph Construction

GUIDE constructs a semantic graph identifying three types of dependencies:

Explicit References:

- Theorem 2 (Section 3.2) ← cited by [Section 4.para1, Section 5.para2, Section 6.para1]
- Lemma 1 (Section 3.1) ← cited by [Section 3.2.para2, Section 4.para2]
- Equation (5) (Section 2.2) ← cited by [Section 3.1.proof, Section 4.para3, Section 5.para3]

Implicit References:

- "the optimality guarantee" (Section 4.para1) → Theorem 2
- "our main theorem" (Section 5.para2) → Theorem 2
- "this bound" (Section 3.2.proof.para2) → Lemma 1

Semantic Dependencies:

- Section 4.para1 `builds_on` Theorem 2 (algorithmic extension)
- Section 5.para2 `validates` Theorem 2 (empirical confirmation)
- Section 6.para1 `extends` Theorem 2 (multi-hotel generalization)

B.1.2 Edit Execution with Dependency Tracking

GUIDE identifies that modifying Theorem 2 triggers the critical dependency rule: *If Theorem 2 changes → MUST verify [4.para1, 5.para2, 6.para1].* The agent executes targeted edits in dependency order:

Step 1: Update Theorem 2 statement (Section 3.2)

Original: "...within $O(\log n)$ of the optimal offline benchmark"
Revised: "...within $O(1)$ of the optimal offline benchmark"

Step 2: Update proof outline to align with new bound

Original: "Combining these bounds across n periods and applying harmonic series properties yields the $O(\log n)$ gap"
Revised: "...we charge each deviation to a constant decrease in a suitable potential function, implying only $O(1)$ total loss"

Steps 3–7: Propagate to dependent sections

- Section 3.2 proof paragraph: Update to "horizon-independent constant bound"
- Section 4.para1: Change "optimality guarantee" to "constant-gap optimality guarantee"
- Section 4.para3: Update reference to "constant-gap guarantee specified in Theorem 2"
- Section 5.para2: Replace "$O(\log n)$ bound" with "$O(1)$ bound"
- Section 6.para1: Insert "preserving the constant additive gap"

B.1.3 Efficiency Analysis

Token Utilization:

- Total document: 10,000 tokens
- Retrieved context: $\sim$1,200 tokens (12% of document)
  - Theorem 2 statement and proof
  - Explicit dependencies: [4.para1, 4.para3, 5.para2, 6.para1]
  - Implicit reference text
- Processing overhead: 7 targeted paragraph replacements

Comparison to Baseline Approaches:

- **Full document rewrite:** Would require processing all 10,000 tokens, with risk of introducing inconsistencies in unrelated sections
- **Local edit without dependency tracking:** Would update Theorem 2 but miss the 6 dependent references, breaking document consistency
- **GUIDE approach:** Processes 12% of document tokens while maintaining global consistency through explicit dependency tracking

B.1.4 Consistency Verification

GUIDE ensures consistency across multiple dimensions:

1. **Referential Integrity:** All explicit citations to Theorem 2 now correctly describe the $O(1)$ bound
2. **Semantic Coherence:** Algorithm design and case study sections align with the tightened theoretical guarantee
3. **Proof Validity:** Proof outline mechanism (potential-based charging) supports the stronger constant bound
4. **Terminology Consistency:** Uniform use of "constant additive gap" and "$O(1)$ bound" throughout dependent sections

The edit preserves document flow while implementing a substantial theoretical improvement that cascades through algorithm description, empirical validation, and discussion sections—demonstrating GUIDE's capability to maintain consistency in complex, interconnected technical documents.

C Evaluation Metrics and Examples

This appendix provides formal definitions of the evaluation metrics used to assess LEDGER performance, along with concrete examples from our benchmark dataset illustrating how each metric is evaluated.

C.1 Metric Definitions

We evaluate system performance across seven complementary metrics organized into three categories: dependency awareness, efficiency, and quality.

Dependency Awareness Metrics:

- **Dependency Detection (Explicit):** Measures whether the system correctly identifies all explicit references in the document, including section citations ("Section 3.2"), figure references ("Figure 1"), table references ("Table 2"), equation citations ("Equation (1)"), and algorithm references ("Algorithm 1"). This metric is computed as the percentage of explicit references correctly identified and included in the retrieved context.
- **Dependency Detection (Implicit):** Measures whether the system correctly resolves implicit references such as anaphoric expressions ("this approach", "the metric", "these results") and contextual references ("the preceding computation", "as described earlier"). This metric is computed as the percentage of implicit references correctly resolved to their referents.
- **Dependency Detection (Semantic):** Measures whether the system recognizes semantic relationships between document elements that lack explicit citations, such as conceptual dependencies (one section building on another), comparisons (contrasting methods), and synthesis (combining results). This metric is computed as the percentage of semantic dependencies correctly identified through graph edges.

Efficiency Metrics:

- **Token Efficiency:** Measures the ratio of tokens retrieved for an edit operation to the total document size. An efficient system retrieves only the minimal context necessary for consistency. This metric is computed as: Token Efficiency $= \frac{\text{Tokens Retrieved}}{\text{Total Document Tokens}} \times 100\%$. Lower percentages indicate higher efficiency, with typical values ranging from 10 to 15% for LEDGER versus 100% for full document approaches.

Quality Metrics:

- **Reference Validity:** Measures whether all explicit references (section numbers, figure citations, table references) remain correct after an edit operation. This metric is computed as the percentage of explicit references that correctly resolve to their intended targets after modifications. A broken reference (e.g., citing "Figure 3" when only Figures 1 and 2 exist) constitutes a failure.
- **Consistency Preservation:** Measures whether edits maintain semantic coherence with dependent content. This includes verifying that implicit references remain meaningful, terminology changes propagate to dependent sections, and semantic relationships remain valid. This metric is computed as the percentage of dependent elements that remain consistent with the edited content.
- **Edit Quality:** Measures whether the edit correctly addresses the instruction while maintaining appropriate style, clarity, and factual accuracy. This metric is computed through comparison with gold standard human edits, assessing whether the modification achieves the stated goal without introducing errors or unintended changes.

C.2 Evaluation Examples

We present three representative examples from our benchmark dataset to illustrate how these metrics are applied in practice.

C.2.1 Example 1: Local Paragraph Refinement

Scenario: Academic paper (12K tokens) with 6 sections. Edit instruction: "Improve the clarity of the second paragraph in Section 4.1 (Limitations) by presenting a more balanced discussion of memory overhead and suggesting potential mitigation strategies."

Document Structure:

- Explicit references: Figure 2 cited in 4 locations; Definition "token efficiency" cited in 2 locations; Algorithm 1 cited in 3 locations
- Implicit references: "the metric" in Section 4.1 refers to Definition "token efficiency"
- Semantic dependencies: Section 4.1 builds on Algorithm 1; Section 5 synthesizes findings from Section 4

Metric Evaluation:

Dependency Detection (Explicit): System must identify that Figure 2 appears in paragraphs 1.3, 3.2.2, 4.1, and 5.2; Algorithm 1 appears in sections 2.2 and 4.1; Definition "token efficiency" appears in sections 3.1 and 4.1. **Pass criterion:** All 5 explicit reference types correctly identified.

Dependency Detection (Implicit): System must resolve that "the metric" in Section 4.1 paragraph 2 refers to Definition "token efficiency" in Section 3.1, requiring coreference resolution across sections. **Pass criterion:** Implicit reference correctly resolved and included in context.

Dependency Detection (Semantic): System must recognize that Section 4 paragraph 1 semantically compares baseline methods with the Local Refinement Module from Section 3.2, despite no explicit citation. **Pass criterion:** Semantic comparison relationship detected.

Token Efficiency: System should retrieve: Section 4.1 second paragraph (target, 150 tokens) + Definition "token efficiency" (80 tokens) + Algorithm 1 (200 tokens) + Figure 2 caption (70 tokens) + Table 1 caption (50 tokens) = approximately 550 tokens. **Pass criterion:** Retrieved tokens $\leq$ 1,500 tokens (12.5% of 12K document).

Reference Validity: After refinement, all explicit references in Section 4.1 paragraph 2 must correctly resolve: Algorithm 1 still exists, Definition "token efficiency" still exists, Section 3.1 still exists. **Pass criterion:** All explicit references remain valid.

Consistency Preservation: Updated paragraph must maintain coherence with the implicit reference "the metric" and align with the memory overhead discussion in Section 5 paragraph 2. **Pass criterion:** No semantic contradictions introduced.

Edit Quality: Paragraph provides balanced discussion of memory overhead, adds at least one concrete mitigation strategy (e.g., "implementing lazy loading of intermediate matrices"), and improves clarity without altering factual content. **Pass criterion:** Instruction requirements satisfied, no factual errors.

C.2.2 Example 2: Content Addition with Dependencies

Scenario: Technical document (5K tokens) explaining credit risk calculations. Edit instruction: "Insert a concrete example illustrating the Probability of Default (PD) component of the Expected Credit Loss calculation at the end of the second paragraph of Section 3.1."

Document Structure:

- Explicit references: Figure 2 cited in Overview, Section 3.1, Section 4.1, Section 6; Equation (1) in Section 3.1
- Implicit references: "the preceding computation" in Section 3.2 refers to Equation (1)
- Semantic dependencies: Section 6 compares Section 3 methodology with legacy approaches

Metric Evaluation:

Dependency Detection (Explicit): System identifies explicit references to Figure 2 in 4 locations, Table 1 in 3 locations, Equation (1) in Section 3.1, and Definition PD in Section 2.2. **Pass criterion:** All explicit

references in relevant sections identified.

Dependency Detection (Implicit): System resolves that "the preceding computation" in Section 3.2 paragraph 2 refers to Equation (1) in Section 3.1, ensuring the new example aligns with this implicit reference. **Pass criterion:** Implicit reference correctly resolved.

Dependency Detection (Semantic): System recognizes that Section 6 paragraph 2 semantically compares Section 3 methodology with legacy spreadsheet approaches, despite no explicit citations. The new example may affect this comparison. **Pass criterion:** Semantic comparison relationship detected.

Token Efficiency: System should retrieve: Section 3.1 all paragraphs (300 tokens) + Definition PD from Section 2.2 (50 tokens) + Table 1 (60 tokens) + Equation (1) (40 tokens) + Figure 2 caption (50 tokens) = approximately 500 tokens. **Pass criterion:** Retrieved tokens $< 10\%$ of 5K document (500 tokens).

Reference Validity: All explicit references (Figure 2, Table 1, Equation (1), Definition PD, Section 2.2) must remain valid after inserting the new example. No renumbering should be triggered. **Pass criterion:** All explicit references still resolve correctly.

Consistency Preservation: The implicit reference "the preceding computation" in Section 3.2 and semantic comparisons in Section 6 remain accurate after example insertion. The new content must not contradict existing methodology descriptions. **Pass criterion:** No contradictions with implicit or semantic dependencies.

Edit Quality: The inserted example clearly demonstrates PD calculation using a simple numerical scenario (e.g., "For a borrower with credit score 650 and debt ratio 0.4, the estimated PD is 2.3%"), is positioned correctly after Section 3.1 paragraph 2, and uses consistent terminology. **Pass criterion:** Example is clear, concrete, and properly positioned.

C.2.3 Example 3: Cross-Reference Update

Scenario: API documentation (20K tokens) with code examples. Edit instruction: "Update the signature of the reserveRoom endpoint in Section 3.2 from reserveRoom(hotelId, guestInfo) to reserveRoom(hotel_id, guest_profile) and make sure every code example and reference throughout the document is updated consistently."

Document Structure:

- Explicit references: Figure 1 cited in 3 locations; Endpoint "/api/v1/reserveRoom" in 3 code listings; Listings 2 and 3 cited in 2 locations
- Implicit references: "this endpoint" in Section 4.1 refers to Section 3.2; "the previous signature" in Section 6 refers to old parameter names
- Semantic dependencies: Listing 3 builds on authentication flow; Section 5 compares reserveRoom vs cancelBooking

Metric Evaluation:

Dependency Detection (Explicit): System identifies all explicit references: Figure 1 in sections 1.2, 3.1, and 4.1; Endpoint "/api/v1/reserveRoom" in Listings 2, 3, and Section 5.1; Table 1 in sections 2.2, 4.1.3, and 4.2.4. **Pass criterion:** All explicit references to endpoint and parameters identified.

Dependency Detection (Implicit): System resolves: "this endpoint" in Section 4.1 paragraph 2 refers to Section 3.2 reserveRoom definition; "these examples" in Section 5 paragraph 2 refers to Listings 2 and 3; "the previous signature" in Section 6 paragraph 1 refers to reserveRoom(hotelId, guestInfo). **Pass criterion:** All implicit references correctly resolved.

Dependency Detection (Semantic): System infers: Listing 3 builds on authentication flow from Section 4.2; Section 5 paragraph 3 compares reserveRoom versus cancelBooking semantically; Section 4.1 paragraph 3 synthesizes Table 1 with Listing 2 output. **Pass criterion:** Semantic dependencies identified.

Token Efficiency: System should retrieve: Section 3.2 endpoint definition (200 tokens) + Listings 2 and 3 (400 tokens each) + all paragraphs referencing endpoint or parameters (600 tokens) + changelog line (50 tokens) = approximately 1,650 tokens. **Pass criterion:** Retrieved tokens $< 10\%$ of 20K document (2,000 tokens).

Reference Validity: After updating parameter names from camelCase to snake_case, all explicit references must still resolve correctly: Figure 1 references unchanged, endpoint path unchanged, listing citations unchanged. **Pass criterion:** All explicit references remain valid.

Consistency Preservation: Implicit references ("this endpoint", "the previous signature") and semantic comparisons remain meaningful after parameter name changes. Text referring to "hotelId" must be updated to "hotel_id" throughout. **Pass criterion:** Terminology changes propagated to all dependent text.

Edit Quality: reserveRoom signature updated in definition and all code examples (Listings 2, 3, and inline examples); code compiles without errors; explanatory text aligns with new parameter names using snake_case convention consistently. **Pass criterion:** All occurrences updated, code valid, terminology consistent.

C.3 Metric Computation

For each test case, we compute metrics through automated validation:

Dependency Detection Metrics: We programmatically compare the set of dependencies identified by the system against the ground truth dependency annotations in each test case. For explicit references, we extract all citations using pattern matching (e.g., "Figure \d+", "Section \d+.\d+"). For implicit references, we verify that anaphoric expressions resolve to the correct antecedents through coreference chains. For semantic dependencies, we check that the graph contains edges representing conceptual relationships marked in the ground truth.

Token Efficiency: We measure the total number of tokens in the context provided to the model for each edit operation and compute the ratio to total document size. This is measured directly through tokenization of the retrieved content.

Reference Validity: We parse the edited document to extract all explicit references and verify that each reference target exists in the updated document. For example, a citation to "Figure 3" must correspond to an actual figure with that identifier. References are validated through programmatic document structure analysis.

Consistency Preservation: We check that implicit references remain resolvable and that terminology changes propagate correctly. For example, if a definition is modified, we verify that dependent sections using that term are also updated. We also check semantic coherence by comparing embeddings of related sections to ensure similarity remains above threshold $\theta = 0.7$.

Edit Quality: For a subset of 200 test cases, we compare system outputs against gold standard human edits. We assess whether the modification achieves the instruction goal, maintains appropriate style and tone, preserves factual accuracy, and introduces no unintended changes. Quality scores are aggregated across these dimensions.

Final scores for each metric represent the percentage of test cases that meet the pass criterion, averaged across all applicable test cases and models.

D Benchmark Construction

We construct a comprehensive benchmark to evaluate consistency preservation, efficiency, and scalability in agentic document editing. This appendix describes the test scenario taxonomy, document generation process, and evaluation framework.

D.1 Benchmark Overview

Our benchmark comprises 1,900 test cases spanning seven categories of editing operations, designed to systematically evaluate agent performance across diverse document structures, edit complexities, and document sizes. Each test case consists of: (1) a source document with explicit structure and dependencies, (2) a natural language editing instruction, (3) ground-truth expected modifications, and (4) validation criteria for consistency, quality, and efficiency.

The benchmark is designed around the core challenges of document editing identified in Section 1: maintaining cross-references during modifications, preserving semantic consistency across dependent sections, and scaling efficiently with document size. Test cases range from localized refinements with

minimal dependencies to structural reorganizations requiring global updates, enabling fine-grained analysis of system capabilities.

D.2 Test Scenario Taxonomy

We organize test scenarios into seven categories, each targeting specific aspects of dependency-aware editing.

D.2.1 Category 1: Local Refinement Edits

Local refinements modify specific paragraphs or sections with minimal cross-document dependencies. These scenarios evaluate whether systems can apply targeted improvements without unnecessary context retrieval or unintended modifications to unrelated content.

Subcategories:

- **Paragraph and section refinement (120 cases):** Instructions targeting clarity improvements, argument strengthening, or content simplification within localized scope. Examples include "Improve the clarity of the second paragraph in Section 3.2" and "Make the proof in Section 4.1 more concise while preserving logical steps." These test the system's ability to retrieve minimal context (target paragraph plus containing section) while avoiding unnecessary full-document processing.

- **Local content enhancement (80 cases):** Adding examples, definitions, or explanatory text within sections. Examples include "Add a concrete example demonstrating the algorithm after its description in Section 3.1" and "Insert a worked example illustrating the calculation process." These test whether systems can identify the appropriate insertion point and maintain local coherence without disrupting document structure.

- **Error correction (60 cases):** Fixing typos, incorrect values, broken references, or outdated information. Examples include "Fix the typo in the third paragraph of Section 2.1" and "Correct the incorrect parameter value in the code example." These test precision in locating and modifying specific elements without introducing unintended changes.

Expected behavior: Systems should retrieve only the target section and immediate structural context, typically 200-800 tokens. Consistency requirements are minimal since dependencies are local. Token efficiency is the primary evaluation focus.

D.2.2 Category 2: Cross-Reference Dependent Edits

Cross-reference edits modify content that is referenced by or depends on multiple other sections, requiring the system to identify and update all dependent locations.

Subcategories:

- **Figure and table caption updates (80 cases):** Modifying captions for elements referenced throughout the document. Example: "Update Figure 3's caption to clarify the x-axis label" where Figure 3 appears in 4-6 sections. Tests whether the system identifies all referencing sections and includes them in context to verify descriptions remain accurate.

- **Definition and terminology changes (70 cases):** Revising definitions or changing terminology used consistently across the document. Example: "Revise the definition of 'token efficiency' in Section 2.1 to add precision" where the term appears in 5-8 locations. Tests both explicit reference tracking (e.g., "see Section 2.1 for the definition") and implicit usage (paragraphs relying on the definition without citation).

- **API and code example updates (60 cases):** Changing function signatures, parameter names, or code structures used in multiple examples. Example: "Update the reserveRoom endpoint signature from reserveRoom(hotelId, guestInfo) to reserveRoom(hotel_id, guest_profile) throughout all code examples." Tests tracking of both explicit code references and implicit dependencies where examples build on earlier code.

- **Theorem and mathematical content updates (50 cases):** Modifying theorems, lemmas, or equations referenced in proofs or derivations. Example: "Revise Theorem 2's bound from O(n log n) to O(n) and verify dependent proofs remain valid." Tests mathematical dependency tracking and semantic coherence validation.

- **Citation and bibliography updates (40 cases):** Modifying citation formats, adding/removing references, or updating bibliography entries. Example: "Change all citations to use APA format instead of ACM format." Tests systematic updates across the document while maintaining reference integrity.

Expected behavior: Systems should identify all sections containing explicit references (via `REFERENCES` edges), implicit dependencies (via `DEPENDS` edges), and semantically related content (via `RELATED` edges). Context retrieval should be proportional to the number of dependencies (typically 800-2,000 tokens for elements with 4-6 references). Consistency preservation is the primary evaluation focus.

D.2.3 Category 3: Structural Reorganization

Structural reorganizations involve section reordering, merging, splitting, or hierarchy changes that require updating all cross-references and maintaining document coherence.
Subcategories:

- **Section reordering (60 cases):** Moving sections to different positions while updating numbering and cross-references. Example: "Move Section 5 to appear before Section 3" requires updating all references like "as described in Section 5" and renumbering subsequent sections. Tests whether systems track both forward and backward references.

- **Section merging and splitting (50 cases):** Combining or dividing sections while maintaining hierarchy. Example: "Split Section 4 into two sections: 'Methodology Overview' and 'Implementation Details"' requires updating section numbers, redistributing subsections, and adjusting cross-references. Tests hierarchical consistency maintenance.

- **Section deletion and addition (30 cases):** Removing or inserting sections while updating references. Example: "Delete Section 3.2 and merge its content into Section 3.1" requires identifying all references to 3.2, updating them to reference 3.1, and renumbering subsequent subsections. Tests dangling reference detection.

- **Hierarchy restructuring (20 cases):** Changing nesting levels and parent-child relationships. Example: "Promote all subsections of Section 4 to be top-level sections" requires restructuring the containment hierarchy and updating all hierarchical references. Tests `CONTAINS` edge maintenance.

Expected behavior: Systems should retrieve the entire structural hierarchy affected by the reorganization, including all sections with references to moved/modified sections. Token usage varies significantly (1,500-4,000 tokens) based on reorganization scope. Both structural integrity and reference validity are critical evaluation dimensions.

D.2.4 Category 4: Content Addition with Integration

Content addition scenarios require inserting new material that integrates coherently with existing structure, maintaining appropriate cross-references and dependencies.
Subcategories:

- **Example and illustration addition (60 cases):** Inserting examples, case studies, or visual elements that reference existing content. Example: "Add a detailed walkthrough example demonstrating the three-phase workflow described in Section 2" requires understanding the workflow definition and creating consistent terminology and structure.

- **New subsection addition (40 cases):** Adding complete subsections that extend existing sections. Example: "Insert a new subsection comparing the proposed approach with the baseline method after Section 4.1" requires understanding both approaches and maintaining consistent comparison criteria.

- **Cross-cutting content addition (20 cases):** Adding content that synthesizes or references multiple existing sections. Example: "Add a paragraph synthesizing findings from Sections 3, 4, and 5 at the beginning of Section 6" requires retrieving context from all referenced sections and maintaining semantic coherence.

Expected behavior: Systems should retrieve all sections referenced in the instruction plus semantically related content to ensure stylistic and terminological consistency. Context retrieval should scale with the number of sections involved (800-1,800 tokens). Coherence with existing content is the primary evaluation focus.

D.2.5 Category 5: Multi-Round Iterative Editing

Iterative editing scenarios simulate realistic collaborative writing workflows with sequences of 3-5 dependent edits, where each edit builds on previous modifications.

Subcategories:

- **Sequential refinement workflow (40 cases):** Progressive refinements to improve content quality. Example: Three-step sequence: (1) "Revise Section 3.2 to emphasize the key contribution," (2) "Add supporting experimental evidence after the revised paragraph," (3) "Strengthen connections between Section 3.2 and the results in Section 5." Tests whether the graph updates incrementally and subsequent edits have access to previous modifications.

- **Iterative restructuring workflow (35 cases):** Multiple structural changes building on each other. Example: Three-step sequence: (1) "Move Section 4 earlier," (2) "Merge it with Section 2," (3) "Add new content to the merged section." Tests incremental graph updates and consistency maintenance across multiple structural operations.

- **Content development workflow (35 cases):** Incremental content addition across sections. Example: Three-step sequence: (1) "Add a definition of 'context efficiency' in Section 2," (2) "Add an example using this definition in Section 3," (3) "Reference both the definition and example when discussing results in Section 5." Tests dependency creation and tracking across editing rounds.

- **Complete revision workflow (50 cases):** End-to-end revision cycles combining multiple edit types. Example: Four-step sequence: (1) "Revise the introduction," (2) "Update Figure 2's caption," (3) "Strengthen the argument in Section 4," (4) "Reorganize section order." Tests sustained consistency over complex editing workflows.

Expected behavior: Systems must update the dependency graph after each edit and use the updated graph for subsequent operations. Token usage should remain constant per edit even as document state evolves. Overall consistency across the editing sequence is the primary evaluation focus.

D.2.6 Category 6: Scalability Test Scenarios

Scalability scenarios apply identical editing operations to documents of varying lengths (5K, 10K, 20K, 50K, 100K tokens) to test whether token usage remains constant as document size increases.

Test configurations:

- **Local edits across sizes (20 operations × 5 sizes = 100 cases):** Simple refinements like "Improve clarity of a paragraph in Section 3.2." Expected token usage should remain constant (500 tokens) regardless of document size.

- **Cross-reference edits across sizes (10 operations × 5 sizes = 50 cases):** Dependency-heavy edits like "Update Figure 3's caption" where the figure is referenced in 4-6 sections (constant across document sizes). Expected token usage should remain constant (1,500 tokens) even though total document size varies by 20×.

- **Structural edits across sizes (10 operations × 5 sizes = 50 cases):** Reorganizations like "Move Section 5 to appear before Section 3." Expected token usage should scale sublinearly, retrieving only affected sections rather than the entire document.

Expected behavior: Token usage should remain constant or scale sublinearly with document size, demonstrating $O(1)$ rather than $O(|D|)$ behavior. This validates the core efficiency claim that dependency-aware retrieval enables scalable editing.

D.2.7 Category 7: Consistency Critical Scenarios

Consistency-critical scenarios are designed as pass/fail tests where any consistency violation (broken reference, terminology mismatch, semantic contradiction) constitutes failure.
Subcategories:

- **Cascading reference updates (50 cases):** Edits that should trigger automatic updates in all dependent locations. Example: "Rename Section 4 to 'Experimental Methodology' " should update all references like "see Section 4" or "as described in the Methods section." Failure modes include: missing references, partially updated references, or broken cross-reference links.
- **Dependency chain edits (30 cases):** Modifications to foundational content that affects derived content. Example: "Modify the definition of 'retrieval precision' in Section 2 to include a threshold parameter" should propagate to all calculations, examples, and results using this metric. Failure modes include: orphaned dependencies, inconsistent terminology, or contradictory statements.
- **Structural integrity tests (40 cases):** Reorganizations that could violate hierarchy if not handled correctly. Example: "Delete Section 3 which contains subsections 3.1, 3.2, 3.3" should properly handle the child sections (either delete them or promote them). Failure modes include: orphaned subsections, broken containment relationships, or incorrect section numbering.

Expected behavior: Systems must achieve 100% consistency on these test cases. Any broken reference, terminology inconsistency, or structural violation is considered a failure. These cases validate the reliability of dependency tracking and consistency verification mechanisms.

D.3 Document Generation Process

We generate synthetic documents with controlled structure and dependencies to enable systematic evaluation.
Document structure: Each document follows a hierarchical structure with 5-12 top-level sections, 2-5 subsections per section, and 3-8 paragraphs per subsection. Documents include structural elements (sections, paragraphs), referenced elements (figures, tables, equations), and semantic content (definitions, examples, proofs). We ensure controlled dependency density: each figure/table is referenced by 4-6 paragraphs, each definition is used in 5-8 locations, and each theorem is cited by 2-4 dependent results.
Size variations: We generate five document size categories by varying content length while maintaining consistent structural complexity:

- **Small (5K tokens):** 5 sections, 15 subsections, 60 paragraphs
- **Medium (10K tokens):** 7 sections, 21 subsections, 100 paragraphs
- **Large (20K tokens):** 9 sections, 30 subsections, 150 paragraphs
- **Very large (50K tokens):** 11 sections, 40 subsections, 250 paragraphs
- **Extreme (100K tokens):** 12 sections, 50 subsections, 400 paragraphs

Dependency injection: We programmatically inject dependencies during document generation to ensure reproducibility. Explicit references are inserted using templates ("as shown in Figure X", "see Section Y"). Implicit dependencies are created through pronoun chains and contextual references ("This approach...", "The aforementioned method..."). Semantic relationships emerge from topical coherence within sections and subsections discussing related concepts.
Ground truth: For each test case, we generate ground-truth expected modifications by applying the edit operation programmatically to the document structure. For reference-updating edits, we identify all

affected locations through dependency analysis. For structural reorganizations, we recompute section numbering and update all cross-references. For content additions, we specify insertion points and expected integration patterns. This ground truth enables automated evaluation metrics described below.

D.4 Benchmark Statistics

Table 4 summarizes the distribution of test cases across categories and complexity levels.

Category	Cases	Avg. Deps	Avg. Tokens
Local Refinement	260	1.2	650
Cross-Reference	300	5.8	1,520
Structural Reorg.	160	8.4	2,350
Content Addition	120	3.6	980
Iterative Editing	160	4.2	1,180
Scalability Tests	780	2-6	500-2,800
Consistency Critical	120	6.5	1,640
Total / Average	**1,900**	**4.7**	**1,380**

Table 4: Benchmark statistics. "Avg. Deps" indicates average number of dependency edges per target element. "Avg. Tokens" indicates expected context size for optimal retrieval. Scalability tests span multiple document sizes, hence the range.

Dependency complexity distribution: 35% of test cases involve localized edits with $\leq$2 dependencies (testing efficiency), 45% involve moderate dependencies with 3-8 dependent sections (testing consistency), and 20% involve high-complexity scenarios with $>$8 dependencies or structural reorganizations (testing scalability and robustness). This distribution ensures comprehensive coverage of real-world editing complexity.

Document type distribution: Documents are evenly distributed across four domains to test generalization: technical reports (25%, emphasis on figures and algorithms), research papers (25%, emphasis on theorems and proofs), API documentation (25%, emphasis on code examples and specifications), and policy documents (25%, emphasis on cross-referencing and hierarchical structure). Domain diversity ensures the evaluation does not overfit to specific document styles.

D.5 Evaluation Protocol

For each test case, we measure four dimensions aligned with our optimization objectives (Section 3):

1. Consistency preservation: We compute three sub-metrics: (a) *Reference validity* checks whether all cross-references resolve correctly after edits (e.g., "see Section 4" still points to an existing Section 4), (b) *Terminology consistency* verifies that all usages of modified terms or definitions remain aligned, and (c) *Semantic coherence* ensures no contradictions are introduced between dependent sections. Consistency is measured as the percentage of test cases where all three sub-metrics pass.

2. Token efficiency: We measure tokens per edit as the sum of input context tokens (document content provided to the LLM) and output tokens (generated modifications). For scalability tests, we additionally compute the scaling coefficient by fitting token usage as a function of document size and verifying it approaches $O(1)$ rather than $O(|D|)$.

3. Edit quality: We assess whether modifications correctly address the instruction using LLM-as-judge evaluation. A subset of 200 test cases includes human-annotated gold-standard edits for validation. Quality is measured as the percentage of test cases where the edit satisfies the instruction without introducing errors.

4. Overall pass rate: We compute the percentage of test cases passing all criteria (consistency, efficiency within expected bounds, and quality), representing end-to-end system reliability.

Automated validation: We implement programmatic validators for consistency metrics:

- **Reference validator:** Parses all cross-references ("Section X", "Figure Y", "Theorem Z") and verifies targets exist with correct identifiers

- **Terminology validator:** Extracts key terms from definitions and verifies consistent usage across all mentions
- **Hierarchy validator:** Checks section numbering sequences, parent-child relationships, and containment consistency
- **Semantic validator:** Uses LLM-based entailment checking to detect contradictions between dependent sections

These automated validators enable large-scale evaluation across 1,900 test cases while maintaining reproducibility and consistency with human judgments (validated on the 200-case human-annotated subset with 94% agreement).

E Pseudocode for LEDGER Algorithm

E.1 GUIDE Algorithm Details

We provide detailed algorithms for the three core components of LEDGER: semantic graph construction, dependency-aware context retrieval, and incremental consistency verification.

E.1.1 Semantic Graph Construction

Algorithm 1 describes how LEDGER constructs and maintains the semantic dependency graph.

The algorithm constructs a semantic graph in three steps. First, it extracts explicit references through pattern matching for citations like "Theorem 2", "Equation (5)", and "Figure 1" (lines 6-13). Second, it detects implicit references through coreference resolution to identify anaphoric expressions like "this bound" or "the lemma above" (lines 14-18). Third, it computes semantic embeddings using DistilBERT and connects nodes with high cosine similarity ($\theta = 0.7$) that lack explicit connections (lines 19-28). The graph construction completes in 2-3 seconds for a typical 50-page paper with 150 semantic units, while incremental updates after edits complete in under 200ms by reprocessing only modified nodes.

E.1.2 Dependency-Aware Context Retrieval

Algorithm 2 describes how LEDGER efficiently retrieves relevant context for each edit operation.

The algorithm retrieves context in four steps. First, it identifies target nodes either through explicit mentions (e.g., "Section 3.2") or semantic similarity for ambiguous instructions (lines 4-11). Second, it expands to include all nodes connected by dependency edges, capturing both upstream context (what the target relies on) and downstream impacts (what depends on the target) (lines 12-16). Third, it prioritizes nodes using an additive scoring scheme: direct targets (100 points), explicit references (50 points, critical for consistency), implicit dependencies (30 points), and semantic relations (10 points, optional context) (lines 17-28). Fourth, it packs content in priority order until the token budget B is exhausted (lines 29-38). This approach typically retrieves 10-15% of document tokens while ensuring all critical dependencies are included, as demonstrated in the Appendix example where only 1,200 of 10,000 tokens were needed.

E.1.3 Consistency Verification

Algorithm 3 describes how LEDGER validates document consistency after edits.

F Potential Risk

Risks. The proposed method's effectiveness depends critically on the accuracy of LLM-based dependency extraction protocols. Errors in identifying explicit references, implicit dependencies, or semantic relationships during graph construction propagate through subsequent editing operations, potentially causing the system to miss critical dependencies or retrieve irrelevant context. The extraction protocols require careful prompt engineering and may exhibit inconsistent performance across document domains, with technical specifications having clearer structural patterns than creative or conversational content. Additionally, the graph construction and incremental update processes introduce computational overhead: while amortized across multiple edits, the initial graph construction for very large documents (>100,000 tokens) or documents with dense interconnections can be time-consuming. The semantic similarity

threshold for RELATED edges represents a design choice that balances precision and recall but may require domain-specific tuning. Furthermore, the method assumes document structure can be reliably parsed into DOM trees, which may fail for poorly formatted documents, legacy formats, or content with inconsistent markup. Finally, the framework's reliance on external graph memory introduces a synchronization challenge: if the graph becomes desynchronized from the document state due to external edits or system failures, consistency guarantees no longer hold until the graph is reconstructed.

G Ethical Considerations

The deployment of AI agents for automated document editing raises several ethical concerns that warrant careful consideration. First, the system's ability to maintain cross-document consistency and propagate changes efficiently could facilitate unauthorized modifications or manipulation of collaborative documents if access controls are inadequate, particularly in settings involving multiple stakeholders or version-controlled content. Second, while the framework assists human editors, over-reliance on automated editing agents may reduce critical human oversight, potentially allowing errors or inappropriate modifications to propagate undetected through dependency chains. The LLM-based dependency extraction and editing components may also inherit biases present in their training data, potentially affecting how relationships are identified or how edits are formulated across different document types, domains, or writing styles. Third, the system processes document content to construct semantic graphs and generate embeddings, raising privacy concerns when applied to sensitive or confidential documents—organizations must ensure appropriate data handling practices and consider on-premises deployment for sensitive applications. Fourth, the framework's effectiveness in maintaining consistency might create false confidence in fully automated editing workflows, whereas human judgment remains essential for evaluating semantic appropriateness, stylistic coherence, and domain-specific correctness that automated systems cannot fully capture. We emphasize that LEDGER is designed as an assistive tool augmenting human editors rather than replacing them, and recommend that deployed systems include human-in-the-loop verification for critical editing operations, maintain comprehensive audit trails, and provide transparency about which modifications were AI-generated versus human-authored.

Algorithm 1 Semantic Graph Construction

Input: Document D, Previous graph $G = (V, E)$
Output: Updated graph $G' = (V', E')$

$V' \leftarrow$ ParseSemanticUnits(D) {Extract sections, paragraphs, equations, figures}

// Step 1: Build explicit reference edges
$E_{\text{explicit}} \leftarrow \emptyset$
for each node $v_i \in V'$ **do**
  refs $\leftarrow$ ExtractReferences(v_i) {e.g., "Theorem 2", "Equation (5)", "Figure 1"}
  for each reference r in refs **do**
    $v_j \leftarrow$ ResolveTarget(r, V')
    $E_{\text{explicit}} \leftarrow E_{\text{explicit}} \cup \{(v_i, v_j, \texttt{REFERENCES})\}$
  end for
end for

// Step 2: Build implicit reference edges
$E_{\text{implicit}} \leftarrow \emptyset$
anaphora $\leftarrow$ DetectAnaphoricExpressions(V') {e.g., "this bound", "the lemma"}
for each (v_i, expr, v_j) in anaphora **do**
  $E_{\text{implicit}} \leftarrow E_{\text{implicit}} \cup \{(v_i, v_j, \texttt{DEPENDS})\}$
end for

// Step 3: Build semantic dependency edges
$E_{\text{semantic}} \leftarrow \emptyset$
for each node $v_i \in V'$ **do**
  v_i.embedding $\leftarrow$ Embed(Summary(v_i)) {DistilBERT embeddings}
end for
for each pair (v_i, v_j) where $i \neq j$ **do**
  sim $\leftarrow$ CosineSimilarity(v_i.embedding, v_j.embedding)
  if sim $> \theta$ **and** no explicit/implicit edge exists **then**
    $E_{\text{semantic}} \leftarrow E_{\text{semantic}} \cup \{(v_i, v_j, \texttt{RELATED})\}$
  end if
end for

$E' \leftarrow E_{\text{explicit}} \cup E_{\text{implicit}} \cup E_{\text{semantic}}$
return $G' = (V', E')$

Algorithm 2 Dependency-Aware Context Retrieval

Input: Edit instruction $\mathcal{E}$, Graph $G = (V, E)$, Token budget B
Output: Context $\mathcal{C} \subseteq V$

// Step 1: Identify target nodes
$\mathcal{N}_{\text{target}} \leftarrow \emptyset$
if $\mathcal{E}$ specifies explicit target (e.g., "Section 3.2") **then**
  $\mathcal{N}_{\text{target}} \leftarrow \{\text{ResolveNode}(\mathcal{E}.\text{target}, V)\}$
else
  $e_{\mathcal{E}} \leftarrow \text{Embed}(\mathcal{E}.\text{instruction})$
  $\mathcal{N}_{\text{target}} \leftarrow \text{TopKSimilar}(e_{\mathcal{E}}, V, k = 3)$
end if

// Step 2: Expand to dependencies
$\mathcal{N}_{\text{deps}} \leftarrow \mathcal{N}_{\text{target}}$
for each $v \in \mathcal{N}_{\text{target}}$ **do**
  $\mathcal{N}_{\text{deps}} \leftarrow \mathcal{N}_{\text{deps}} \cup \{u : (u, v, t) \in E\}$ {Upstream: what v depends on}
  $\mathcal{N}_{\text{deps}} \leftarrow \mathcal{N}_{\text{deps}} \cup \{u : (v, u, t) \in E\}$ {Downstream: what depends on v}
end for

// Step 3: Prioritize by dependency type
for each $v_i \in \mathcal{N}_{\text{deps}}$ **do**
  $\text{score}[v_i] \leftarrow 0$
  if $v_i \in \mathcal{N}_{\text{target}}$ **then**
    $\text{score}[v_i] \leftarrow \text{score}[v_i] + 100$ {Direct target}
  end if
  if $\exists(v_i, v_j, \texttt{REFERENCES}) \in E : v_j \in \mathcal{N}_{\text{target}}$ **then**
    $\text{score}[v_i] \leftarrow \text{score}[v_i] + 50$ {Explicit dependency (critical)}
  end if
  if $\exists(v_i, v_j, \texttt{DEPENDS}) \in E : v_j \in \mathcal{N}_{\text{target}}$ **then**
    $\text{score}[v_i] \leftarrow \text{score}[v_i] + 30$ {Implicit dependency}
  end if
  if $\exists(v_i, v_j, \texttt{RELATED}) \in E : v_j \in \mathcal{N}_{\text{target}}$ **then**
    $\text{score}[v_i] \leftarrow \text{score}[v_i] + 10$ {Semantic similarity}
  end if
end for

// Step 4: Pack context within token budget
$\text{ranked} \leftarrow \text{SortDescending}(\mathcal{N}_{\text{deps}}, \text{score})$
$\mathcal{C} \leftarrow \{\}$, $\text{tokens} \leftarrow 0$
for each v_i in ranked **do**
  $c_i \leftarrow \text{Content}(v_i)$
  if $\text{tokens} + |c_i| \leq B$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_i\}$
    $\text{tokens} \leftarrow \text{tokens} + |c_i|$
  else
    break
  end if
end for
return $\mathcal{C}$

Algorithm 3 Consistency Verification

Input: Modified nodes $\mathcal{N}_{\text{mod}} \subseteq V$, Updated document D', Graph $G = (V, E)$
Output: Violations $\mathcal{V}$, Updated graph G'

$\mathcal{V} \leftarrow \emptyset$

// Check 1: Reference integrity
for each $(v_i, v_j, \texttt{REFERENCES}) \in E : v_i \in \mathcal{N}_{\text{mod}}$ **do**
  if v_j does not exist in D' **then**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{(\texttt{BROKEN_REF}, v_i, v_j)\}$
  end if
end for

// Check 2: Terminology consistency
for each $v_i \in \mathcal{N}_{\text{mod}}$ **do**
  for each $(v_j, v_i, t) \in E : t \in \{\texttt{REFERENCES}, \texttt{DEPENDS}\}$ **do**
    orig_term $\leftarrow$ ExtractTerminology(v_i.before)
    new_term $\leftarrow$ ExtractTerminology(v_i.after)
    if orig_term $\neq$ new_term **then**
      $\mathcal{V} \leftarrow \mathcal{V} \cup \{(\texttt{TERM_CHANGE}, v_i, v_j, \text{orig}, \text{new})\}$
    end if
  end for
end for

// Check 3: Semantic coherence
for each $v_i \in \mathcal{N}_{\text{mod}}$ **do**
  $e_{\text{new}} \leftarrow$ Embed(Summary(v_i.after))
  for each $(v_i, v_j, \texttt{RELATED}) \in E$ **do**
    sim $\leftarrow$ CosineSimilarity(e_{new}, v_j.embedding)
    if sim $< \theta_{\text{min}}$ **then**
      $\mathcal{V} \leftarrow \mathcal{V} \cup \{(\texttt{SEM_DRIFT}, v_i, v_j)\}$
    end if
  end for
end for

// Update graph incrementally
$G' \leftarrow G$
for each $v_i \in \mathcal{N}_{\text{mod}}$ **do**
  Remove old edges incident to v_i from G'
  Recompute edges for v_i using Algorithm 1
  Add new edges to G'
end for
return $\mathcal{V}, G'$