ProChunker: Self-Optimizing Procedural Chunking for Agent-Ready Workflow Graphs

Soumajit Roy*, **Sayanta Adhikari, Vinayak S Puranik,**
Prakash Mandayam Comar
Amazon, India
r.soumajit@iitg.ac.in, {sayantaa, puranikv, prakasc}@amazon.com

Abstract

Can an LLM read a 30+ page Standard Operating Procedure (SOP) and produce a team of agents that reliably execute it? Industrial SOPs bury workflow logic in prose, leave inter-step dependencies implicit, and scatter referenced entities across sections, forcing planners to parse, sequence, and tool-map within one context window. We argue that the missing primitive is *procedural chunking*: decomposing a document into self-contained executable units. Unlike RAG-style chunking, each chunk must be independently executable and chunks must jointly form a valid DAG encoding data and process dependencies, turning a prose SOP into a compact orchestrable workflow graph. We present *ProChunker*, a three-phase self-optimizing LLM pipeline (Boundary Detection, Metadata Extraction, Dependency Construction) with *Stochastic Best-of-N* prompt exploration guided by five custom metrics (Boundary Clarity, Entity Reference Coverage, Agent Self-Containedness Agreement, Step Coverage, Graph Validity); we then fine-tune Qwen3-1.7B on the pipeline's own phase outputs, cutting latency by $\sim$27 min. Across three corpora (500 BioProBench protocols, six Industrial SOPs, 14 SOP-Bench documents), *ProChunker* improves the five-metric composite over the strongest baseline by $+0.036$ to $+0.111$ across corpora. Our study shows ProChunker increases agent construction success rate by $+17\%$ and resolves $>90\%$ of procedural ambiguity.

1 Introduction

Enterprise workflows are increasingly documented as Standard Operating Procedures (SOPs), long prose documents (15–30+ pages) describing conditional branches, cross-referenced policy tables, and multi-actor handoffs. Automating their execution with LLM-based agents (Wang et al., 2024; Xi et al., 2025) is attractive but unreliable in practice, not due to model capability gap but representation gap. SOPs are written for humans, with task sequences narrated in prose, inter-step dependencies left implicit, and referenced entities (templates, tables, tools) scattered across sections. A planner must simultaneously parse, infer dependencies, identify callable tools, and decide execution order within a single context window (Liu et al., 2024).

Existing Multi-Agent Systems (Hong et al., 2024; Wu et al., 2024) assume a structured task specification, while existing chunkers (Chase, 2022; Liu, 2022) optimize for *retrieval*. Neither addresses what a planner needs, a compact representation where each unit of work declares its actor, inputs, outputs, reference pointers, and position in a dependency graph, so reasoning happens at the chunk rather than document level. We call this *procedural chunking*, imposing three constraints absent in retrieval chunking, every chunk must be independently executable from its declared context, every referenced entity must be traceable to a declared source, and chunks must jointly form a valid DAG encoding data and process dependencies.

Building this representation at scale surfaces three coupled difficulties. First, single-call extraction does not fit industrial SOPs within context limits and produces boundary errors, dropped entities, and broken dependency chains when it does. Second, no existing benchmark measures whether a chunk is operationally self-contained or whether the resulting graph faithfully encodes the source process; chunk-size variance and retrieval F1 optimize form, not function. Third, chunking rules that work for one SOP often fail on another due to domain, process style, and writing conventions, so a static prompt is insufficient (Zhou et al., 2023; Yang et al., 2024; Khattab et al., 2024). We address these with a three-phase pipeline (Boundary Detection, Metadata Extraction, Dependency Construction) where each phase has its own optimizable prompt, five complementary metrics captur-

*Work done during an internship at Amazon.

ing semantic, operational, and graph-level quality, and a closed-loop optimizer routing phase-specific feedback and accumulating transferable refinement rules.

Our contributions are:

(i) A *three-phase LLM pipeline* for procedural chunking that preserves documented workflow edges while removing redundant data-flow shortcuts.

(ii) *Five custom metrics* (BC, ERC, ASA, SC, GV) for procedural chunking with LLM-as-judge components (Zheng et al., 2023; Liu et al., 2023).

(iii) A *Stochastic Best-of-N optimizer* with a rules engine routing per-metric feedback to the responsible phase, extending single-prompt optimizers (Zhou et al., 2023; Yang et al., 2024; Khattab et al., 2024) to a multi-prompt, multi-metric setting and *cross-document generalization* consolidating per-document rules into a static policy with minor performance drop.

(iv) *Fine-tuning Qwen3-1.7B* (Team, 2025) on the phase outputs that are generated in the primary pipeline, tracking its quality with $\sim$27 min latency reduction.

(v) *Evaluation on three corpora* (BioProBench, SOP-Bench, Industrial SOPs) showing $+0.036$ to $+0.111$ composite gains over the strongest baseline, with downstream procedural chunking lifting agent construction success rate by 17% and matching retrieval tuned baselines on RAG.

2 Related Works

Document chunking. Classical methods split text by fixed windows or recursive separators (LangChain (Chase, 2022)), or at sentence-embedding gaps (LlamaIndex (Liu, 2022)). LLM-driven variants followed: LumberChunker (Duarte et al., 2024) detects content shifts across sliding windows, QChunker (Zhao et al., 2026) runs a four-stage QA debate, MoC (Zhao et al., 2025) routes to granularity-specific experts, and HiChunk (Lu et al., 2025) builds multi-level boundaries iteratively. All optimize retrieval relevance, judged by token coherence or QA accuracy, and omit structural annotations like actors, inputs, outputs, or inter-chunk dependencies needed for workflow graph construction.

Prompt optimization and LLM-judge evaluation. Automatic prompt tuning has shifted to LLM-driven refinement: APE (Zhou et al., 2023) proposes and scores prompt variants, OPRO (Yang et al., 2024) rewrites prompts from past prompt–score trajectories, and DSPy (Khattab et al., 2024) compiles declarative pipelines via few-shot optimization. Each targets a single prompt against one scalar, ignoring pipelines with interacting prompts and multiple evaluation signals. LLM-as-judge evaluation (Liu et al., 2023) is now standard but risks circularity, when generator and evaluator share a model family, scores can favor fluency over correctness.

3 Proposed Method: ProChunker

Problem Definition. We extract an executable workflow graph from an unstructured document, assuming it is *self-contained* (all required information is present). This decomposes into: (i) semantically meaningful segmentation of actionable chunks, and (ii) construction of a directed execution graph over these chunks by resolving inter-chunk dependencies. We further hypothesize that multi-agent construction systems produce higher-quality agents when given a structured workflow graph rather than the raw document. Formally, for an agent construction system $\mathcal{A}$, we aim to show $\mathcal{A}(\mathcal{G}) \succ \mathcal{A}(\mathcal{S})$, where $\succ$ denotes superiority under agent constructability metrics, $\mathcal{G}$ is the extracted workflow graph with chunks, and $\mathcal{S}$ is the raw document.

3.1 Pipeline Formulation

Let $\mathcal{S}$ denote an unstructured document and $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ the target workflow graph, where nodes $\mathcal{V} = \{v_1, \ldots, v_n\}$ represent actionable chunks and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ capture directed dependencies. Each edge $e = (v_i, v_j)$ is typed as a *data dependency* (output of v_i consumed by v_j) or *process dependency* (v_j executes after v_i due to procedural ordering). The pipeline is $\mathcal{F}_{\boldsymbol{\pi}} : \mathcal{S} \to \mathcal{G}$, where $\boldsymbol{\pi} = (\pi_s, \pi_m, \pi_{dd}, \pi_{pd})$ is a *prompt policy* tuple governing each stage (Figure 1(a)):

(i) Segmentation f_{seg}: Partitions $\mathcal{S}$ into k non-overlapping segments $\{C_1, \ldots, C_k\}$, classifying each as *procedural* (contains ≥ 1 actor, ≥ 1 action verb, non-empty input/output sets) or *reference* (unique resources like tables/protocols consumed by procedural segments). Boundaries use linguistic/structural cues: explicit flow connectives, emergence of new named outputs, conditional branch-

ing, actor changes, and shifts from procedural to definitional discourse. Base prompt π_s in Appendix I.

(ii) Metadata Extraction f_{meta}: (a) For each *procedural* segments, extracts m_i comprising actor, action summary, input/output entity sets, gateway types and conditionals, and reference pointers. (b) For *reference* segments, extracts entity name, definition, and source text (indicating which procedural segment requires which reference). Schema and prompt π_m in Appendix I.

(iii) Dependency Resolution f_{dep}: Given $M = \{m_i\}_{i=1}^{k}$ and $\mathcal{S}$, builds $\mathcal{E}$ via two dependency types. *Data*: C_i depends on C_j when input$(C_i) \cap$ output$(C_j) \neq \emptyset$ (via π_{dd}). *Process*: C_i runs after C_j when $\mathcal{S}$ explicitly states so, regardless of data flow (via π_{pd}). We merge both graphs via transitive reduction on data edges only, then union with process edges; reference segments are untouched. Full procedure in Algorithm 2 (Appendix C.2), depicted in Figure 1(b). The pipeline composes as $\mathcal{G} = \mathcal{F}_{\boldsymbol{\pi}}(\mathcal{S}) = (\mathcal{V}, f_{\text{dep}}([f_{\text{meta}}(\mathcal{V}; \pi_m), \mathcal{S}]; \pi_{dd}, \pi_{pd}))$ where $\mathcal{V} = f_{\text{seg}}(\mathcal{S}; \pi_s)$.

Evaluation and iterative prompt optimization. We score $\mathcal{G}$ via a metric evaluator $\Phi(\mathcal{G}, \mathcal{S})$. A feedback generator f_{fb} produces structured diagnostics $\Psi = f_{\text{fb}}(\mathcal{G}, \Phi)$ identifying failure modes. An optimizer $f_{\text{opt}} : (\boldsymbol{\pi}, \Psi) \to \boldsymbol{\pi}'$ refines the policy. Refinement runs for K iterations: $\boldsymbol{\pi}^{(t+1)} = f_{\text{opt}}(\boldsymbol{\pi}^{(t)}, f_{\text{fb}}(\mathcal{G}^{(t)}, \Phi))$ where $\mathcal{G}^{(t)} = \mathcal{F}_{\boldsymbol{\pi}^{(t)}}(\mathcal{S})$, $t = 0, \ldots, K-1$, with early-stop on stagnation.

3.2 Evaluation Metrics Φ

Standard chunking metrics like chunk-size variance or token count (Lewis et al., 2020; Barnett et al., 2024) optimize form, not function: they cannot tell whether a boundary severs a causal dependency, whether an agent can execute a chunk without hidden context, or whether the graph faithfully encodes the source process. Our pipeline must produce chunks that are simultaneously semantically coherent, operationally self-contained, and graph-consistent, so we introduce five complementary metrics that double as optimizer feedback signals.

(i) Boundary Clarity (BC): For consecutive chunks (C_i, C_{i+1}), embed the last few sentences of C_i and first few of C_{i+1} via a sentence embedder; $BC_i = 1 - \cos(\mathbf{e}_{\text{tail}}, \mathbf{e}_{\text{head}})$. High values indicate genuine topic shift; low values signal mid-thought cuts. Document BC averages over chunks.

(ii) Entity Reference Coverage (ERC): An LLM extracts every domain entity (total) per chunk and batch-matches each against four pools: chunk's own outputs, parent-chunk outputs, declared inputs, and referenced definitions (covered). $ERC_i = |\text{covered}|/|\text{total}|$. Low ERC pinpoints dropped references or missing dependency edges. Document ERC averages over chunks.

(iii) Agent Self-Containedness Agreement (ASA): A restricted agent (chunk-only context) and a full-context agent (entire document) execute the same task prompt; a judge scores agreement from 1.0 (identical) to 0.0 (contradictory). Confirms isolated executability, catching implicit references and unresolved anaphora that static checks miss. Document ASA averages over chunks.

(iv) Step Coverage (SC): Step recall (fraction of extracted process steps landing in some chunk) and chunk precision (fraction of chunks containing ≥ 1 step); their harmonic mean yields an alignment F_1 penalizing both under-chunking (lost actions) and over-chunking (filler blocks). Computed at document level.

(v) Graph Validity (GV): Validating a full graph in one LLM call is unreliable at scale. Algorithm 3 decomposes the graph at perfect cut points (nodes S with cross$(S) = 0$), ensuring every ancestor's edges stay within the subgraph. Each dependency-complete subgraph is verified against the source for missing edges, spurious edges, and broken branching; scores aggregate to a document-level total.

Full operational details such as extraction prompts, LLM-judge rubrics, etc., are in Appendix B. subsection 3.3 and subsection 3.4 present three ProChunker variants with progressive latency improvements.

3.3 ProChunker-BoN: Stochastic Best-of-N Trajectory Search

We frame prompt optimization as a tree search over policies $\pi = (\pi_s, \pi_m, \pi_{dd}, \pi_{pd})$ (Figure 1(c)). Each node stores a policy with its manifest and metric scores; root is $\pi^{(0)}$, and each depth decides whether to expand or stop. Key steps:

(i) Composite objective: Φ returns five scores combined via weighted harmonic mean $s = \frac{\sum_i w_i}{\sum_i w_i / s_i}$, which collapses when any metric nears zero, preventing the optimizer from trading dimensions off.

(ii) Feedback and routing: The current-best manifest and metrics feed a feedback LLM return-

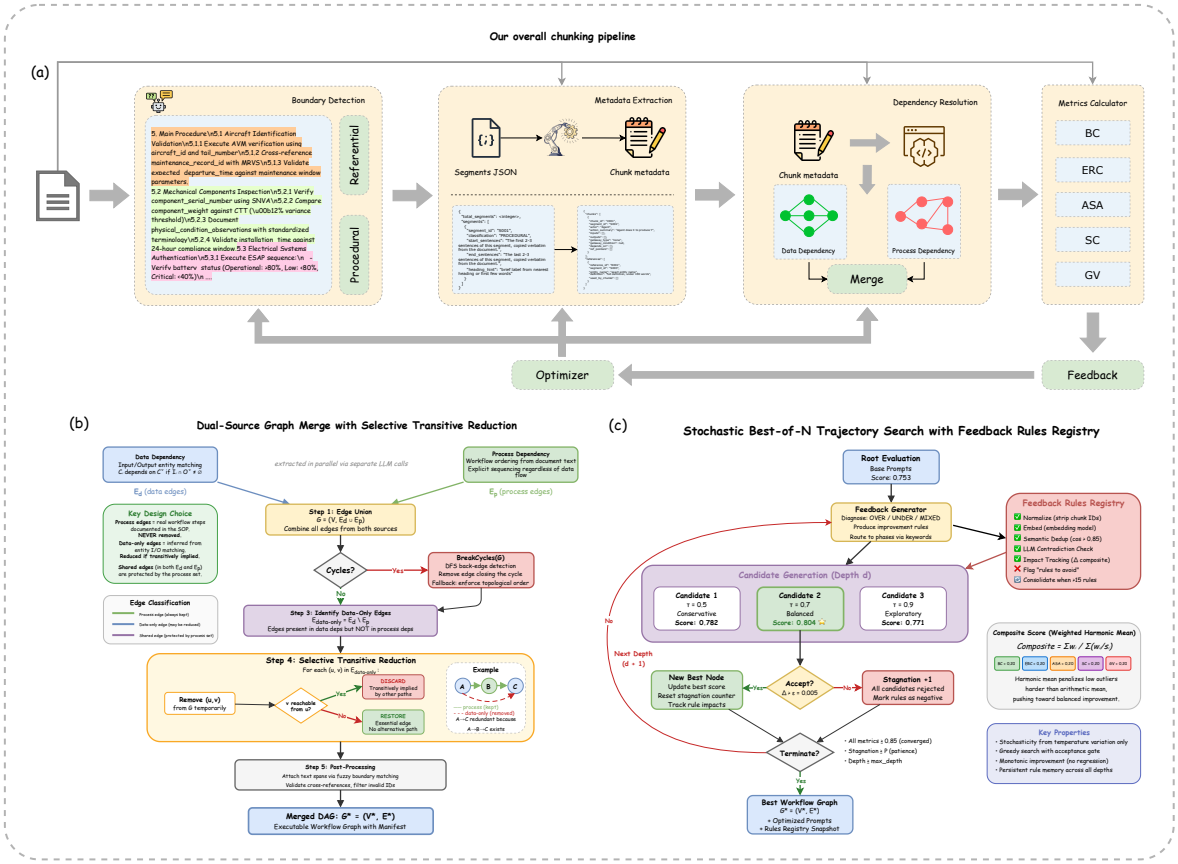


Figure 1: Overview of the proposed framework. **(a)** Three-phase chunking pipeline with five evaluation metrics and a closed-loop prompt optimization cycle. **(b)** Dual-source graph merge: data and process edges are unioned, cycles broken via DFS, and selective transitive reduction removes only data-only edges implied by alternative paths; process edges are never removed. **(c)** Stochastic Best-of-N search: three candidates at varied temperatures are evaluated per depth, accepted only on improvement, with a persistent rules registry preventing regression.

ing a diagnosis label, per-metric issue lists, per-chunk failure annotations, and short *refinement rules*. Feedback is partitioned across the four prompts via a fixed category-to-phase table (Appendix, Table 8); ASA failures route by mode: *reference_gap*$\rightarrow \pi_m$, *dependency_gap*$\rightarrow \pi_{dd}$, *boundary_error*$\rightarrow \pi_s$.

(iii) Rules Engine: The registry $\mathcal{R}$ serves as a shared memory of refinement rules across the search. As new rules arrive from feedback, they are periodically consolidated. Each rule maintains a running impact score(contribution of the rule towards the metric improvement) updated from its observed effect on accepted or rejected nodes, allowing useful patterns to accumulate and transfer across depths and documents.

(iv) Branch, evaluate, accept: At depth d, the search generates B candidates from current best. Each candidate s_n receives the same partitioned feedback but its optimizer samples at distinct temperature $\tau_n \in \{0.5, 0.7, 0.9\}$, yielding conservative-to-bold edits. Each runs the full pipeline. Let $\Delta = \max_n s_n - s^*$ where s^* is current best; with tolerance ϵ=0.005: accept as improvement if $\Delta > \epsilon$, accept within tolerance if $-\epsilon < \Delta \leq \epsilon$, else reject.

The tolerance band keeps small downhill moves reachable since LLM-judged metrics are noisy. Search terminates on convergence (all metrics $\geq \theta$), stagnation (P depths without strict improvement), or depth cap $D_{\max}$. Full procedure in Appendix C (Algorithm 1).

3.4 Generalized Prompts and Student Fine-Tuning

Best-of-N is per-document and costly at scale (60+ LLM calls, 30+ mins per document), so we compress it in two steps. First, across a training subset, we keep every refinement rule that produced an accepted improvement. Rules appearing in at least a fraction of per-document registries are folded into $\pi^{(0)}$ via one optimizer call per phase, yielding generalized policy π^{gen} (**ProChunker-Gen**). We compare ProChunker-Gen against ProChunker-BoN on

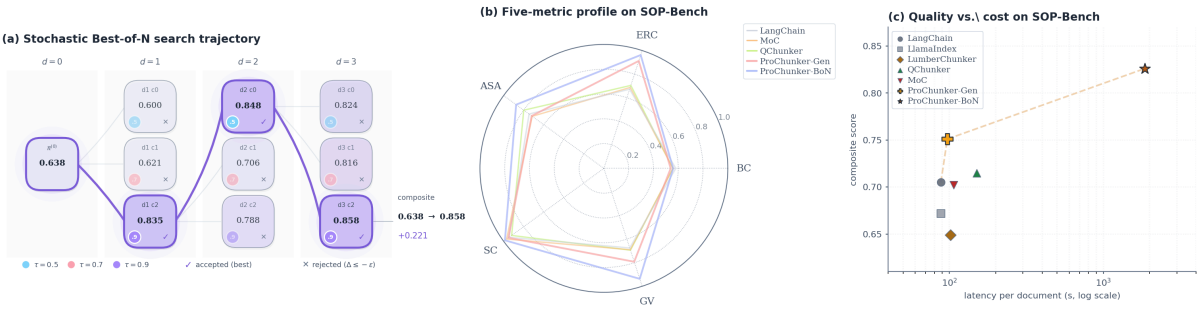


Figure 2: **(a)** Stochastic Best-of-N trajectory on SOP-Bench/`warehouse_package_inspection`: three candidates fan out at $\tau \in \{0.5, 0.7, 0.9\}$ per depth, ϵ-gate accepts the best (✓) and rejects the rest (×). **(b)** Five-metric profile on SOP-Bench: ProChunker-BoN dominates on ERC and GV while baselines saturate SC. **(c)** Quality vs. latency on SOP-Bench: ProChunker-Gen and -BoN sit on the Pareto frontier (dashed line)

a held-out set in Appendix F.2.

Generalized prompts still cost four LLM calls per document (∼10 mins). We therefore fine-tune Qwen3-1.7B (Team, 2025) on phase-wise input–output pairs generated under π^{gen}, with phase-specific instruction prompts, yielding **ProChunker-Distill**. To enhance fine-tuning, we also include the model's chain-of-thought thinking (captured in `<think>`) alongside its outputs as training targets. A single fine-tuned student (vs. four separate models) shares latent behavior and runs on one consumer GPU (inference ∼2 mins). Dataset construction, hyperparameters, and ablations in Appendices A and G. The full chain is a compression: ProChunker-BoN explores, ProChunker-Gen consolidates stable rules into a reusable prompt, and ProChunker-Distill moves that behavior into local weights.

4 Experiments

We evaluate along three axes: chunking quality on three SOP corpora under the metrics of subsection 3.3, RAG utility, and agent constructability. Configuration is in Appendix D; a BoN case study (Aircraft Inspection from SOP-Bench) is in Appendix J.

Setup. *BioProBench* (Liu et al., 2025) (14,675 protocols, 500 held-out), *Industrial SOPs* (six internal SOPs), and *SOP-Bench* (Nandi et al., 2026) (14 cross-domain SOPs: inspection, content moderation, customer service, fraud detection, warehousing). For proprietary reasons we refer to the large language models we use by role rather than by name: **Model 1** is the reasoning LLM that performs chunk generation, prompt optimization and all metric evaluation throughout the paper; **Model 2** and **Model 3** are frontier models from independent families used only to validate the metrics (§4.2); and **Model 4** generates answers in the downstream RAG evaluation. Models 1 and 2 are frontier-scale models of comparable capacity, while Models 3 and 4 are progressively smaller. The whole pipeline runs on Model 1; BoN uses B=3, P=2, $D_{\max}$=3, θ=0.85, w_i=0.2. Qwen3-1.7B (Team, 2025) is fine-tuned on BioProBench phase outputs only (Appendix G); Industrial SOPs and SOP-Bench serve as out-of-domain test sets for Distill. Six baselines (LangChain (Chase, 2022), LlamaIndex (Liu, 2022), LumberChunker (Duarte et al., 2024), QChunker (Zhao et al., 2026), MoC (Zhao et al., 2025), HiChunk (Lu et al., 2025)) emit chunk text only; a shared LLM enrichment supplies metadata, process steps, and a workflow graph so all five metrics apply. Adaptation in Appendix E.

4.1 Chunking Quality

Table 1 compares all methods across the three corpora on the full five-metric suite.

Table 1 presents chunking quality across three corpora. Our pipeline leads on ERC on the public corpora (Gen 0.882 on BioProBench, BoN 0.962 on SOP-Bench) and posts the largest ERC lift on Industrial SOPs (+0.312 for BoN), as workflow-aware chunks keep entity references grounded. On the composite, we beat the strongest baseline by +0.036 (BioProBench, vs. MoC) and +0.111 (SOP-Bench, vs. QChunker), driven by sharp GV lifts; on Industrial SOPs, BoN tops the strongest baseline composite delta by +0.086 (+0.269 vs. +0.184 for LumberChunker), with Distill (+0.220) outperforming Gen (+0.210). HiChunk achieves strong ASA and SC on the public corpora but collapses to single chunks, leaving BC and composite undefined. Ablations in Appendix F (Tables 9, 10). Figure 2 illustrates the search trajectory, five-metric profile, and quality-vs-cost frontier.

Table 1: Chunking quality across three corpora. HiChunk's BC is not reported (collapses to one chunk); its composite is undefined. ProChunker-BoN is not run on BioProBench. ProChunker-Distill is fine-tuned on BioProBench only; Industrial SOPs and SOP-Bench are out-of-domain. For Industrial SOPs, we report relative improvements (Δ) against a fixed baseline constant. Values are mean $\pm$ std across 5 runs. Best is **bold**, second underlined.

Method	BC	ERC	ASA	SC	GV	Comp.
BioProBench (500 held-out protocols)						
LangChain	$0.510_{\pm 0.012}$	$0.628_{\pm 0.015}$	$0.803_{\pm 0.020}$	$\underline{0.991}_{\pm 0.003}$	$0.738_{\pm 0.032}$	$0.698_{\pm 0.013}$
LlamaIndex	$\mathbf{0.608}_{\pm 0.010}$	$0.687_{\pm 0.015}$	$0.780_{\pm 0.023}$	$0.986_{\pm 0.004}$	$0.620_{\pm 0.031}$	$0.713_{\pm 0.013}$
LumberChunker	$\underline{0.591}_{\pm 0.013}$	$0.626_{\pm 0.016}$	$0.805_{\pm 0.019}$	$0.987_{\pm 0.003}$	$0.694_{\pm 0.035}$	$0.716_{\pm 0.013}$
QChunker	$0.465_{\pm 0.013}$	$0.709_{\pm 0.016}$	$\underline{0.832}_{\pm 0.022}$	$0.985_{\pm 0.004}$	$\underline{0.784}_{\pm 0.029}$	$0.709_{\pm 0.011}$
MoC	$0.545_{\pm 0.012}$	$0.676_{\pm 0.016}$	$0.823_{\pm 0.020}$	$0.986_{\pm 0.003}$	$0.782_{\pm 0.026}$	$0.733_{\pm 0.014}$
HiChunk	–	$0.655_{\pm 0.019}$	$\mathbf{0.872}_{\pm 0.019}$	$\mathbf{0.998}_{\pm 0.003}$	$0.142_{\pm 0.024}$	–
ProChunker-Gen	$0.508_{\pm 0.013}$	$\mathbf{0.882}_{\pm 0.010}$	$0.828_{\pm 0.023}$	$0.980_{\pm 0.004}$	$\mathbf{0.851}_{\pm 0.022}$	$\mathbf{0.769}_{\pm 0.011}$
ProChunker-Distill	$0.521_{\pm 0.014}$	$\underline{0.859}_{\pm 0.011}$	$0.826_{\pm 0.018}$	$0.974_{\pm 0.004}$	$0.697_{\pm 0.028}$	$\underline{0.740}_{\pm 0.012}$
Industrial SOPs (6 documents, Δ relative to a fixed baseline constant)						
LangChain	-0.111	$+0.144$	$+0.264$	$+0.453$	$+0.148$	$+0.123$
LlamaIndex	$+0.069$	$+0.137$	$+0.208$	**+0.500**	$+0.017$	$+0.152$
LumberChunker	$+0.078$	$+0.182$	$+0.259$	$+0.445$	$+0.075$	$+0.184$
QChunker	-0.006	$+0.136$	**+0.317**	$\underline{+0.460}$	$+0.183$	$+0.183$
MoC	$+0.074$	$+0.099$	$+0.303$	$+0.420$	$+0.050$	$+0.161$
HiChunk	–	$+0.018$	$+0.155$	$+0.175$	-0.308	–
ProChunker-BoN	**+0.112**	**+0.312**	$+0.296$	$+0.348$	**+0.396**	**+0.269**
ProChunker-Gen	$+0.072$	$\underline{+0.275}$	$+0.289$	$+0.397$	$+0.183$	$+0.210$
ProChunker-Distill	$\underline{+0.099}$	$+0.147$	$\underline{+0.313}$	$+0.348$	$\underline{+0.329}$	$\underline{+0.220}$
SOP-Bench (14 cross-domain documents)						
LangChain	$0.580_{\pm 0.015}$	$0.672_{\pm 0.026}$	$0.746_{\pm 0.032}$	$0.946_{\pm 0.008}$	$0.675_{\pm 0.051}$	$0.705_{\pm 0.020}$
LlamaIndex	$\underline{0.611}_{\pm 0.017}$	$0.777_{\pm 0.021}$	$0.738_{\pm 0.031}$	$\mathbf{1.000}_{\pm 0.004}$	$0.463_{\pm 0.050}$	$0.672_{\pm 0.021}$
LumberChunker	$\mathbf{0.655}_{\pm 0.017}$	$0.535_{\pm 0.029}$	$0.815_{\pm 0.031}$	$0.929_{\pm 0.008}$	$0.499_{\pm 0.058}$	$0.649_{\pm 0.022}$
QChunker	$0.557_{\pm 0.017}$	$0.704_{\pm 0.021}$	$0.799_{\pm 0.032}$	$0.920_{\pm 0.010}$	$0.694_{\pm 0.051}$	$0.715_{\pm 0.019}$
MoC	$0.558_{\pm 0.018}$	$0.687_{\pm 0.025}$	$0.715_{\pm 0.036}$	$0.970_{\pm 0.006}$	$0.690_{\pm 0.041}$	$0.702_{\pm 0.021}$
HiChunk	–	$0.834_{\pm 0.020}$	$\underline{0.843}_{\pm 0.028}$	$\mathbf{1.000}_{\pm 0.005}$	$0.136_{\pm 0.034}$	–
ProChunker-BoN	$0.559_{\pm 0.016}$	$\mathbf{0.962}_{\pm 0.011}$	$\mathbf{0.875}_{\pm 0.027}$	$\underline{0.987}_{\pm 0.006}$	$\mathbf{0.937}_{\pm 0.028}$	$\mathbf{0.826}_{\pm 0.013}$
ProChunker-Gen	$0.539_{\pm 0.019}$	$\underline{0.912}_{\pm 0.015}$	$0.720_{\pm 0.041}$	$0.952_{\pm 0.007}$	$0.792_{\pm 0.041}$	$0.751_{\pm 0.019}$
ProChunker-Distill	$0.542_{\pm 0.018}$	$0.830_{\pm 0.016}$	$0.808_{\pm 0.040}$	$0.925_{\pm 0.009}$	$\underline{0.797}_{\pm 0.065}$	$\underline{0.780}_{\pm 0.023}$

Table 2: **Judge independence on SOP-Bench (14 documents).** The frozen manifests of Table 1 are re-scored by three judges drawn from independent model families; generation is untouched and only the scorer changes. Model 1 is the pipeline model used throughout this paper; Models 2 and 3 are external. Columns: BoN/Gen (ours), LaCh LangChain, LuCh LumberChunker, QCh QChunker, MoC, HiCh HiChunk. Best per row **bold**.

Judge	BoN	Gen	LaCh	LuCh	QCh	MoC	HiCh
Entity Reference Coverage (ERC)							
Model 1	**0.962**	0.912	0.672	0.535	0.704	0.687	0.834
Model 2	**0.858**	0.814	0.706	0.669	0.735	0.687	0.702
Model 3	**0.835**	0.752	0.503	0.546	0.659	0.561	0.684
Graph Validity (GV)							
Model 1	**0.937**	0.792	0.675	0.499	0.694	0.690	0.136
Model 2	**0.686**	0.597	0.462	0.556	0.575	0.682	0.375
Model 3	**0.742**	0.582	0.479	0.450	0.610	0.676	0.447
Agent Self-Containedness Agreement (ASA)							
Model 1	**0.875**	0.720	0.746	0.815	0.799	0.715	0.843
Model 2	0.572	0.514	0.445	0.479	**0.577**	0.512	0.501
Model 3	0.586	0.561	0.538	**0.603**	0.554	0.559	0.556

4.2 Judge Independence

Three of the five metrics (ERC, ASA, GV) are LLM-scored, and in Table 1 the scorer is Model 1, the same model that produced the chunks. The reported margins could therefore reflect Model 1 preferring its own output rather than genuine structural quality. We test this in two ways using the independent-family judges Model 2 and Model 3: first by changing only the scorer, then by changing the scorer that the search itself optimises against.

Table 3: **Cross-family optimization.** The full Best-of-N search re-run with Model 3 scoring every candidate; generator and optimizer are unchanged. Root and best composites are both scored by Model 3, so the improvement is measured entirely by the independent evaluator. Industrial SOP absolute scores are withheld; only the improvement is reported. Per-document breakdown in Appendix F.3.

Corpus	n	Root	Best	Mean Δ	Improved
SOP-Bench	14	0.723	0.806	$+0.082$	13/14
Industrial SOPs	6	—	—	$+0.056$	4/6

Swapping the judge. We freeze the manifests behind Table 1 and re-score them with Model 2 and Model 3, changing nothing but the scorer (Table 2). For ERC and GV, all three judges rank ProChunker-BoN first on both (ERC and GV), and the ordering of the seven methods is largely preserved. Absolute scores drop with the weaker judges (Model 2 and Model 3), but the ranking does not change.

ASA behaves differently. ProChunker-BoN finishes second under both independent judges, trailing the first by 0.005 under Model 2 and 0.017 under Model 3, and the method that ranks first changes from judge to judge. We read this as a property of the metric rather than of the chunks. ASA asks the judge to simulate a restricted agent and a full-context agent and then compare their outputs. A weaker judge simulates both agents less faithfully, so the two outputs end up looking alike irrespective of the chunking method used, and the metric loses its ability to separate methods: the spread across the seven methods narrows from 0.160 under Model 1 to 0.132 and 0.065 under Models 2 and 3. We therefore claim no cross-family ASA advantage, and recommend frontier-class judges when this metric is used.

Swapping the judge inside the loop. In this section we analyze the method by changing the scorer, the search (BoN) optimizes against. We re-run the full Best-of-N search with Model 3 scoring every candidate, keeping Model 1 as generator and optimizer (Table 3). If the search were only learning what Model 1 rewards, scoring with Model 3 should flatten the gains. Instead the composite rises by $+0.082$ on SOP- Bench, on 13 of 14 documents, and by $+0.056$ on Industrial SOPs, on 4 of 6. Model 3 scores both the root and the best node,

so the entire gain is measured by the independent evaluator and never by the model that produce d the chunks.

Table 4: Agent Constructability Rate (ACR) on six Industrial SOPs. Per-SOP Raw ACR is normalized to 1.0; Procedural Chunking shows the ratio relative to Raw. Average is step-count-weighted.

SOP	Raw (norm.)	Proc. Chunk	Δ
SOP 1	1.000	1.284	+0.284
SOP 2	1.000	1.084	+0.084
SOP 3	1.000	1.073	+0.073
SOP 4	1.000	1.200	+0.200
SOP 5	1.000	1.250	+0.250
SOP 6	1.000	1.249	+0.249
Average	1.000	**1.170**	**+0.170**

Table 5: RAG on doc2dial (domains: *dmv*, *ssa*, *studentaid*, *va*) under Eval Queries (original turns) and LLM based Queries. Per-column best **bold**, second underlined.

Eval Queries				
Method	**BLEU**	**ROUGE-L**	**METEOR**	**Token F1**
LangChain	0.0441	0.1715	0.2921	0.2133
LlamaIndex	0.0415	0.1650	0.2863	0.2083
LumberChunker	0.0433	0.1745	0.2886	0.2146
HiChunk	0.0406	0.1669	0.2781	0.2076
QChunker	0.0446	0.1733	0.2972	0.2166
ProChunker-Distill	**0.0464**	**0.1799**	**0.3007**	**0.2208**
LLM-generated Queries				
LangChain	0.2424	0.5240	0.6693	0.5292
LlamaIndex	0.2379	0.5124	0.6602	0.5231
LumberChunker	**0.2518**	0.5300	0.6637	0.5321
HiChunk	0.2198	0.4817	0.6254	0.4997
QChunker	0.2499	**0.5359**	**0.6773**	**0.5371**
ProChunker-Distill	0.2498	0.5286	0.6650	0.5335

4.3 Downstream Validation

We validate on agent construction (industrial SOPs) and RAG (doc2dial).

Agent Constructability. This metric evaluates chunking quality through agent construction. Well-defined chunks with clear boundaries and explicit I/O should enable seamless workflow generation, while poorly structured chunks cause mapping failures and ambiguous interfaces. A Strands (Amazon Web Services, 2025) meta-agent consumes each SOP and attempts to build an executable agent by (i) parsing into structured steps, (ii) converting to agent-executable format (structural completeness, S_1), (iii) mapping to automation capabilities (mappability, S_2), (iv) generating typed tool implementations (interface clarity, S_3), (v) executing a mock runner, and (vi) assembling a scorecard. Each dimension is scored GREEN/YELLOW/RED, and we aggregate these into an Agent Constructability Rate (ACR) defined as ACR $= (|\text{GREEN}|\cdot 1.0 + |\text{YELLOW}|\cdot 0.5)/N$ over N steps (Appendix H). We compare *Raw* (original SOP) and *Procedural Chunking* (our structured chunk manifest).

Retrieval-Augmented QA. Each chunker's output feeds a RAG pipeline on `doc2dial` (Feng et al., 2020): bge-base-en-v1.5 retrieval, top-k=5. Two query sets: *Eval* (original turns) and *LLM-generated* (3/doc). Only Distill is shown since BoN is infeasible at scale and Gen is domain-specific; Distill is the configuration we evaluate for low-cost inference.

Procedural chunking improves ACR on all six Industrial SOPs (Table 4) (+17% step-weighted relative gain); largest gains on SOPs with complex branching (*SOP 1*: +28.4%). On RAG (Table 5), Distill leads every metric on Eval Queries; on LLM-generated Queries it stays within 0.01 of QChunker on BLEU / Token F1 and within 0.01 of LumberChunker on BLEU. The narrow ordering across both query sets is consistent with RAG being a coarser test of chunk quality than Table 1's direct metrics. Scoring rubric and gap analysis in Appendix H.

5 Conclusion

We introduced *procedural chunking* as the missing primitive between long-form SOPs and executable multi-agent systems, realized in **ProChunker**: a three-phase pipeline emitting a validated workflow DAG rather than retrieval text spans, paired with five complementary metrics (BC, ERC, ASA, SC, GV) and a Stochastic Best-of-N optimizer with a persistent rules registry. Across BioProBench (500), Industrial SOPs (6), and SOP-Bench (14), ProChunker improves the composite score over the strongest baseline by +0.036 to +0.111 and ERC by up to +0.17, while generalized prompts retain most of the gain at four LLM calls per document and fine-tuned Qwen3-1.7B preserves competitive quality with ∼27 min lower latency. Procedural chunking also lifts agent constructability by 17% on Industrial SOPs and matches retrieval-tuned baselines on doc2dial RAG, positioning chunking as a first-class representational problem for agentic systems, with cross-document workflows and human-in-the-loop rule curation as natural next steps.

Limitations

While ProChunker delivers consistent gains across three corpora and three operating configurations (BoN, Gen, Distill), a few directions remain open for future work. Our evaluation already spans 520 documents across scientific protocols and cross-domain SOPs, and extending it to even larger industrial corpora would further sharpen statistical estimates beyond the trends we already observe. The ASA and GV metrics use LLM judges, evaluated through independent low-temperature calls to decouple them from the generator, and developing fully model-agnostic judges is an interesting orthogonal direction. On the efficiency side, our generalization step already reduces BoN's 60+ LLM calls to a single pass at inference, and the remaining Pareto gap between Gen and BoN points to additional headroom for smarter search strategies. Finally, ProChunker operates on individual SOPs, matching the standard chunking setting and the structure of our benchmarks; extending it to multi-document workflows with cross-references is a promising direction for future work.

References

Amazon Web Services. 2025. Strands agents sdk: A model-driven approach to building and running ai agents.

Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. 2024. Seven failure points when engineering a retrieval augmented generation system. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, CAIN '24, page 194–199, New York, NY, USA. Association for Computing Machinery.

Harrison Chase. 2022. LangChain.

André V. Duarte, João DS Marques, Miguel Graça, Miguel Freire, Lei Li, and Arlindo L. Oliveira. 2024. LumberChunker: Long-form narrative document segmentation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 6473–6486, Miami, Florida, USA. Association for Computational Linguistics.

Song Feng, Hui Wan, Chulaka Gunasekara, Siva Patel, Sachindra Joshi, and Luis Lastras. 2020. doc2dial: A goal-oriented document-grounded dialogue dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8118–8128, Online. Association for Computational Linguistics.

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

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.

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 *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Jerry Liu. 2022. Llamaindex.

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

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Yuyang Liu, Liuzhenghao Lv, Xiancheng Zhang, Li Yuan, and Yonghong Tian. 2025. Bioprobench: Comprehensive dataset and benchmark in biological protocol understanding and reasoning. *arXiv preprint arXiv:2505.07889*.

Wensheng Lu, Keyu Chen, Ruizhi Qiao, and Xing Sun. 2025. Hichunk: Evaluating and enhancing retrieval-augmented generation with hierarchical chunking. *ArXiv*, abs/2509.11552.

Subhrangshu Nandi, Arghya Datta, Nikhil Vichare, Rohith Nama, Udita Patel, Indranil Bhattacharya, Shivam Asija, Arushi Gupta, Giuseppe Carenini, Jing Xu, Shayan Ray, Huzefa Raja, Aaron Chan, Francesco Carbone, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, Prince Grover, Sreyoshi Bhaduri, and 3 others. 2026. Sop-bench: Complex industrial sops for evaluating llm agents. In *KDD*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

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

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 9 others. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 68(2):121101.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

Jihao Zhao, Zhiyuan Ji, Zhaoxin Fan, Hanyu Wang, Simin Niu, Bo Tang, Feiyu Xiong, and Zhiyu Li. 2025. MoC: Mixtures of text chunking learners for retrieval-augmented generation system. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5172–5189, Vienna, Austria. Association for Computational Linguistics.

Jihao Zhao, Daixuan Li, Pengfei Li, Shuaishuai Zu, Biao Qin, and Hongyan Liu. 2026. Qchunker: Learning question-aware text chunking for domain rag via multi-agent debate. In *Proceedings of the ACM Web Conference 2026*, WWW '26, page 2207–2218, New York, NY, USA. Association for Computing Machinery.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*.

Appendix

A Dataset Construction and Statistics

Our training data originates from the BioProbench dataset, which contains scientific protocols stored as structured JSON records. We first converted these JSON records into procedural text SOPs by extracting the protocol steps, materials, and contextual information into a coherent markdown document for each record. This conversion was necessary because our pipeline operates on unstructured text, not structured JSON.

To generate phase-wise training pairs, we ran five inference jobs using Model 1 with the generalized prompts from the cross-document consolidation. Each job corresponds to one of the five pipeline phases: (1) document → segment detection output, (2) document → process extraction output, (3) document + segments → metadata extraction output, (4) metadata → data dependency resolution output, and (5) document + chunks → process dependency output. For each SOP, this produced five input-output pairs. We preserved the chain-of-thought reasoning (enclosed in `<think>` tags) in all outputs, as this reasoning captures the decision-making process behind each phase. Table 6 summarizes the BioProbench dataset used for training data generation. Table 7 reports the document-level statistics across all 14,675 SOPs.

Table 6: BioProbench dataset overview (14,675 SOPs total).

Property	Value
Total SOPs	14,675
Train split	14,175
Test set (held out)	500

Table 7: SOP document statistics (14,675 documents).

	Mean	Std	Median	Min	Max
Character length	7,104	6,231	5,758	342	127,465
Word count	1,109	975	905	47	21,075
Token count	2,269	1,992	1,790	80	46,810
Process steps	27.4	26.5	22	1	1,070
Sections	8.5	8.3	6	0	107
Max heading depth	2.3	0.6	2	1	6

The median SOP is roughly 900 words (1,790 tokens) with 22 process steps and 6 sections. The distribution is heavily right-skewed (Figure 3): the longest document reaches 127K characters (46K tokens), while the shortest is just 342 characters.

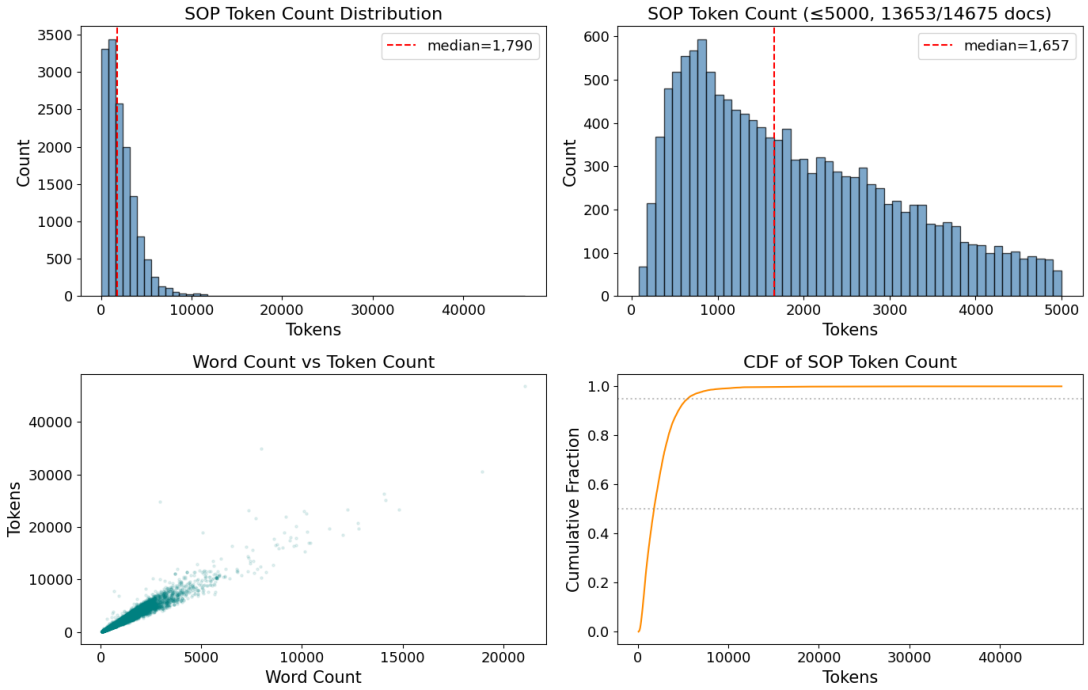


Figure 3: Token length distribution of BioProbench SOPs. The distribution is heavily right-skewed with a median of 1,790 tokens.

B Evaluation Metrics in Detail

This section expands on the metric definitions in the main paper (Section on Evaluation Metrics Φ subsection 3.2) and addresses common implementation questions. Each of the five metrics is computed per-chunk or per-boundary, then aggregated to a document-level score; the composite used by the Best-of-N optimiser is the weighted harmonic mean of the five aggregate scores with equal weights.

B.1 Boundary Clarity (BC)

BC is computed over every adjacent chunk pair in the manifest. For pair (C_i, C_{i+1}), the tail window is the last 2 to 3 sentences of C_i and the head window is the first 2 to 3 sentences of C_{i+1}; window size is clipped to the chunk length for very short chunks. Both windows are embedded with a pretrained embedding model, and the per-pair score is $BC_i = 1 - \cos(\mathbf{e}_{\text{tail}}, \mathbf{e}_{\text{head}})$. The document-level BC is the mean over all adjacent pairs; singleton manifests and the first chunk contribute no pair and are excluded from the mean. Empirically, scores in $[0.5, 0.85]$ correspond to clean transitions, values below 0.4 signal a mid-thought cut that agents will struggle to interpret, and values above 0.85 typically indicate a section-level break (also acceptable). BC is fast and cheap but insensitive to fine-grained coherence beyond surface semantics.

B.2 Entity Reference Coverage (ERC)

ERC measures whether every entity a chunk mentions can be traced to a declared source, which is a prerequisite for an agent to execute the chunk without hallucinating. Entities are extracted by LLM with a prompt that targets domain nouns (actors, artifacts, tools, systems, documents, data items) while excluding generic verbs and function words (full prompt in Appendix I). For each extracted entity, a second LLM call determines whether it matches any item in four source pools: (i) the chunk's own declared outputs, (ii) outputs of all transitively-reachable parent chunks in the dependency graph, (iii) the chunk's declared inputs, and (iv) entity names from referenced definition blocks. Matching is semantic rather than string-exact: synonyms, partial matches, and singular/plural variants count, with the match judgement produced by the LLM rather than a fixed similarity threshold. The per-chunk score is $ERC_i = |\text{covered}|/|\text{total}|$ and the document-level ERC is the mean over all procedural chunks. Reference chunks (definition-only) are excluded because they have no entities to resolve.

Low ERC points at one of two actionable failure modes. If the missing entities are tools or policy artifacts, the Boundary Detection phase failed to create a reference segment for them. If the missing entities are intermediate data items, the Dependency Resolution phase missed a parent edge that

would have exposed the parent chunk's outputs.

B.3 Agent Self-Containedness Agreement (ASA)

ASA operationalises the question "can this chunk be executed given only its declared context?" by running the execution twice and comparing outcomes. Both runs use the same action summary as the task prompt. The *restricted agent* sees only the chunk text, its declared inputs and outputs, and the content of any referenced definition blocks. The *full-context agent* sees the entire source document. Each agent returns a structured action record: the action it took, the output value it produced, the output name, a confidence level, and a free-text report of any missing context. A third LLM (the *agreement judge*) classifies the pair as IDENTICAL (1.0), EQUIVALENT (0.9), PARTIAL (0.5), CONTRADICTORY (0.0), or UNRELATED (0.0). Partial and contradictory cases additionally receive a *failure mode* label drawn from {*reference_gap*, *dependency_gap*, *boundary_error*, *null*}. These failure modes are what the optimiser's feedback router uses to decide which prompt to update: *reference_gap* $\rightarrow \pi_m$, *dependency_gap* $\rightarrow \pi_{dd}, \pi_{pd}$, *boundary_error* $\rightarrow \pi_s$. The document-level ASA is the mean score over all procedural chunks.

ASA is more expensive than the other metrics (three LLM calls per chunk) but catches failure modes no static check can: implicit pronoun antecedents resolved elsewhere in the document, unstated preconditions the full-context agent infers from surrounding text, and hidden domain conventions that a restricted agent cannot know.

ASA and the self-containedness assumption. Our problem definition assumes the source document is self-contained, i.e. all information required to execute the workflow is present somewhere in the text. Because ASA also measures self-containedness, it is worth stating precisely what it does and does not conflate. ASA is a *relative* measurement: it compares what a chunk-restricted agent can do against what a full-document agent can do, so it quantifies whether the chunking *routed* the document's information to the point of use — not whether the document itself is complete. When a required piece of information is genuinely absent from the document, both agents lack it equally, produce similarly incomplete action records, and the agreement judge scores them as agreeing; neither agent can outperform the other, and the shortfall surfaces in both agents' free-text missing-context reports rather than in the ASA score. Consequently a low ASA has exactly one interpretation: the chunk boundary severed information that *was* available in the document but was not routed to the chunk through `depends_on` or `ref_pointers`. ASA therefore does not reward a document that happens to contain everything, nor penalise a chunking for information the document never had; documents with genuine external dependencies are out of scope for the present formulation and are discussed in the Limitations.

B.4 Step Coverage (SC)

Process Extraction: f_{pe} identifies an ordered set of executable steps $\mathcal{W} = \langle w_1, \ldots, w_q \rangle$ from the full document. A passage qualifies if it contains a recognized actor, a recognized verb, and a non-empty result. The practical difficulty is that SOPs entangle actionable content with definitions and policy rationale, and consecutive executable passages do not always warrant separate steps. We addressed both through a multi-layered prompting strategy with explicit extraction rules and exclusion criteria. The steps $\mathcal{W}$ serve as ground truth for Step Coverage.

Step Coverage: SC measures alignment between the two views of the document's procedural content: the ordered step list $\mathcal{W}$ produced by the process-extraction phase f_{pe}, and the chunk set $\mathcal{V}$ produced by *ProChunker*. The two views are extracted independently and must agree about *what* the document actually does, even if they carve the content differently. Mapping is done by a single LLM call that reads the step list and the chunk action-summaries and returns a bipartite assignment: each step is mapped to zero or one containing chunks, and each chunk is mapped to zero or more covered steps. From this assignment we compute step recall $R = |\text{mapped steps}|/|\mathcal{W}|$ and chunk precision $P = |\text{mapped chunks}|/|\mathcal{V}|$, and SC is the harmonic mean $2PR/(P+R)$. Low recall means under-chunking: consecutive executable actions were collapsed into one chunk and some got lost. Low precision means over-chunking: the segmentation produced chunks that carry no executable content.

B.5 Graph Validity (GV)

GV verifies that the final dependency graph $\mathcal{G}$ faithfully encodes the process described in the source

document. Direct validation of the full graph in one LLM call is unreliable once the graph exceeds around fifteen nodes: the prompt gets long, edges are lost in the reasoning, and the judge hallucinates structure that was never there. We therefore decompose $\mathcal{G}$ into boundary-safe subgraphs using Algorithm 3, cap the subgraph count at $K = 5$, and validate each independently.

Each subgraph validation produces four signals. *Edges correct* is the fraction of edges inside the subgraph that the LLM finds supported by the document. *Missing edges* is a list of dependencies the document describes but that the graph omits. *Spurious edges* is a list of edges the graph invents without textual support. *Gateway issues* cover broken branching, mainly mismatched gateway types (XOR/AND/OR) or missing conditional labels. The per-subgraph `is_valid` is true iff there are zero missing edges, zero spurious edges, and zero gateway issues; the structural sub-score is 0.5 if the subgraph is structurally complete (all required convergence/divergence nodes present) and 0 otherwise. The document-level GV is the mean of per-subgraph validity scores. Cross-subgraph dependencies are not re-validated because the cut-point construction guarantees each subgraph is already dependency-complete.

B.6 Composite Score

The composite score s combines the five aggregate metrics via a weighted harmonic mean with equal weights w_i:

$$s = \frac{\sum_i w_i}{\sum_i \frac{w_i}{s_i}}. \tag{1}$$

Harmonic mean is chosen over arithmetic mean because a near-zero score on any one metric pulls s toward zero, which prevents the optimiser from trading one dimension for another (for example, boosting BC by splitting aggressively at the cost of ASA and GV). The convergence threshold $\theta = 0.85$ is applied to the minimum of the five aggregate scores rather than to the composite, so the Best-of-N search only terminates early when every metric is independently strong.

C Algorithm Design Choices

C.1 Stochastic Best-of-N: Full Procedure

This appendix expands the methodology description in Section 3.3 with the complete search procedure, the feedback routing table, and the registry operations.

C.1.1 Rules Registry Operations

The registry $\mathcal{R}$ is a persistent store of refinement rules with four operations invoked in order whenever new rules arrive from feedback:

- **Normalization.** Document-specific identifiers are stripped via regex. Patterns like `C003`, `S005`, `R002` map to placeholder tokens (`[chunk]`, `[segment]`, `[ref]`), and constructions like `'X' (C003)` collapse to `the 'X' step`. Normalization makes rules portable for cross-document generalization.
- **Semantic deduplication.** We embed each rule and compute similarity with other active rules in the same phase, dropping any pair with cosine similarity above 0.85.
- **Contradiction resolution.** For rules sharing phase and category, an LLM judge labels the pair as *contradictory*, *compatible*, *duplicate*, or *refinement*. On contradiction, a composite tie-breaker decides which rule stays active. On refinement, the two rules are merged into the judge's proposed unified form.
- **Consolidation.** When the active rule count in the registry for any phase exceeds a threshold (we use 15), the registry asks the LLM to compress the rule set into a minimal clear list.

C.1.2 Full Search Algorithm

Algorithm 1 gives the complete procedure. The notation matches Section 3.3: π is a prompt tuple, Φ the evaluator, Ψ the feedback generator, $\mathcal{R}$ the registry, θ the convergence threshold, ϵ the acceptance tolerance, B the branching factor, P the patience, and $D_{\max}$ the depth cap. We use $\mathcal{O}_\tau(\pi, f, \mathcal{R})$ for the optimizer LLM that rewrites prompt π given feedback f and registry context, sampled at temperature τ.

C.1.3 Complexity and Cost

Each depth performs B full pipeline evaluations plus $4B$ optimizer calls (four prompts per candidate). The pipeline itself makes five LLM calls (Phases 1 through 3 plus process extraction plus process dependency) and five metric calls, with within-phase and within-metric parallelism. A depth therefore costs $\mathcal{O}(B)$ pipeline evaluations and $\mathcal{O}(B)$ optimizer calls; the accept-reject gate adds no inference. Compared to the linear baseline that accepts every optimizer output unconditionally and costs $3N + 12$ LLM calls per iteration with N chunking evaluations, Best-of-N trades roughly $B\times$ more per-depth LLM calls for the ability to

Table 8: Fixed category-to-phase feedback routing. Rows are feedback fields produced by the feedback LLM, columns are the four optimizable prompts. ASA failures are split by `failure_mode` so per-chunk signals drive updates in the correct phase. The summary refinement rules go to every phase, but each optimizer's system prompt restricts it to rules within its scope.

Feedback field	π_s	π_m	π_{dd}	π_{pd}
`overall_diagnosis`	✓	✓	✓	✓
`under_chunking_issues`	✓			
`over_chunking_issues`	✓			
`boundary_issues`	✓			
`step_coverage_issues`	✓			
`step_coverage_analysis`	✓			
`entity_coverage_issues`		✓		
ASA `reference_gap` chunks		✓		
ASA `boundary_error` chunks		✓		
`graph_validity_issues`			✓	✓
`graph_validity_analysis`			✓	✓
ASA `dependency_gap` chunks			✓	
`summary_rules_for_refinement`	✓	✓	✓	✓

detect and reject regressions and to accumulate rule-level impact estimates in $\mathcal{R}$.

C.2 Dual-Source Graph Merge

We extract two separate edge sets for the workflow graph: data dependencies (from entity input/output matching) and process dependencies (from the document's explicit workflow ordering). The question is how to combine them. A naive union produces redundant edges because the data extractor often infers shortcut edges that the process extractor already covers through intermediate steps. But we cannot simply apply standard transitive reduction to the merged graph either, because that risks deleting process edges that represent real, documented workflow steps. The algorithm needs to clean up data-flow shortcuts without ever touching edges that the document explicitly describes. Algorithm 2 gives the full procedure.

Transitive reduction (lines 8-15) only touches data-only edges. Process edges are never removed because they represent explicit workflow steps. If data dependency gives $A \to C$ but process dependencies provide $A \to B \to C$, the data edge is dropped as redundant while both process edges stay.

To make the issue concrete, consider a real pattern we saw repeatedly in our SOPs. Suppose the document describes a four-step workflow for order validation:

```
C1: Retrieve order details  (outputs: order_record)
C2: Validate eligibility    (inputs: order_record,
                             outputs: eligibility_status)
C3: Check pricing rules     (inputs: order_record,
                             outputs: pricing_ok)
C4: Generate approval report(inputs:
                      eligibility_status, pricing_ok)
```

The process dependency extractor reads the document and correctly identifies the sequential flow: $C1 \to C2 \to C3 \to C4$. The data dependency extractor, working from entity matching, identifies: $C1 \to C2$ (via `order_record`), $C1 \to C3$ (via `order_record`), $C2 \to C4$ (via `eligibility_status`), and $C3 \to C4$ (via `pricing_ok`). Notice the data extractor also produces $C1 \to C3$, which is a valid data-flow edge but redundant given the process chain $C1 \to C2 \to C3$.

Now consider what happens with different reduction strategies:

No reduction (naive union): The merged graph has 6 edges. The extra edge $C1 \to C3$ clutters the graph and confuses the planner, which may try to execute $C3$ immediately after $C1$ without waiting for $C2$.

Global transitive reduction: Standard transitive reduction would examine every edge and remove it if an alternative path exists. This could remove the process edge $C2 \to C3$ because the path $C2 \to C4 \leftarrow C3$ exists indirectly, or it could remove $C1 \to C2$ if it finds $C1 \to C3 \to \ldots$ as an alternative. Either removal would destroy a real workflow step that the document explicitly describes.

Our selective reduction: We only consider edges that appear in the data set but not in the process set. In this example, $C1 \to C3$ is data-only (the process extractor gave us $C1 \to C2 \to C3$ instead). We temporarily remove $C1 \to C3$ and check: is $C3$ still reachable from $C1$? Yes, via $C1 \to C2 \to C3$. So we discard it. All four process edges ($C1 \to C2$, $C2 \to C3$, $C2 \to C4$, $C3 \to C4$) are untouched because they are in the process set. The result is a clean 5-edge graph that preserves every documented workflow step.

Algorithm 1 Stochastic Best-of-N Trajectory Search

Require: Document $\mathcal{D}$, initial policy $\pi^{(0)}$, evaluator Φ, feedback gen Ψ, registry $\mathcal{R}$
Require: Thresholds θ, ϵ; branching B; patience P; depth cap $D_{\max}$; temperatures $\mathcal{T} = \{\tau_1, \ldots, \tau_B\}$
Ensure: Best policy π^* with manifest $\mathcal{G}^*$ and composite score s^*

$\mathcal{G}^{(0)} \leftarrow \mathcal{F}(\mathcal{D}, \pi^{(0)})$; $\mathbf{s}^{(0)} \leftarrow \Phi(\mathcal{G}^{(0)})$; $s^{(0)} \leftarrow$ Composite$(\mathbf{s}^{(0)})$
if $\min_i s_i^{(0)} \geq \theta$ **then**
  return $(\pi^{(0)}, \mathcal{G}^{(0)}, s^{(0)})$ {Root already converged}
end if
$(\pi^*, \mathcal{G}^*, s^*) \leftarrow (\pi^{(0)}, \mathcal{G}^{(0)}, s^{(0)})$; $\rho \leftarrow 0$ {stagnation counter}
for $d = 1$ **to** $D_{\max}$ **do**
  $f \leftarrow \Psi(\mathcal{G}^*, \mathbf{s}^*)$ {diagnosis, issues, rules}
  $\{f_s, f_m, f_{dd}, f_{pd}\} \leftarrow$ PARTITION(f) {Table 8}
  $\mathcal{R} \leftarrow$ REGISTRYADMIT$(\mathcal{R}, f)$ {normalize, dedup, resolve, consolidate}
  $\mathcal{C} \leftarrow \emptyset$
  for $k = 1$ **to** B **do**
    $\pi_s^{(k)} \leftarrow \mathcal{O}_{\tau_k}(\pi_s^*, f_s, \mathcal{R})$; $\pi_m^{(k)} \leftarrow \mathcal{O}_{\tau_k}(\pi_m^*, f_m, \mathcal{R})$ {parallel}
    $\pi_{dd}^{(k)} \leftarrow \mathcal{O}_{\tau_k}(\pi_{dd}^*, f_{dd}, \mathcal{R})$; $\pi_{pd}^{(k)} \leftarrow \mathcal{O}_{\tau_k}(\pi_{pd}^*, f_{pd}, \mathcal{R})$
    $\pi^{(k)} \leftarrow (\pi_s^{(k)}, \pi_m^{(k)}, \pi_{dd}^{(k)}, \pi_{pd}^{(k)})$
    $\mathcal{G}^{(k)} \leftarrow \mathcal{F}(\mathcal{D}, \pi^{(k)})$; $\mathbf{s}^{(k)} \leftarrow \Phi(\mathcal{G}^{(k)})$; $s^{(k)} \leftarrow$ Composite$(\mathbf{s}^{(k)})$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{(\pi^{(k)}, \mathcal{G}^{(k)}, \mathbf{s}^{(k)}, s^{(k)})\}$
  end for
  $k^\dagger \leftarrow \arg\max_k s^{(k)}$; $\Delta \leftarrow s^{(k^\dagger)} - s^*$
  if $\Delta > \epsilon$ **then**
    $(\pi^*, \mathcal{G}^*, \mathbf{s}^*, s^*) \leftarrow$ candidate $k^\dagger$; $\rho \leftarrow 0$
  else if $\Delta > -\epsilon$ **then**
    $(\pi^*, \mathcal{G}^*, \mathbf{s}^*, s^*) \leftarrow$ candidate $k^\dagger$; $\rho \leftarrow \rho + 1$ {within-tolerance}
  else
    $\rho \leftarrow \rho + 1$ {reject all, keep current best}
  end if
  UPDATERULEIMPACTS$(\mathcal{R}, \Delta)$ {write Δ back to applied rules}
  if $\min_i s_i^* \geq \theta$ **then**
    return $(\pi^*, \mathcal{G}^*, s^*)$ {converged}
  end if
  if $\rho \geq P$ **then**
    return $(\pi^*, \mathcal{G}^*, s^*)$ {stagnation}
  end if
end for
return $(\pi^*, \mathcal{G}^*, s^*)$ {depth cap reached}

The cycle-breaking step handles a different failure mode. Cycles arise when the LLM incorrectly identifies a bidirectional dependency, for example claiming both that $C2$ needs output from $C4$ and that $C4$ needs output from $C2$. We detect these via DFS and remove the back-edge (the one that closes the cycle in traversal order), which in practice corresponds to the less plausible dependency direction. If the graph is still cyclic after back-edge removal, we fall back to enforcing topological order based on chunk ID numbering, which serves as a proxy for document position.

C.3 Cut-Point Subgraph Sampling

Validating an entire dependency graph in a single LLM call is impractical for two reasons: the prompt would be too long, and the LLM tends to lose accuracy when reasoning about large graph structures. We therefore need to sample subgraphs, but the sampling must be done carefully. A randomly sampled subgraph might cut through the middle of a dependency chain, making it impossible for the LLM to judge whether the edges are correct without seeing the full chain. Our solution uses cut-point analysis. A node S is a perfect cut point when no ancestor of S has a direct edge to any descendant of S, meaning all paths from the "before S" portion of the graph to the "after S" portion must pass through S. Splitting the graph at a perfect cut point produces two subgraphs that are dependency-complete: every edge within each subgraph can be validated without needing information from the other subgraph.

Algorithm 2 Procedural Graph Extraction with Dual-Source Merge

Require: Document $\mathcal{D}$, Prompts $\pi = \{\pi_s, \pi_m, \pi_{dd}, \pi_p, \pi_{pd}\}$
Ensure: Executable workflow graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$

$S \leftarrow f_{\text{seg}}(\mathcal{D}, \pi_s)$; $\mathcal{W} \leftarrow f_{\text{pe}}(\mathcal{D}, \pi_p)$ {Parallel}
$M \leftarrow f_{\text{meta}}(S, \mathcal{D}, \pi_m)$ {Sequential}
$E_d \leftarrow f_{\text{data}}(M, \pi_{dd})$; $E_p \leftarrow f_{\text{proc}}(\mathcal{D}, M, \pi_{pd})$ {Parallel}
$G \leftarrow (\mathcal{V},\ E_d \cup E_p)$
if G contains cycles **then**
  $G \leftarrow$ BREAKCYCLES(G)
end if
$E_{\text{data-only}} \leftarrow E_d \setminus E_p$
for each $(u, v) \in E_{\text{data-only}}$ **do**
  Remove (u, v) from G
  if v is reachable from u in G **then**
    keep removed {Transitively implied}
  else
    Restore (u, v) to G {Essential edge}
  end if
end for
Attach text spans; validate cross-references
return $\mathcal{G}^* = (\mathcal{V}^*, \text{edges}(G))$

The greedy selection strategy picks cut points that maximize new node coverage, ensuring we do not waste our subgraph budget on overlapping regions. The tail subgraph handles nodes not covered by any prefix subgraph. For disconnected graphs (multiple weakly connected components), we skip cut-point analysis entirely and treat each component as a natural subgraph, since disconnected components are already boundary-safe by definition. We chose $K = 5$ as the maximum number of subgraphs based on empirical observation: most SOPs produce graphs with 8 to 25 chunks, and 5 subgraphs typically cover 80%+ of the nodes while keeping the validation prompt within token limits. Figure 4 illustrates the full algorithm procedure.

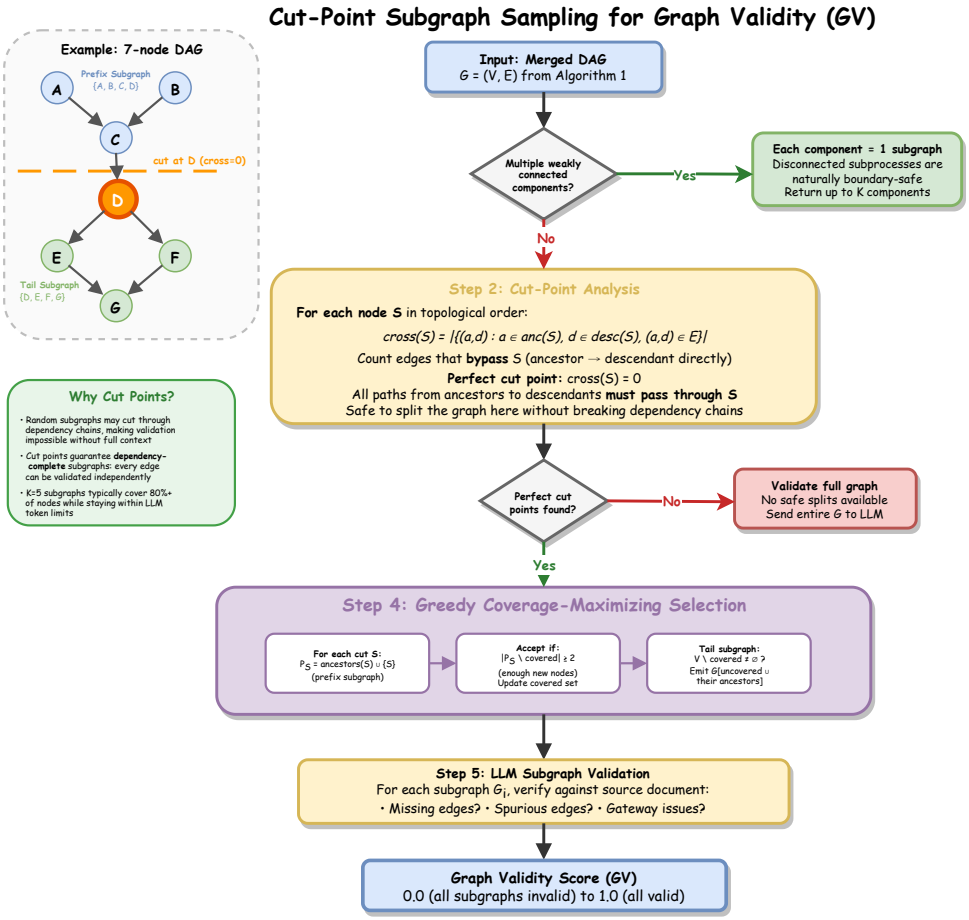


Figure 4: Cut-point subgraph sampling for Graph Validity: the DAG is split at nodes with zero cross-edges into boundary-safe subgraphs for independent LLM validation.

D Experimental Setup

Datasets. BioProBench is a collection of scientific protocols from Bio-protocol, Protocol Exchange, and Protocol.io, covering molecular biology, cell culture, biochemistry, and computational biology. Document lengths range from single-page protocols to 30+ step procedures. We use 14,175 documents for training data generation and hold out 500 documents as the test set. **Industrial SOPs** is an internal collection of operational SOPs and operational workflows, used for per-document Best-of-N evaluation.

Pipeline configuration. All three pipeline phases, process extraction, and process dependency run on Model 1. BC uses a pretrained sentence-embedding model with 2-3 sentence tail/-head windows. The metric LLMs (ASA restricted, ASA full-context, agreement judge, ERC extraction, ERC matching, GV validation) all use Model 1 at low temperature. The Stochastic Best-of-N search uses branching B=3 at temperatures $\{0.5, 0.7, 0.9\}$, patience P=2, depth cap $D_{\max}$=3, convergence threshold θ=0.85, and acceptance tolerance ϵ=0.005. The composite score uses equal metric weights (0.20). The student is Qwen3 1.7B, fine-tuned on phase-wise input–output pairs produced by Model 1 under the generalized prompts; finetuning details are in Appendix G.

Downstream RAG setup. We use the IBM `doc2dial` dataset, a conversational document grounding benchmark across four public-service domains (*dmv*, *ssa*, *studentaid*, *va*). Underlying documents are treated as SOPs and fed to the chunker. Retrieval uses `BAAI/bge-base-en-v1.5` sentence embeddings with top-k=5 chunks per query, and answer generation uses Model 4. Two query sets are evaluated: *Eval Queries* are the original doc2dial conversational turns, and *LLM Queries* are three questions per document generated by Model 1 from the document text. Answer quality is measured against reference answers via BLEU, ROUGE-L, METEOR, and token F1.

Extended result discussion. On the three metrics every method supports (BC, ERC, ASA), our pipeline's advantage is concentrated in ERC, where the workflow-aware extraction keeps every

Algorithm 3 Cut-Point Subgraph Sampling for Graph Validation

Require: DAG $G = (\mathcal{V}, \mathcal{E})$, Max subgraphs K
Ensure: Boundary-safe subgraphs $\{G_1, \ldots, G_m\}$, $m \leq K$
if G has multiple weakly connected components **then**
  return each component as a subgraph (up to K)
end if
for each node S in topological order **do**
  $\text{cross}(S) \leftarrow |\{(a, d) : a \in \text{anc}(S), \; d \in \text{desc}(S), \; (a, d) \in \mathcal{E}\}|$
end for
cuts $\leftarrow \{S : \text{cross}(S) = 0\}$
if cuts $= \emptyset$ **then**
  return $\{G\}$
end if
covered $\leftarrow \emptyset$
for each $S \in$ cuts **do**
  $P_S \leftarrow \text{ancestors}(S) \cup \{S\}$
  if $|P_S \setminus \text{covered}| \geq 2$ **then**
    Emit $G[P_S]$; covered $\leftarrow$ covered $\cup P_S$
  end if
end for
if $\mathcal{V} \setminus$ covered $\neq \emptyset$ **then**
  Emit tail subgraph over uncovered nodes
end if

entity reference tied to a declared source (own output, parent output, declared input, or referenced definition). BC is mid-range because our pipeline produces longer chunks than size-based splitters, which makes semantic transitions across boundaries smoother rather than sharper. ASA is strongest on HiChunk for BioProBench (0.872) and QChunker for Industrial SOPs (0.817), which both use LLM-driven boundary logic similar to ours; our generalized prompts trade a small ASA margin for large SC and GV gains. On the RAG benchmark, the narrow gaps between chunkers are consistent with observations that retrieval quality saturates quickly with reasonable chunking, and that metric-level differences visible in direct chunking evaluation compress when queries are short and answerable from a single retrieved passage. The gap between the Eval Queries and LLM-generated Queries absolute scores comes from the query distribution: LLM-generated questions target document-level facts rather than local turn-level clarifications, which rewards retrieval precision and is where our pipeline's more coherent chunks matter most.

E Baseline Adaptations

All baselines are evaluated under a shared post-processing step: each baseline emits chunk text only, then a common LLM enrichment pass supplies everything the five metrics require but the baselines do not natively produce — per-chunk workflow metadata (actor, action summary, inputs, outputs, dependencies, reference pointers), the process-step list needed for Step Coverage, and the dependency graph needed for Graph Validity. This decouples chunk boundaries, which the baselines control, from metadata and graph construction, which is held constant across every method so that all five metrics are computed identically for baselines and for our pipeline. The enrichment is a single prompt that receives the raw chunk boundaries plus the full source document; it is the same prompt for every baseline. The adaptations for each method are summarized below.

- **LangChain** (Chase, 2022) `RecursiveCharacterTextSplitter`. Used directly with chunk size 1500 characters and 150-character overlap, separator hierarchy `["\n\n", "\n", " ", ""]`. No modifications.
- **LlamaIndex** (Liu, 2022) semantic chunker. Reimplemented `SemanticSplitterNodeParser` on `BAAI/bge-small-en-v1.5`, buffer size 1, breakpoint at the 95th percentile of cosine distance between adjacent sentence groups.
- **LumberChunker** (Duarte et al., 2024). Paragraph-grouped token-threshold cycling with θ=550 tokens; the original backbone replaced with Model 1 because the paper's finetuned backbone is not public.
- **QChunker** (Zhao et al., 2026). Four-stage question-driven pipeline (outline, segment, judge, rewrite); each stage replaced with an LLM prompt because qchunker-3B, Judge LoRA, and Rewrite weights are not released. `[MASK]` boundary recovery with Levenshtein fallback preserved.
- **MoC** (Zhao et al., 2025) (Mixture of Chunkers). Router-expert pipeline collapsed into a single LLM call with a granularity-aware prompt because the router and four expert models are not released. Paper's `[MASK]` recovery preserved.
- **HiChunk** (Lu et al., 2025). Sliding-window hierarchical segmentation with the paper's verbatim English prompt; Qwen3-4B backbone replaced with Model 1 because the finetuned weights are not released.

None of the baselines natively produce the process-step list required for Step Coverage or the dependency graph required for Graph Validity. The shared enrichment pass described above supplies both, which is what makes all five metrics — including SC and GV — well defined for every base-

line entry in Table 1. Because this step could in principle flatter or handicap either side, we quantify its effect directly in Appendix F.5: applying the same generic enrichment to our own chunk boundaries *lowers* GV below the baselines, confirming the shared harness does not favour our pipeline.

F Supplementary Experimental Results

F.1 Per-SOP Comparison on Industrial SOPs

Table 9 gives the per-document breakdown of the Industrial SOPs results from Table 1: ProChunker-BoN, ProChunker-Gen, and ProChunker-Distill.

ProChunker-BoN leads the Industrial SOPs with the highest composite delta ($+0.269$), driven by the strongest GV ($+0.396$) and ERC ($+0.311$) lifts: its acceptance gate rejects candidates that regress these metrics, which the generalized and fine-tuned variants cannot replicate at inference time. ProChunker-Gen trails at $+0.165$ composite delta; the gap concentrates in GV ($+0.166$, dragged down by SOP-4 at -0.150) because static rules cannot adapt to document-specific dependency patterns. ProChunker-Distill recovers to $+0.220$ despite using a single 1.7B-parameter forward pass per phase; it matches ProChunker-Gen on ASA ($+0.313$ vs $+0.291$) and exceeds it on GV ($+0.329$ vs $+0.166$), showing that dependency reasoning transfers well through fine-tuning.

F.2 Root vs ProChunker-Gen vs ProChunker-BoN on Held-Out Documents

Table 10 reports a three-way comparison on 10 held-out BioProBench documents: the base prompt policy (Original), ProChunker-Gen (single pass, no per-document optimization), and ProChunker-BoN.

ProChunker-Gen improve 8 of the 10 documents (mean $+0.045$), with the strongest per-metric lifts on ERC ($+0.107$) and GV ($+0.100$). The two regressions (Protocol.io-7823 and Protocol.io-4718) involve formatting patterns the universal rules did not capture. ProChunker-BoN adds a further $+0.090$ on top of generalization, with the gap largest on structurally atypical documents (Protocol.io-4718: $+0.321$) and negligible on well-organized ones (Protocol-exchange-200: $+0.001$). This pattern characterises the cost–quality trade-off between the two: ProChunker-Gen is sufficient for well-formed documents, while ProChunker-BoN's additional cost is repaid only on the long tail where structure deviates from the training distribution.

F.3 Cross-Family Optimization: Per-Document Trace

Table 11 expands the SOP-Bench row of Table 3. The full Best-of-N search is re-run with Model 3 ($\sim$100B+, an independent model family) scoring every candidate; the generator and the prompt optimizer remain Model 1. Both the root composite and the best composite are computed by Model 3, so the reported improvement is measured end-to-end by the foreign evaluator and never by the model that produced the chunks.

On the six Industrial SOPs the same procedure yields a mean composite improvement of $+0.056$ under Model 3, with four of the six documents improving; the two that do not both terminate on stagnation from root scores already at the top of that corpus's range. Per-document absolute scores are withheld for confidentiality, so only the aggregate improvement is reported.

F.4 Statistical Significance

Table 12 reports paired Wilcoxon signed-rank tests on the per-document composite scores of the 14 SOP-Bench documents. The test is paired because every method sees the same document set, and non-parametric because per-document composites are neither normally distributed nor independent of document difficulty. Tests are one-sided: our hypothesis is directional (ProChunker improves over a given baseline), so the alternative is that the per-document differences are positive.

The $+0.036$ composite gain on BioProBench deserves separate comment, because it is the smallest headline number in the paper and the reported standard deviations (± 0.011–0.014) are of a similar magnitude. Those deviations are the variance of the corpus-level mean across five independent runs, not the standard error of the difference. With 500 paired documents the standard error of the mean difference is approximately $0.014/\sqrt{500} \approx 0.0006$, two orders of magnitude below the effect, so the gain is well outside run-to-run noise despite appearing marginal next to the per-run spread.

F.5 Baseline Fairness: Enriched BoN

Every baseline receives the shared LLM enrichment described in Appendix E, which supplies the metadata and dependency graph the baselines do not produce natively. Because that step is itself

Table 9: Per-SOP comparison on Industrial SOPs (Δ relative to a fixed baseline constant): ProChunker-BoN, ProChunker-Gen, and ProChunker-Distill. Absolute scores withheld for confidentiality.

	ProChunker-BoN						ProChunker-Gen						ProChunker-Distill					
SOP	BC	ERC	ASA	SC	GV	Comp.	BC	ERC	ASA	SC	GV	Comp.	BC	ERC	ASA	SC	GV	Comp.
SOP-1	+0.100	+0.421	+0.327	+0.409	+0.500	+0.325	+0.084	+0.228	+0.310	+0.100	+0.214	+0.095	+0.095	+0.147	+0.400	+0.214	+0.500	+0.242
SOP-2	−0.043	+0.274	+0.316	+0.500	+0.350	+0.227	−0.043	+0.406	+0.337	+0.500	+0.350	+0.250	+0.035	+0.144	+0.367	+0.333	+0.300	+0.212
SOP-3	+0.085	+0.375	+0.238	+0.386	+0.400	+0.276	+0.085	+0.291	+0.275	+0.320	+0.100	+0.199	+0.170	+0.250	+0.350	+0.318	+0.000	+0.192
SOP-4	+0.363	+0.289	+0.288	+0.200	+0.375	+0.298	+0.114	+0.255	+0.260	+0.322	−0.150	+0.099	+0.093	+0.157	+0.157	+0.220	+0.375	+0.188
SOP-5	+0.096	+0.199	+0.373	+0.338	+0.500	+0.276	+0.112	+0.240	+0.311	+0.371	+0.285	+0.134	+0.134	+0.055	+0.267	+0.500	+0.500	+0.248
SOP-6	+0.070	+0.311	+0.235	+0.257	+0.250	+0.214	+0.079	+0.229	+0.253	+0.368	+0.200	+0.214	+0.068	+0.130	+0.338	+0.500	+0.300	+0.236
Average	+0.112	+0.311	+0.296	+0.348	+0.396	**+0.269**	+0.072	+0.275	+0.291	+0.330	+0.166	**+0.165**	+0.099	+0.147	+0.313	+0.348	+0.329	**+0.220**

Table 10: Three-way comparison on 10 held-out BioProBench documents. Original: base prompts only. ProChunker-Gen, and ProChunker-BoN. $\Delta(G-O)$ is the lift from generalization; $\Delta(B-G)$ is the additional lift from per-document search on top of generalization.

Document	**Orig.**	**ProChunker-Gen**	**ProChunker-BoN**	**Δ(G−O)**	**Δ(B−G)**
Protocol-exchange-140	0.654	0.824	0.809	+0.170	−0.016
Protocol.io-8088	0.627	0.755	0.811	+0.127	+0.056
Protocol.io-4597	0.657	0.781	0.845	+0.124	+0.065
Protocol-exchange-1046	0.544	0.641	0.850	+0.097	+0.209
Protocol-exchange-1050	0.628	0.719	0.809	+0.091	+0.090
Protocol-exchange-1143	0.777	0.804	0.848	+0.026	+0.044
Protocol-exchange-200	0.847	0.855	0.856	+0.008	+0.001
Protocol.io-5193	0.782	0.785	0.821	+0.003	+0.035
Protocol.io-7823	0.766	0.716	0.815	−0.049	+0.098
Protocol.io-4718	0.720	0.574	0.895	−0.146	+0.321
Mean	**0.700**	**0.745**	**0.836**	**+0.045**	**+0.090**

Table 11: Cross-family optimization on SOP-Bench, per document. Root and best composite are both scored by Model 3. The search improves 13 of 14 documents; the single regression (`referral_abuse_detection_v2`) terminates on stagnation at a root score already above the corpus mean.

Document	**Root**	**Best**	**Δ**
`aircraft_inspection`	0.713	0.761	+0.048
`content_flagging`	0.711	0.832	+0.121
`customer_service`	0.714	0.791	+0.077
`dangerous_goods`	0.752	0.761	+0.009
`email_intent`	0.651	0.861	+0.210
`know_your_business`	0.720	0.825	+0.104
`order_fulfillment`	0.818	0.864	+0.046
`patient_intake`	0.735	0.804	+0.069
`referral_abuse_detection_v1`	0.624	0.779	+0.155
`referral_abuse_detection_v2`	0.785	0.782	−0.003
`traffic_spoofing_detection`	0.784	0.842	+0.057
`video_annotation`	0.718	0.752	+0.034
`video_classification`	0.680	0.768	+0.088
`warehouse_package_inspection`	0.723	0.856	+0.133
Mean	**0.723**	**0.806**	**+0.082**

an LLM call, it could in principle be the source of our advantage rather than the chunk boundaries. We test this directly with an ablation that inverts the usual comparison: instead of giving baselines our pipeline, we give our pipeline the baselines' harness.

Concretely, we take ProChunker-BoN's final chunks and *strip all native metadata* — actor, inputs, outputs, `depends_on`, reference pointers, and the merged graph — leaving only the raw text boundaries. Those boundaries are then re-processed through the identical single-call enrichment prompt the baselines use. The result, *Enriched BoN*, has exactly ProChunker-BoN's boundary positions but baseline-quality metadata.

Table 12: One-sided paired Wilcoxon signed-rank tests on per-document composites (SOP-Bench, n=14). Mean Δ is the average per-document difference. All six comparisons are significant at $p < 0.01$.

Comparison	**Mean Δ**	**p-value**	**Significance**
BoN vs QChunker	+0.159	0.000061	$p < 0.001$
BoN vs MoC	+0.166	0.000122	$p < 0.001$
BoN vs HiChunk	+0.196	0.000061	$p < 0.001$
Gen vs QChunker	+0.079	0.000854	$p < 0.001$
Gen vs MoC	+0.086	0.005371	$p < 0.01$
Gen vs HiChunk	+0.116	0.000854	$p < 0.001$

Table 13: Baseline-fairness ablation on SOP-Bench (14 documents). All three conditions are scored in a single run so the comparison is controlled; values therefore differ slightly from the 5-run means of Table 1.

Condition	**Mean GV**
Native BoN (full three-phase pipeline)	0.917
Enriched BoN (shared baseline enrichment)	0.644
Best baseline, MoC (same enrichment)	0.651

Three conclusions follow. First, the shared enrichment does not favour us: applied to our own boundaries it *lowers* GV to the level of the strongest baseline, so if the harness were biased, the bias runs against ProChunker rather than for it. Second, ProChunker's graph quality comes from the dedicated three-phase architecture — boundary detection, then metadata extraction, then dependency resolution with graph merging — and not from boundary placement alone, since the boundaries are

held fixed here. Third, the single-call enrichment struggles more with finer-grained segmentations: ProChunker produces 15–25 chunks per document against 3–10 for the baselines, so it must infer many more dependency edges from one prompt. The baselines receive the strongest graph this harness can produce precisely because their coarser boundaries leave fewer edges to resolve.

F.6 Single-Call Long-Context Baseline

A natural objection to a multi-phase pipeline is that a sufficiently long context window makes it unnecessary: a frontier model could read the whole SOP and emit the entire manifest in one pass. We test this with a *Single-Call* baseline that receives the full document in a single prompt with a 1M-token context window and is asked for chunks, metadata, and the dependency graph together.

Table 14: Single-call long-context baseline. SOP-Bench values for ProChunker-BoN and QChunker are the 5-run means of Table 1; Single-Call is a single run, as it appears in no other table. Industrial SOP figures are relative to ProChunker-BoN, whose absolute scores are withheld.

Method	ERC	GV	Comp.
SOP-Bench (14 documents)			
ProChunker-BoN	0.962	0.937	0.826
Single-Call	0.814	0.730	0.707
QChunker (best baseline)	0.704	0.694	0.715
Industrial SOPs (6 documents, relative to BoN)			
ProChunker-BoN	1.00×	1.00×	1.00×
Single-Call	0.78×	0.62×	0.80×

The single call trails the pipeline on both corpora, and the shortfall concentrates in GV — the metric that measures whether the dependency graph faithfully encodes the document. On the Industrial SOPs, which are longer and more heavily cross-referenced, single-call GV drops to $0.62\times$ ProChunker-BoN. Dependency resolution appears to be the part of the task that benefits most from being isolated into its own phase with its own prompt: asked to segment, annotate, and wire a graph simultaneously, the model produces plausible chunks but a substantially weaker graph. Context length and model capability alone therefore do not substitute for the multi-phase decomposition.

F.7 Entity Coverage Distribution

ERC is aggregated as a mean over chunks, which raises a fair question: a document whose chunks average well could still contain one chunk with an unresolvable entity reference, and that single chunk is enough to break an agent's execution. We optimise for the mean during search because LLM-based entity extraction occasionally emits false positives, and a strict minimum makes the search gradient hypersensitive to a single mis-extracted generic noun, causing the optimiser to reject otherwise large improvements. For evaluation, however, the distribution matters.

Table 15: ERC distribution on SOP-Bench (14 documents). Min ERC is the lowest per-chunk coverage observed for that method; Gap is Mean − Min; the last column is the fraction of chunks whose entities are fully resolved.

Method	Mean	Min	Gap	% fully cov.
ProChunker-BoN	**0.915**	**0.655**	0.260	**67.6**
ProChunker-Gen	0.848	0.627	**0.222**	42.1
Single-Call	0.814	0.357	0.457	48.4
QChunker	0.767	0.277	0.490	27.0
MoC	0.661	0.203	0.459	24.3

ProChunker-BoN's worst chunk scores 0.655, above the *mean* of two of the baselines and well above every baseline's minimum. The mean-to-minimum gap is also the tightest among the non-ProChunker systems, so the high average reflects uniformly high coverage rather than a few excellent chunks masking broken ones. The clearest separation is in the last column: two thirds of ProChunker-BoN's chunks have every entity reference resolved, against roughly a quarter for the strongest baselines.

G Model Finetuning Details

We fine-tuned Qwen3 1.7B using LoRA (rank 16, $\alpha = 32$, dropout 0) on all attention and MLP projection layers (`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`). Training used the Unsloth framework with the following configuration: learning rate 2×10^{-4}, linear scheduler with 5 warmup steps, AdamW optimizer with weight decay 0.001, batch size 1 per device with gradient accumulation of 2 steps, bf16 precision, max sequence length 128K tokens, and 3 epochs. We used DeepSpeed ZeRO for memory efficiency and Liger kernel for throughput optimization. Training was performed with the `train_on_responses_only` strategy, masking the user turn and training only on the assistant response (including the `<think>` reasoning and the final JSON output).

The instruction format follows the Qwen chat template with phase-specific system prompts. Each training example is a single-turn conversation: the

user message contains the phase instruction plus the input (document text for Phase 1, document + segments for Phase 2, metadata JSON for Phase 3), and the assistant message contains the chain-of-thought reasoning followed by the structured JSON output.

G.1 Ablation: Training Without Thinking

To understand the contribution of chain-of-thought reasoning to the fine-tuned model's performance, we trained a variant where the <think> blocks were stripped from the training targets, leaving only the final JSON output. Table 16 reports the results.

Removing the thinking traces causes a clear drop: average composite delta falls from $+0.220$ (with thinking, Table 9) to $+0.134$, a decrease of 0.086. The largest degradation is on Graph Validity ($+0.329 \rightarrow +0.017$, a drop of 0.312), which tracks the intuition that dependency resolution requires multi-step reasoning about entity flow and workflow ordering; without the explicit chain-of-thought the student tends to emit structurally plausible but semantically inconsistent edges. ASA also drops from $+0.313$ to $+0.192$, indicating that the restricted-agent self-containedness judgments suffer when the model cannot internally reason about which context is actually needed. SC stays high ($+0.439$) because step-to-chunk alignment is a largely surface-level matching task that does not benefit much from chain-of-thought.

G.2 Ablation: Training on Short SOPs Only (Under 5K Tokens)

To test how training data size and complexity affect the fine-tuned model, we trained a separate Qwen3 1.7B model using only SOPs shorter than 5K tokens from the BioProbench dataset. These short SOPs have simpler structure, fewer chunks, and less conditional branching than the full training set. We then evaluated this model on the same internal SOPs used in the main experiments, which are substantially longer and more complex. Table 17 reports the results.

The short-SOP-trained model averages $+0.106$ composite delta, compared to $+0.220$ for the full-data Qwen3 1.7B (Table 9), a drop of 0.114. The most visible symptom is under-chunking: the model produces 3 to 13 chunks per SOP versus the 15 to 25 typical of the full-data model, because it learned from documents with far fewer atomic steps. ASA holds up well ($+0.334$) because fewer chunks tend to be more self-contained by construction, but the smaller chunk count hurts SC ($+0.209$) whenever a long SOP's process steps spill across chunks the short-trained model never learned to produce. ERC ($+0.094$) and the more skewed GV distribution (individual SOPs ranging -0.200 to $+0.450$) confirm that the model did not see enough complex entity reference patterns or deep dependency structures during training to handle them reliably on longer documents.

H Agent Constructability: Pipeline and Scoring Details

Structured chunking transforms unstructured SOP documents into formal workflow specifications that are more amenable to automation. To validate this claim, we choose agent construction as a downstream task. Given an SOP, can a system produce a working automated agent—complete with typed tool signatures, executable mock implementations, and a runnable orchestration script?

Unlike LLM-as-a-judge approaches that rate document quality, our evaluation attempts to *build* the agent, generates executable artifacts, runs them, and reports where construction fails. A Strands meta-agent processes each SOP through a seven-stage pipeline, scoring every extracted step on three dimensions: structural completeness (S_1), capability mappability (S_2), and interface clarity (S_3). Each dimension receives a traffic-light grade (GREEN/YELLOW/RED). The Strands meta-agent processes each SOP through seven stages, producing executable artifacts at each:

1. Parse the SOP into structured steps.
2. Convert to agent-executable format; score structural completeness (S_1).
3. Map each step to an automation capability; score mappability (S_2).
4. Generate typed tool implementations; score interface clarity (S_3).
5. Generate the agent runner script.
6. Execute the mock agent in a subprocess.
7. Assemble a deterministic scorecard (no LLM involvement in final scoring).

The agent constructability rate (ARC) aggregates across all N steps:

$$\text{ACR} = \frac{|\text{GREEN}| \times 1.0 + |\text{YELLOW}| \times 0.5}{N} \quad (2)$$

Table 16: Ablation: Qwen3 1.7B trained without chain-of-thought reasoning (Δ relative to a fixed baseline constant). Performance drops substantially on all metrics, particularly GV.

SOP	Chunks	BC	ERC	ASA	SC	GV	Comp.
SOP-1	24	+0.069	+0.222	+0.204	+0.467	+0.150	+0.201
SOP-2	14	−0.013	+0.166	+0.200	+0.500	−0.150	+0.066
SOP-3	15	+0.105	+0.059	+0.153	+0.477	+0.000	+0.125
SOP-4	18	+0.142	+0.152	+0.189	+0.500	+0.100	+0.194
SOP-6	20	+0.076	+0.086	+0.150	+0.237	−0.250	−0.016
SOP-5	22	+0.098	+0.198	+0.255	+0.450	+0.250	+0.234
Average		+0.080	+0.147	+0.192	+0.439	+0.017	+0.134

Table 17: Ablation: Qwen3 1.7B trained on SOPs under 5K tokens only, evaluated on full-length internal SOPs (Δ relative to a fixed baseline constant).

SOP	Chunks	BC	ERC	ASA	SC	GV	Comp.
SOP-1	7	+0.134	−0.087	+0.400	+0.500	+0.450	+0.198
SOP-2	13	+0.016	+0.166	+0.338	+0.500	−0.100	+0.115
SOP-3	3	+0.050	−0.029	+0.267	−0.227	+0.400	−0.001
SOP-4	6	+0.124	+0.273	+0.333	+0.083	+0.170	+0.185
SOP-6	9	+0.106	+0.138	+0.311	+0.375	−0.200	+0.060
SOP-5	9	+0.039	+0.101	+0.356	+0.022	+0.000	+0.081
Average		+0.078	+0.094	+0.334	+0.209	+0.120	+0.106

H.1 Scoring Rubric

Each dimension (S_1, S_2, S_3) receives a traffic-light grade:

- GREEN: Fully derivable, no assumptions needed.
- YELLOW: Derivable with assumptions; each assumption is logged as a gap question for subject-matter experts.
- RED: Cannot be derived—critical information is missing.

The overall step score is $\min(S_1, S_2, S_3)$.

H.2 Gap Questions

Every YELLOW or RED step must produce at least one gap question—a specific, actionable query that, when answered by a domain expert, resolves the ambiguity.

```
Examples:
    - What is the URL for the system mapping
        data source?
    - What is the content for the notification?
```

These are precise engineering requirements corresponding directly to parameters, tool configurations, or branching conditions needed for agent implementation.

H.3 Raw vs. Procedural Chunking Modes

In raw mode, the meta-agent receives only SOP markdown and must both discover what each step requires and assess whether the SOP describes it. In procedural chunking mode, the chunk manifest declares per-step structural metadata: inputs, outputs, inter-step dependencies, reference pointers to tools and templates, and gateway conditions for branching logic. The manifest performs structural analysis; the meta-agent performs construction. The consistent improvement across all six SOPs (Table 4) confirms that the structured representations produced by our pipeline carry measurable downstream utility, directly reducing the ambiguity an agent-building system must resolve.

H.4 Score Distribution

Table 18: Step-level score distribution across evaluation modes. Avg Rate is the mean ACR; Avg Gaps is the mean number of gap questions per SOP.

Mode	GREEN	YELLOW	RED	Avg Rate (norm.)	Avg Gaps ↓
Raw SOP	32%	58%	10%	1.000	1.62×
Procedural Chunking	42%	57%	<1%	1.170	1×

Table 18 shows the step-level score distribution. The most significant shift is in RED steps: raw SOP mode produces 10% RED steps, while manifest-augmented mode produces <1%. RED steps represent complete blockers where the agent cannot be built due to missing information. The manifest eliminates nearly all such blockers by providing the structural specification that raw SOP text lacks. GREEN steps increase from 32% to 42%, indicating that the manifest's declared inputs, outputs, and dependencies provide sufficient information for the evaluator to generate complete tool signatures without assumptions.

H.5 Gap Analysis

Procedural Chunking reduces gap questions by 1.62× on average: raw SOP mode produces 1.62× as many ambiguity-triggering gaps as the manifest-augmented mode, with reductions consistent across all six SOPs, ranging from 1.11× (SOP-4) to 2.43× (SOP-3) (Table 19). Combined with the

Table 19: Gap questions per SOP. Manifest Gaps is normalized to 1×; Raw Gaps shows the multiplier relative to manifest. Higher multiplier means more ambiguities in raw mode.

SOP	Raw Gaps	Manifest Gaps (norm.)	Reduction
SOP-1	1.66×	1×	40%
SOP-2	1.54×	1×	35%
SOP-3	2.43×	1×	59%
SOP-4	1.11×	1×	10%
SOP-6	1.66×	1×	40%
SOP-5	1.81×	1×	45%
Average	**1.62×**	1×	**38%**

near-elimination of RED blockers and the consistent ACR improvement reported in Table 4, these results confirm that the structured representations produced by our chunking and graph extraction pipeline carry measurable downstream utility, directly reducing ambiguity and accelerating the path from SOP document to executable agent.

H.6 Human Verification of RED Steps

The metrics in this paper are computed by LLMs, and the Agent Constructability Rate is no exception: the meta-agent's GREEN/YELLOW/RED judgements are model-assigned. To check that these judgements track something a person would recognise, we ran a targeted human review of the failure cases.

Protocol. Annotators were domain experts already familiar with the six Industrial SOPs. They independently reviewed *every* RED-flagged step produced in both evaluation modes — Raw SOP and Procedural Chunking — across all six documents. For each flagged step they were shown the step, the reason recorded for the flag (missing dependency, unresolved entity, or unclear gateway condition), and the surrounding document, and asked a single question: is this a genuine ambiguity that a human executor would also have to resolve before proceeding?

Findings. In Procedural Chunking mode, annotators agreed that the RED classifications corresponded to real questions, of the form "which approval threshold applies here?" or "what system receives this output?", that a human executing the step would likewise need answered. In Raw SOP mode, which produces 10% RED steps against under 1% for Procedural Chunking (Table 18), annotators observed that many of the flagged questions could in fact be answered from information present elsewhere in the document; the difficulty was that the construction system could not surface that information without structured chunk boundaries and explicit dependency links. This distinction matters for interpreting the ACR improvement: it suggests our structured representation largely removes ambiguity that arises from how the procedure is *presented*, rather than ambiguity arising from genuinely absent information.

Scope and limitations. This is a targeted validation of the downstream failure cases, not a validation of the five-metric suite. Annotators judged RED-flagged steps only; they did not score BC, ERC, ASA, SC, or GV, and we report no inter-annotator agreement statistic. Full metric-level human annotation would require experts to verify chunk boundaries, entity flows, dependency edges, and gateway conditions across 15–30 page documents, which we judged impractical at a scale that would yield meaningful agreement numbers. What the review does establish is a two-step chain of evidence: improving the intrinsic metrics through Best-of-N optimisation produces a measurable ACR gain, and the failure cases that gain eliminates are ones human experts confirm were real blockers.

I Initial Prompts

Below are the complete base prompts used for each of the five pipeline phases before any optimization. Template variables (e.g., `{{ input }}`) are replaced at runtime with the actual document text or prior phase outputs.

I.1 Segmentation

```
You are a document processing expert
     specializing in workflow and process
     documents such as Standard Operating
     Procedures (SOPs), business process
     descriptions, and operational manuals.

Your task is to read the document provided and
     identify every chunk boundary and
     reference segment boundary, returning
     boundary markers with start/end sentence
     snippets and classification labels.
```

```
Your sole job is boundary detection and
    classification.

---

## CORE DEFINITIONS

A CHUNK (PROCEDURAL segment) is a self-
    contained unit of work. It:
- Takes specific named inputs
- Performs one discrete action
- Produces specific named outputs
- Can be completed independently given its
    inputs and the reference store
- Has a clear actor and a clear action verb

A REFERENCE segment is a passage that describes
     what something IS. It:
- Defines EXACTLY ONE term, entity, data item,
    table, policy, appendix or template
- Provides background, context, or policy
    rationale for that single entity
- Contains no action that produces a new output
- Could be removed from the process flow
    without breaking the sequence

CRITICAL: Each REFERENCE segment must define
    only ONE entity. If a paragraph defines
    multiple entities (e.g., "Data 1" AND "
    Data 2"), split it into separate REFERENCE
     segments.

---

## BOUNDARY RULES

A boundary is the exact point where one segment
     ends and the next begins. A boundary is
    triggered by:
- Flow connector language: "then", "next", "
    after this", "once complete", "subsequently
    ", "upon completion", "following this"
- A new named output being introduced: "
    generates", "produces", "the result is", "
    outputs", "returns", "yields", "creates",
    "submits"
- A conditional gateway: "if", "when", "in case
     of", "depending on", "otherwise", "else",
     "should X occur", "in the event that"
- A new actor taking responsibility for the
    action
- A switch from procedural language to
    definitional language: "is defined as", "
    refers to", "note:", "background:", "for
    context", "where X means", "X consists of"

### When to SPLIT (DIFFERENT segments)

Create separate segments when:
- A flow connector separates two actions
- The second action consumes an output produced
     by the first action
- A different actor takes over
- A conditional branch or gateway separates
    them
- A reference/definitional passage appears
    between two actions and the process clearly
     resumes afterward

### When to KEEP TOGETHER (SAME segment)

Do NOT split when:
- They describe sub-steps of a single
    indivisible task
- No intermediate named output is produced
    between them
- The same actor continues throughout without
    interruption
- The second part cannot meaningfully complete
    without the first having just run
    immediately before it
- Numbered sub-steps (e.g., 1a, 1b, 1c) under a
     single heading that share the same actor
    and produce one combined output
- A paragraph that elaborates on the
    immediately preceding action without
    introducing a new actor or output

AIM for 15-25 total segments for a typical SOP.
     Avoid creating more than 30 segments
    unless the document genuinely has that many
     distinct steps with distinct outputs.

---

## CLASSIFICATION RULES

Label a segment PROCEDURAL if it:
- Contains a clear action verb with an
    identifiable actor
- Has something being done to something else
- Produces a new state, artifact, decision, or
    data item
- Uses procedural language: imperative
    constructs, active process verbs
- Represents a step that must occur in sequence
     for the process to advance

Label a segment REFERENCE if it:
- Defines a term, entity, concept, table, or
    template
- Lists properties or attributes of an entity
- Provides background, context, or policy
    rationale
- Uses definitional language
- Does not produce any new artifact or advance
    the process
- Could be removed from the process flow
    without breaking the sequence

When a single paragraph contains both a
    definition and an action, report TWO
    segments.

---

## OUTPUT FORMAT

Return ONLY a valid JSON object:
{
  "total_segments": <integer>,
  "segments": [
    {
      "segment_id": "S001",
      "classification": "PROCEDURAL",
      "start_sentences": "The first 2-3
          sentences, copied verbatim.",
      "end_sentences": "The last 2-3 sentences,
          copied verbatim.",
      "heading_hint": "brief label"
```

```
    }
  ]
}

## DOCUMENT

{{ input }}
```

I.2 Process Extraction

```
You are a process analyst specializing in
    extracting executable workflows from
    operational documents such as Standard
    Operating Procedures (SOPs), business
    process descriptions, and operational
    manuals.

Your task is to read the document below and
    identify only the distinct executable
    steps -- actions that can be performed by
    a human or system to produce a result.

---

## WHAT IS AN EXECUTABLE STEP

An executable step is a sentence or passage
    that describes an ACTION. It has:
- An actor: who or what performs the action (
    person, role, system, department)
- A verb: what action is being taken (submit,
    verify, approve, calculate, send, create,
    update, check)
- A result: what changes, gets produced, or
    moves forward as a consequence

## WHAT TO EXCLUDE

Do NOT extract the following:
- Definitions: passages that define what
    something IS rather than what to DO
- Background and Context: rationale, history,
    or explanatory information
- Policy Statements: rules or constraints
    without a specific action sequence
- Metadata and Notes: headers, footnotes,
    revision history

## EXTRACTION RULES

1. Keep logically grouped actions together: If
    multiple small actions form a single
    coherent task with one meaningful outcome,
     keep them as ONE step.
2. Step = meaningful business action: A step
    should represent a task that produces a
    recognizable outcome or decision.
3. Preserve conditional structure: If a step
    has a condition, keep the condition
    attached to the action.
4. Keep the actor explicit: Each step should
    make clear who or what performs the action.

5. Maintain document order: List steps in the
    order they appear in the document.
6. Use concise descriptions: Each step should
    be one clear sentence summarizing the task.

## TRIVIALLY SIMPLE CLASSIFICATION

A document is "trivially simple" if ALL of
    these conditions are met:
- It contains 3 or fewer executable steps
- It has no conditional branches
- It has no loops or iterative processes

## OUTPUT FORMAT

Return ONLY a valid JSON object:
{
  "steps": [
    "Actor performs action on object to achieve
         result.",
    ...
  ],
  "step_count": <integer>,
  "is_trivially_simple": <boolean>
}

## DOCUMENT

{{ input }}
```

I.3 Metadata Extraction

```
You are a document processing expert
    specializing in workflow and process
    documents such as Standard Operating
    Procedures (SOPs), business process
    descriptions, and operational manuals.

You are given a workflow/SOP document and a pre
    -computed list of segment boundaries with
    classifications (PROCEDURAL or REFERENCE).

Your task is to extract structured metadata for
     each segment and resolve dependencies
    between chunks.

---

## CRITICAL RULES - READ FIRST

1. ONLY use segment_ids from the provided
    segment list. Do NOT invent new IDs.
2. ONLY create reference_ids for REFERENCE
    segments in the provided list.
3. One PROCEDURAL segment = One chunk. One
    REFERENCE segment = One reference entry.
    Do not split or merge.
4. ref_pointers must ONLY contain reference_ids
     from REFERENCE segments.

---

## INPUT

You receive two inputs:
1. The full document text.
2. A segment list from a prior boundary-
    detection pass.

### Segment List

{{ segments_json }}

---
```

```
## EXTRACTION RULES FOR PROCEDURAL SEGMENTS (
    chunks)

For each segment classified as PROCEDURAL,
    extract:
- chunk_id: Sequential, format "C001", "C002",
    etc.
- segment_id: From the boundary list.
- actor: The role, system, or entity performing
     the action.
- action_summary: One sentence. Format: "[actor
    ] [verb] [object] to produce [output]".
- inputs: Array of named data items this chunk
    requires.
- outputs: Array of named data items this chunk
     produces.
- gateway_type: One of "none", "XOR", "AND", "
    OR".
- gateway_condition: Exact condition text if
    gateway_type is not "none". Null otherwise.

- depends_on: Array of chunk_ids whose outputs
    appear in this chunk's inputs.
- ref_pointers: Array of reference_ids that
    this chunk uses.

Rules:
1. Do not invent inputs or outputs. Only
    extract named data items from the document.

2. Inputs and outputs must be nouns or noun
    phrases, not verbs.
3. The same output name must not appear in two
    different chunks.
4. depends_on must only reference earlier
    chunk_ids.
5. Every chunk with gateway_type other than "
    none" must have a non-null
    gateway_condition.

## EXTRACTION RULES FOR REFERENCE SEGMENTS

For each segment classified as REFERENCE,
    extract:
- reference_id: Sequential, format "R001", "
    R002", etc.
- segment_id: From the boundary list.
- entity_name: Exact name of the entity as used
     in the document.
- definition: Full definition, under 100 words.
- used_by_chunks: Array of chunk_ids that
    reference this entity.

## OUTPUT FORMAT

Return ONLY a valid JSON object:
{
  "chunks": [ { "chunk_id": "C001", "segment_id
      ": "S001", "actor": "...", "
      action_summary": "...", "inputs": [], "
      outputs": [], "gateway_type": "none", "
      gateway_condition": null, "depends_on":
      [], "ref_pointers": [] } ],
  "references": [ { "reference_id": "R001", "
      segment_id": "S003", "entity_name":
      "...", "definition": "...", "
      used_by_chunks": [] } ]
}

## DOCUMENT

{{ input }}
```

I.4 Data Dependency

```
You are a dependency resolution specialist. You
     are given a unified JSON containing
    chunks and references extracted from a
    workflow/SOP document.

Your task is to resolve all data-flow
    dependencies and reference linkages.

---

## CRITICAL RULES - READ FIRST

1. ONLY use IDs that exist in the input data.
    Do NOT invent new IDs.
2. ref_pointers must ONLY reference
    reference_ids from the "references" array.
3. depends_on must ONLY reference chunk_ids
    from the "chunks" array.
4. used_by_chunks must ONLY reference chunk_ids
     from the "chunks" array.

---

## INPUT DATA

{{ unified_json }}

---

## RESOLUTION RULES

### depends_on (chunk-to-chunk data flow)

For each chunk:
1. Scan its inputs list.
2. Find which earlier chunk has that item in
    its outputs list.
3. Add that chunk_id to depends_on, along with
    a reason explaining the data-flow
    relationship and the shared_entities that
    connect the two chunks.
4. depends_on must only reference earlier
    chunks unless the document explicitly
    describes a loop.

Validation: For every depends_on entry, verify
    the referenced chunk's outputs contain at
    least one item from the current chunk's
    inputs. Remove any entry that fails.

### ref_pointers (chunk-to-reference linkage)

For each chunk:
1. Scan its inputs list AND action_summary text.

2. For each entity_name in the references list,
     check if it appears.
3. Add the reference_id to ref_pointers if
    found.

### used_by_chunks (reference-to-chunk linkage)

For each reference:
```

```
1. Scan all chunks' inputs lists AND
    action_summary text.
2. If the entity_name appears, add that
    chunk_id to used_by_chunks.

Note: used_by_chunks must be the exact inverse
    of ref_pointers.

---

## OUTPUT FORMAT

Return ONLY a valid JSON object:
{
  "chunks": [
    {
      "chunk_id": "C003",
      "segment_id": "S003",
      "depends_on": [
        { "chunk_id": "C001", "reason": "C003
            inputs 'review_decision' which is
            output by C001", "shared_entities":
             ["review_decision"] }
      ],
      "ref_pointers": ["R001"]
    }
  ],
  "references": [
    { "reference_id": "R001", "segment_id": "
        S005", "used_by_chunks": ["C003"] }
  ]
}

IMPORTANT: Each entry in depends_on is an
    OBJECT with chunk_id, reason, and
    shared_entities -- NOT a bare string.
```

I.5 Process Dependency

```
You are analyzing process flow dependencies
    between workflow chunks.

Given:
1. The ORIGINAL DOCUMENT describing the
    workflow
2. A list of CHUNKS already extracted from the
    document

Your task: Determine which chunks must execute
    BEFORE other chunks based on the PROCESS
    FLOW order in the document.

Process flow dependency means:
- The document explicitly states Step A happens
     before Step B
- One action logically must precede another per
     the workflow
- There is a sequential order described in the
    procedure

## ORIGINAL DOCUMENT
{{ document }}

## CHUNKS
{% for chunk in chunks %}
{{ chunk.chunk_id }}: {{ chunk.action_summary
    }}
{% endfor %}

## INSTRUCTIONS

1. Read the document to understand the workflow
     sequence
2. For each chunk, identify which other chunks
    must complete BEFORE it
3. Only include dependencies that are PROCESS-
    BASED (order of execution)
4. Focus on sequential flow, branching points,
    and convergence points

## OUTPUT FORMAT

Return ONLY a valid JSON object:
{
  "process_edges": [
    {
      "from": "C001",
      "to": "C002",
      "reason": "The SOP states form submission
          (C001) must complete before system
          validation (C002)",
      "edge_type": "sequential"
    }
  ]
}

Rules:
- "from" chunk must complete before "to" chunk
- Only include edges where document clearly
    indicates ordering
- "reason" must explain WHY this ordering
    exists
- "edge_type" must be one of: "sequential", "
    conditional_branch", "convergence"
```

I.6 ERC Metric: Entity Extraction

```
You are an entity extraction expert. Extract
    all named entities from the provided text
    that are relevant to workflow/process
    execution.

## Focus On
Extract entities that represent:
- Data items, files, documents, reports,
    spreadsheets
- System names, tools, databases, queries
- Templates, forms, email formats
- Status values, codes, identifiers
- Roles, actors, teams, departments
- Process outputs, artifacts, deliverables

## Do NOT Extract
- Generic verbs or actions
- Common words (the, a, is, etc.)
- Pronouns (it, they, this, etc.)

## Text to Analyze
{{ chunk_text }}

Return ONLY a JSON array of entity strings.
Example: ["Query Tool", "Email Template 1", "
    sourcing file", "Process Owner"]
```

I.7 ERC Metric: Entity Matching (Batch)

```
You are an entity matching expert. Given a list
     of entities from a chunk's text,
```

```
    determine if EACH entity can be traced to
    any of the available sources.

## Entities to Match
{% for entity in entities %}
{{ loop.index }}. "{{ entity }}"
{% endfor %}

## Available Sources
### Source 1: Chunk's Own Outputs
{{ chunk_outputs }}

### Source 2: Parent Chunk Outputs and Text
{{ parent_outputs }}
{{ parent_texts }}

### Source 3: Referenced Definitions
{{ reference_entities }}

## Task
For EACH entity, determine if it can be traced
    to ANY source. Consider fuzzy matching:
    synonyms, partial matches, abbreviations,
    semantic equivalence.

Return a JSON ARRAY with one object per entity:
- entity: exact entity name
- matched: boolean
- source_type: "output" | "parent" | "reference
    " | null
- matched_to: specific item matched, or null
- confidence: 0.0-1.0
- reasoning: brief explanation (1 sentence)
```

I.8 ASA Metric: Restricted Agent

```
You are an agent executing a specific task from
     a workflow. You have LIMITED context --
    only the task description and specifically
     referenced information.

## Available Context
### Reference Information
{{ references }}

### Input Data
{{ inputs }}

## Your Task
{{ task_description }}

## Full Task Text
{{ chunk_text }}

## Instructions
Based ONLY on the context provided above:
1. Perform the action described in the task
2. Determine what output you would produce
3. Report if you feel you are missing any
    context

Return JSON: { action_taken, output_value,
    output_name, confidence, missing_context }
```

I.9 ASA Metric: Full-Context Agent

```
You are an agent executing a specific task from
     a workflow. You have FULL context -- the
    entire document is available to you.

## Full Document
{{ full_document }}

## Input Data
{{ inputs }}

## Your Task
{{ task_description }}

## Instructions
Based on ALL the context available in the
    document above:
1. Perform the action described in the task
2. Determine what output you would produce
3. Report if you feel you are missing any
    context

Return JSON: { action_taken, output_value,
    output_name, confidence, missing_context }
```

I.10 ASA Metric: Agreement Judge

```
You are an agreement judge comparing two agent
    outputs for the same task.

## Task Description
{{ task_description }}

## Restricted Agent Output (Limited Context)
{{ restricted_output }}

## Full-Context Agent Output (Full Document)
{{ full_context_output }}

## Agreement Classifications
- IDENTICAL: Both produced exactly the same
    output
- EQUIVALENT: Same substantive conclusion,
    different phrasing
- PARTIAL: Outputs overlap but restricted agent
     missed information
- CONTRADICTORY: Opposite or conflicting
    conclusions
- UNRELATED: Outputs do not address the same
    thing

Return JSON:
- agreement_class: one of the above
- rationale: one-sentence explanation
- failure_mode: if PARTIAL/CONTRADICTORY/
    UNRELATED, one of:
  "reference_gap" | "dependency_gap" | "
      boundary_error" | null
```

I.11 GV Metric: Graph Validation

```
You are validating workflow graph structure
    against the source document.

Given:
1. The ORIGINAL DOCUMENT
2. SUBGRAPHS extracted from the workflow
    dependency graph

Each subgraph is boundary-safe: every node's
    incoming and outgoing edges within the
    subgraph are complete.
```

```
Subgraph types:
- cut_prefix: All ancestors of a cut-point node
     plus the cut point itself.
- tail: Remaining nodes after cut-point
    prefixes.
- full: Complete graph when no natural cut
    points exist.
- component: A disconnected subprocess.

## ORIGINAL DOCUMENT
{{ document }}

## SUBGRAPHS TO VALIDATE
{{ subgraphs }}

## VALIDATION CRITERIA
1. Edges Correctness: Does each edge exist in
    the document?
2. Missing Edges: Dependencies that should
    exist but are missing?
3. Gateway Semantics: Do gateway types match
    the document?
4. Parallel/Diamond Structures: Are convergence
     edges correct?

Return JSON:
{
  "subgraph_validations": [
    { "subgraph_num": 1, "subgraph_type": "
        cut_prefix", "center_node": "C007",
      "is_valid": true, "structure_correct":
          true,
      "missing_edges": [], "spurious_edges": [],
           "gateway_issues": [],
      "reasoning": "brief explanation" }
  ],
  "overall_validity": 0.85,
  "missing_edges_feedback": "...",
  "incorrect_edges_feedback": "..."
}
```

I.12 Feedback Generator

```
You are a chunking quality analyst. Analyze a
    chunk manifest and its evaluation metrics,
     then generate actionable feedback.

## INPUTS
1. Chunk Manifest (chunks with text, actor,
    inputs, outputs, depends_on)
2. Process Extraction Steps (atomic executable
    steps as ground truth)
3. Evaluation Metrics (BC, ERC, ASA, Step
    Coverage, Graph Validity)

## ANALYSIS TASKS
1. Step Alignment: Map each chunk to process
    steps, check merge compatibility
2. Under-Chunking: Chunks covering incompatible
     steps (multi-actor, conditional
    divergence)
3. Over-Chunking: Chunks that should merge (
    same actor, linear chain, very short)
4. Low-Metric Chunks: Diagnose specific issues
    for low BC/ERC/ASA
5. Step Coverage: Review uncovered steps (
    missing vs merged vs reference)
6. Graph Validity: Review missing/spurious
    edges, gateway issues

## OUTPUT
Return JSON with:
- overall_diagnosis: UNDER_CHUNKING |
    OVER_CHUNKING | HEALTHY | MIXED
- under_chunking_issues: [{chunk_id,
    covered_steps, issue, suggested_split}]
- over_chunking_issues: [{chunk_ids, issue,
    suggested_merge}]
- boundary_issues: [{boundary, bc_score,
    recommendation}]
- entity_coverage_issues: [{chunk_id, erc_score
    , missing_entities}]
- step_coverage_issues: [{uncovered_step,
    reason, recommendation}]
- graph_validity_issues: [{issue_type,
    chunk_ids, reasoning, recommendation}]
- summary_rules_for_refinement: [actionable
    rule strings]

CRITICAL: When referencing chunks, always
    include the action_summary in parentheses
    after the ID.

## CHUNK MANIFEST
{{ manifest }}

## PROCESS EXTRACTION STEPS
{{ process_steps }}

## METRIC RESULTS
{{ metrics }}
```

I.13 Prompt Optimizer

```
You are a prompt engineering expert. Improve a
    chunking prompt based on feedback from a
    suboptimal run.

## PHASE BEING OPTIMIZED
{{ phase_name }}
(Phase responsibilities are injected
    dynamically based on which prompt is being
     optimized.)

## FEEDBACK FROM PREVIOUS RUN
### Quality Metrics
{{ metrics }}

### Phase-Specific Feedback
(Feedback is partitioned by keyword routing.
    Each phase only sees its own relevant
    feedback: boundary issues for Phase 1,
    entity coverage for Phase 2, dependency
    issues for Phase 3, process ordering for
    Phase 4.)

{{ feedback }}

## CURRENT PROMPT TO OPTIMIZE
{{ current_prompt }}

## YOUR TASK
Rewrite the prompt to fix the issues relevant
    to this phase.

CRITICAL CONSTRAINTS:
- Phase 1: Apply merge/split guidance, ensure
    step coverage
```

```
- Phase 2: Do NOT add merge/split logic, keep "
    one segment = one chunk"
- Phase 3/4: Do NOT create new IDs, ensure
    valid DAG, remove spurious edges

HARD CONSTRAINTS:
1. Rewritten prompt MUST NOT exceed 12000
    characters
2. When adding a rule that contradicts an
    existing one, REMOVE the old rule
3. Prefer concise, high-impact rules over
    verbose explanations

Output the complete rewritten prompt -- no
    explanations, no markdown fences, just the
     raw prompt text.
```

J Case Study: Optimization Trace for Aircraft Inspection SOP from SOP-Bench Dataset

This section provides a complete trace of ProChunker-BoN on the Aircraft Inspection SOP from SOP-Bench, covering depth 0 (root evaluation with base prompts) through depth 1 (first optimization step). The search ran for 3 depths with branch factor 3, producing 10 evaluations over $\sim$31 minutes. The search path was root $\rightarrow$ d1_c2 $\rightarrow$ d3_c2, improving composite from 0.659 to 0.795 ($+0.135$). We focus on the root $\rightarrow$ d1_c2 transition (largest single improvement, $+0.116$).

J.1 Depth 0: Root Evaluation

J.1.1 Pipeline Output

The base prompts produce 9 chunks and 5 reference entries for this pre-flight airworthiness verification SOP. The SOP describes a linear multi-layered inspection protocol: aircraft identification $\rightarrow$ mechanical components $\rightarrow$ electrical systems $\rightarrow$ discrepancy reporting $\rightarrow$ maintenance record reconciliation $\rightarrow$ final report.

Table 20: Root Phase 2 output: 9 chunks for Aircraft Inspection SOP.

ID	Actor	Action Summary	Deps
C001	Maint. Personnel	Execute AVM verification using aircraft_id and tail_number	–
C002	Maint. Personnel	Verify component serial numbers using SNVA	C001
C003	Maint. Personnel	Validate electrical systems authentication (ESAP)	C002
C004	Maint. Personnel	Execute mechanical diagnostics	C001
C005	Maint. Personnel	Generate discrepancy reports for failures	C003, C004
C006	Maint. Personnel	Execute cross-check reporting, update MRVS	C005
C007	Maint. Personnel	Reconcile maintenance records with findings	C006
C008	Maint. Personnel	Generate Airworthiness Verification Report	C007
C009	Maint. Personnel	Perform incident reporting (if applicable)	C008

Only 5 of the 11 available reference entities (AVM, CTT, ESAP, MRVS, SNVA, input parameters, thresholds) were captured. Missing references include Component Tolerance Threshold (CTT), battery status thresholds, input parameter specifications, and the digital maintenance record update format.

J.1.2 Root Metrics

Table 21: Root evaluation metrics for Aircraft Inspection.

	BC	ERC	ASA	SC	GV	Comp.
Root	0.473	0.768	0.856	1.000	0.500	**0.659**

GV = 0.500 is the critical bottleneck. The validator found: (i) over-segmentation produced two chunks (C004, C005) with unclear dependency ordering; (ii) a spurious parallel edge from C001 to C004 conflicting with the document's strictly sequential procedure.

ERC = 0.768: Several domain entities (battery_status thresholds, component weight $\pm 2\%$ variance, installation_time 24-hour compliance window) appear in chunk text but are not traceable to declared inputs or reference entries.

ASA = 0.856: High agreement (7/9 EQUIVALENT). Two PARTIAL scores on C005 (discrepancy reporting) where the restricted agent lacked threshold definitions needed to determine which condition constitutes a "failure."

SC = 1.000: All 15 process steps mapped to chunks.

J.2 Feedback and Routing

The feedback generator diagnoses the state as *MIXED* and produces 6 rules:

1. Merge C004 into C002/C003 (mechanical diagnostics is part of mechanical inspection) $\rightarrow$ P1
2. Remove spurious parallel edge C001$\rightarrow$C004; enforce strict sequential ordering per document sections $\rightarrow$ P3, PD
3. Add reference entries for CTT ($\pm 2\%$ threshold), battery_status thresholds (Operational/Low/-Critical), installation_time 24-hour window $\rightarrow$ P1, P2
4. Add input parameter specifications (Aircraft Documentation, Component Verification Data, Electrical Systems Data) as reference entries $\rightarrow$ P1, P2
5. Capture the output format specification (Airworthiness Verification Report structure) as a reference entry $\rightarrow$ P1, P2

6. Reduce chunk count from 9 to ∼6 to align with the document's five procedural sections plus one output section → P1

J.3 Depth 1: Candidate Selection

Table 22: Depth 1 candidates. d1_c2 ($\tau = 0.9$) is selected.

Cand.	τ	Ch.	Ref.	BC	ERC	ASA	SC	GV	Comp.
d1_c0	0.5	5	11	.478	.971	.640	.933	1.0	.740
d1_c1	0.7	5	9	.462	1.000	.740	.933	1.0	.759
d1_c2	**0.9**	**5**	**12**	**.462**	**1.000**	**.820**	**.933**	**1.0**	**.775**

All three candidates fix GV to 1.0 (the spurious parallel edge and over-segmentation are resolved). The key differentiator is ASA: $\tau = 0.9$ achieves 0.820 vs. 0.740 ($\tau = 0.7$) and 0.640 ($\tau = 0.5$). The aggressive temperature produced richer reference entries (12 vs. 9 and 11), giving the restricted agent enough declared context to match the full-context agent's outputs.

d1_c2 is accepted: $\Delta = 0.775 - 0.659 = +0.116 > \epsilon = 0.005$.

J.4 What Changed: Root → d1_c2

Chunk count: 9 → 5. Merged C004 (mechanical diagnostics) into C002 (mechanical inspection), removed C009 (incident reporting folded into C006's report generation), consolidated C007+C008 into a single reconciliation+report chunk.

Reference entries: 5 → 12. Added: CTT definition with $\pm 2\%$ variance, battery status thresholds (Operational >80%, Low <80%, Critical <40%), circuit continuity retry limit (max 3), installation_time 24-hour window, Aircraft Documentation input parameters, Component Verification Data input parameters, Electrical Systems Data input parameters.

Graph structure: Strict linear chain C001→C002→C003→C004→C005, matching the document's section ordering (5.1→5.2→5.3→5.4→5.5+6). No parallel edges.

Metric impact:

- GV: 0.500 → 1.000 (+0.500). Spurious edge removed; sequential ordering enforced.
- ERC: 0.768 → 1.000 (+0.232). All domain entities now traceable to reference entries or declared inputs.
- ASA: 0.856 → 0.820 (−0.036). Slight regression—fewer chunks means each chunk is denser, and the restricted agent occasionally missed one sub-procedure within a larger chunk.
- SC: 1.000 → 0.933 (−0.067). One process step (incident reporting) was folded into the report generation chunk rather than standalone; the step matcher counts it as a partial match.
- BC: 0.473 → 0.462 (−0.011). Negligible change; fewer boundaries means fewer measurements.

The composite rose from 0.659 to 0.775 (+0.116) despite small regressions on ASA, SC, and BC, because the harmonic mean is dominated by the weakest metric—and the weakest metric (GV) jumped from 0.500 to 1.000.

J.5 Depth 3: Final Improvement

The search continued for two more depths. At depth 3, candidate d3_c2 ($\tau = 0.9$) achieves the final best composite of 0.795:

Table 23: Final search path node: d3_c2.

	BC	ERC	ASA	SC	GV	Comp.
d3_c2	0.477	0.922	0.900	1.000	1.000	**0.795**

The final manifest has 6 chunks and 11 references. The gain from depth 1 to depth 3 came from: (i) recovering SC to 1.000 by splitting the over-merged report chunk back into reconciliation (C005) and report generation (C006); (ii) lifting ASA from 0.820 to 0.900 by adding more precise reference entries that the restricted agent could use.

The final workflow graph is a clean linear DAG: C001 (Aircraft ID Validation) → C002 (Mechanical Inspection) → C003 (Electrical Authentication) → C004 (Discrepancy Reporting) → C005 (Record Reconciliation) → C006 (Airworthiness Report).