EditPPT: Faithful Long-Deck Slide Editing via Structured Tool-Using Multi-Agent with Dual-Modal Validators

Anonymous ACL submission

Abstract

Automating slide editing requires simultaneously satisfying modification accuracy, preservation fidelity, and robustness to deck length. Existing LLM-based systems often fail on real-world presentation files because they rely on idealized intermediate representations or open-ended code generation, which are prone to cascading errors in long decks. We introduce EDITPPT, a multi-agent framework that reformulates slide editing as a *constrained tool-selection problem*. By executing localized shape-level operations through the native PowerPoint COM interface, EDITPPT narrows the LLM action space while preserving the application-resolved structure of user-authored decks. By separating validation across modalities, our dual-modal validation provides more robust assessment of both instruction fidelity and visual quality. We also present DECKEDITBENCH, a benchmark with 28 human-authored decks, 582 slides, and 183 editing prompts across short, medium, and long deck tiers. Experiments show that EDITPPT achieves a 99.5% execution rate, 88.7% slide-targeting F1, 82.5% instruction following, and 91.5% object preservation overall, while maintaining strong performance on long decks. Our code and benchmark are available at **here**.

1 Introduction

Slide decks are a ubiquitous medium for organizing and communicating information across business, academic, and professional contexts (Shi et al., 2025). Naturally, automating slide workflows has emerged as a high-value task. Recent systems have made rapid progress in presentation generation from natural-language instructions, source documents, and user preferences (Ge et al., 2025; Zheng et al., 2025; Yang et al., 2025; Zeng et al., 2025; Zheng et al., 2026; Wu et al., 2026); however, reliable slide *editing* remains a significant challenge (Jung et al., 2026; Ofengenden et al., 2025; Jang et al., 2026). Unlike generation, slide editing is strictly bound by three simultaneous constraints: modification accuracy, preservation fidelity, and robustness to deck length. An editing system must execute a requested modification while leaving unrelated content and formatting untouched. Crucially, this preservation behavior must remain stable as the presentation grows in length and complexity, since unintended shifts in fonts, layouts, or shapes can render an entire deck unusable.

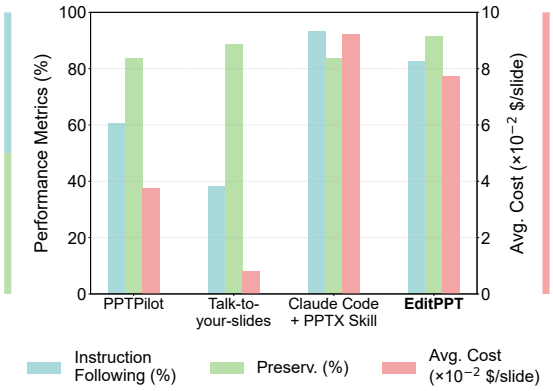

Figure 1: EDITPPT maintains strong editing across deck lengths while preserving unrequested objects.

A central obstacle is the mismatch between idealized editing representations and native `.pptx` files. Many LLM-based pipelines operate on regularized HTML, code, or structured markup representations where layers, coordinates, and object boundaries are explicit (Ge et al., 2025; Zheng et al., 2025; Yang et al., 2025; Jang et al., 2026). User-authored presentations rarely satisfy these assumptions because they include grouped objects, implicit formatting, inconsistent layouts, and artifacts whose semantic role differs from their file-level type (Wu et al., 2026; Pan et al., 2026). Direct manipulation of native files through code generation or OOXML editing avoids some representational loss, but it requires the LLM to synthesize brittle stateful pro-

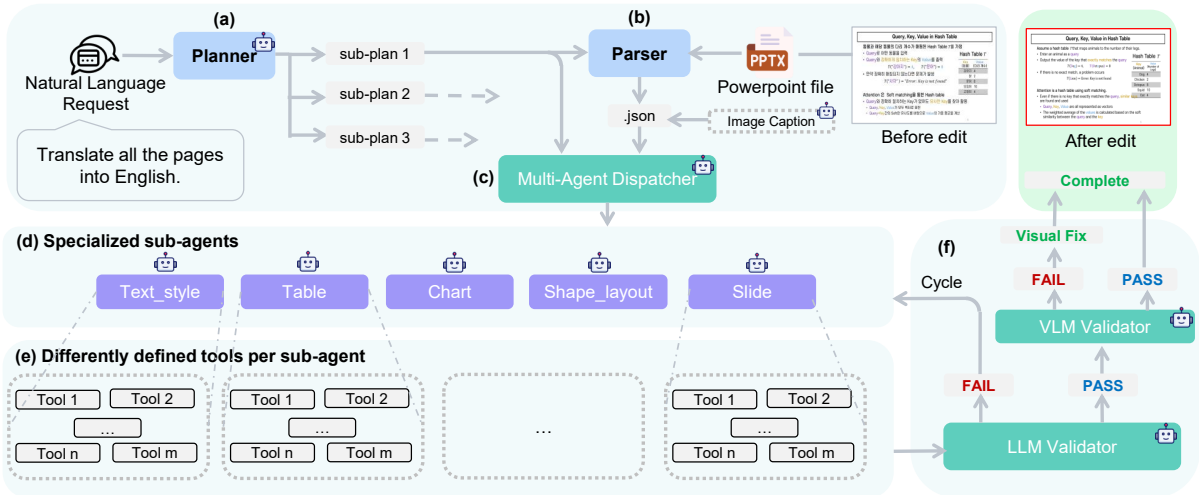

Figure 2: (a) The Planner translates the user's high-level intent into a specification targeting specific slides. (b) The Parser converts target slides into structured JSON; images are captioned by a VLM. (c) The Multi-Agent Dispatcher routes each slide element to the appropriate sub-agent. (d) Each sub-agent invokes tools with task-specific parameters. (e) The tools execute edits via COM. (f) The LLM Validator re-parses the edited slides, compares pre- and post-edit states, and verifies task completion. Once passed, a VLM checks for visual defects.

grams or edit a complex schema where small targeting errors can propagate across slides (Anthropic, 2026d; Jung et al., 2026; Ofengenden et al., 2025). These failure modes become more pronounced in long, dense decks, yet existing benchmarks still emphasize relatively short, generated, HTML-based, or modular editing settings.

Our key observation is that practical edits are more regular than the files that contain them. Although native decks are structurally irregular, most user requests decompose into bounded operations over recurring object types, including text frames, tables, charts, shapes, and images. This motivates EDITPPT, a length-robust, layout-aware framework that formulates slide editing as a *constrained tool-selection problem*: the LLM interprets the instruction and selects typed operations, while execution is handled by deterministic shape-level tools through the COM interface. EDITPPT combines a COM-based parser, specialist edit agents with narrow tool scopes, and a modality-decoupled validator that checks both instruction fulfillment and visual layout anomalies, thereby reducing the cascading errors caused by open-ended program synthesis or raw file-structure modification.

To evaluate this setting, we introduce DECKEDIT-BENCH, a depth-oriented benchmark with 28 human-authored decks, 582 slides, and 183 natural-language editing prompts across short ($\leq$10 slides), medium (11 to 30 slides), and long (>30 slides) tiers. As summarized in Figure 1, EDITPPT completes 99.5% of benchmark instances and achieves 88.7% slide-targeting F1, 82.5% instruction following, and 91.5% object preservation overall; qualitative comparisons in Figure 3 further illustrate its ability to apply requested edits while preserving unrelated slide content. On long decks, it maintains 90.5% slide F1, 86.6% instruction following, and 91.7% preservation, while PPTPilot and Talk-to-Your-Slides show stronger instruction-following degradation as deck length increases. These results indicate that native, constrained tool execution mitigates length-induced error cascades in practical slide editing.

Contributions. Our main contributions are summarized as follows:

- We propose EDITPPT, a tool-augmented multi-agent framework that reformulates slide editing as a constrained tool-selection problem, executing shape-level operations via the native PowerPoint COM interface.
- We release DECKEDIT-BENCH, a depth-oriented benchmark with 28 human-authored decks and 183 prompts that capture the structural complexity and length diversity of real-world presentation workflows.
- We empirically demonstrate that EDITPPT significantly outperforms existing code-generation and structure-editing baselines

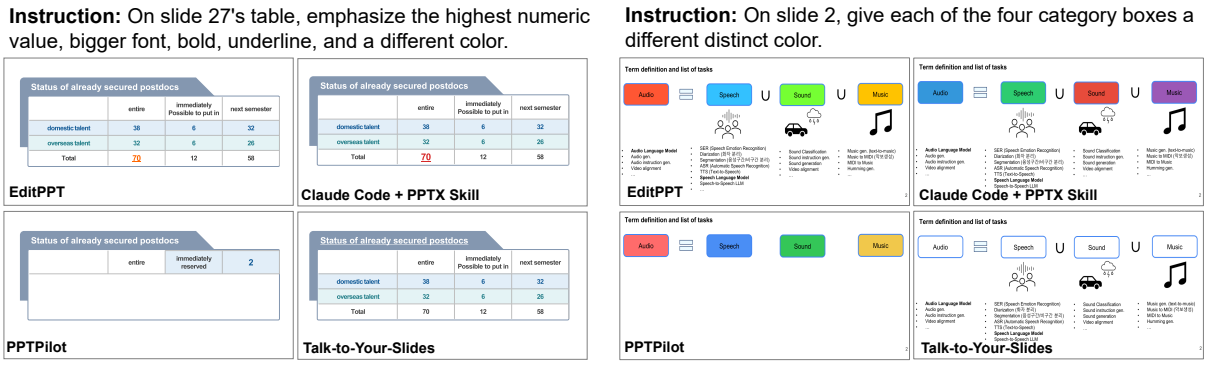

Figure 3: Qualitative comparison of presentation-editing strategies. EDITPPT constrains execution to typed shape-level tools, reducing accidental edits to unrelated slide content.

while exhibiting strong stability against deck-length scaling.

2 Related Work

Presentation automation. A broad line of work has studied automated presentation construction, including instruction-driven slide synthesis, academic paper-to-slide generation, and interactive deck authoring systems (Ge et al., 2025; Zheng et al., 2025; Yang et al., 2025; Zeng et al., 2025; Zheng et al., 2026; Wu et al., 2026). These systems typically operate in regularized representation spaces, such as HTML, structured markup, or generated code, which are effective when the system controls the layout from the beginning but are less suited to editing user-authored `.pptx` files with grouped objects, inconsistent layers, implicit formatting, and semantic-structural mismatches. Our work therefore focuses on a distinct setting: instruction-driven editing of native PowerPoint decks, where the system must identify the intended targets, apply local modifications, and preserve unrelated content and formatting in the original file.

Presentation editing benchmarks. Recent benchmarks have begun to evaluate presentation editing as a first-class agentic task. Talk-to-Your-Slides introduces TSBench for efficient object-model-based slide editing (Jung et al., 2026), PPTArena evaluates in-place PowerPoint edits over real slides with structural and visual judging (Ofengenden et al., 2025), and DECKBench studies multi-turn academic slide generation and editing in a multi-agent HTML workflow (Jang et al., 2026). These efforts establish the importance of instruction following, locality, and visual quality for slide editing. DECKEDIT-BENCH complements them by emphasizing length scaling over native, human-authored decks with heterogeneous PowerPoint objects and layout artifacts.

Stage	Input	Output
Planning	User instruction	Slide-level tasks
Parsing	Target slides	Shape JSON states
Dispatch	Tasks and shapes	Specialist calls
Execution	Typed arguments	COM edits
Text Validation	Before/after states	Accept or recover
Vision Validation	After slide Capture	Accept or visual-fixer call

Table 1: The EDITPPT pipeline keeps each step grounded in PowerPoint-resolved objects, from planning to validation.

3 Method

3.1 Overview and Design Rationale

EDITPPT processes an editing request through a five-stage pipeline (Figure 2). Given a user-supplied `.pptx` deck and a natural-language instruction, the system (i) parses the relevant slides into structured representations of native PowerPoint objects, augmented with selective visual metadata, (ii) decomposes the instruction into a per-slide task list, (iii) dispatches each task to a specialist agent, (iv) executes deterministic shape-level operations through the PowerPoint COM interface, and (v) validates the result using complementary structural and visual validators before either accepting the edit or triggering recovery. The pipeline runs sequentially against a single PowerPoint application instance, with slide-level checkpoints used to support recovery. Table 9 summarizes the input and output of each stage.

The key design choice is to use the running PowerPoint application as a shared interface for both parsing and editing, following recent evidence that object-model-based slide manipulation can be more efficient and style-preserving than pixel-level

System	Deck Length	Performance					Efficiency	
		Execution Rate ↑	Slide F1 ↑	Instruction Following ↑	Preserv. ↑	Visual Quality ↑	Avg. Time (s/slide) ↓	Avg. Cost ($\times 10^{-2}$\$/slide) ↓
PPTPilot	Short	66.7%	80.8%	69.7%	83.8%	84.8%	24.9	4.93
	Medium	86.4%	83.2%	68.1%	78.5%	90.0%	8.9	3.03
	Long	85.4%	69.8%	52.6%	88.2%	84.7%	11.5	4.38
	Overall	80.3%	76.8%	60.6%	83.6%	87.3%	**11.3**	3.75
Talk-to-Your-Slides	Short	64.8%	75.9%	52.1%	72.6%	76.3%	122.3	3.24
	Medium	39.5%	61.3%	39.6%	85.6%	82.6%	109.3	0.63
	Long	50.0%	57.7%	34.9%	93.2%	73.7%	89.2	0.47
	Overall	49.7%	60.7%	38.1%	88.6%	77.8%	100.9	**0.79**
Claude Code + PPTX Skill	Short	100.0%	93.8%	83.3%	81.8%	94.6%	83.8	26.85
	Medium	100.0%	95.6%	92.8%	86.2%	91.6%	27.3	8.94
	Long	100.0%	95.2%	95.4%	82.0%	84.7%	26.0	7.09
	Overall	**100.0%**	**95.3%**	**93.4%**	83.8%	88.2%	30.4	9.20
EditPPT	Short	100.0%	86.6%	76.8%	93.7%	92.7%	69.6	9.45
	Medium	100.0%	87.0%	78.7%	90.9%	93.4%	36.5	5.60
	Long	97.9%	90.5%	86.6%	91.7%	86.7%	40.7	9.36
	Overall	99.5%	88.7%	82.5%	**91.5%**	**89.9%**	40.8	7.74

Table 2: Overall and length-wise comparison on DECKEDIT-BENCH. Each evaluation instance corresponds to one deck-prompt pair. Slide F1 denotes slide-targeting F1. Instruction Following is measured as the full-or-partial editing success rate on successfully targeted slides. Preserv. denotes Object Preservation Rate, reflecting shape-level preservation capability. Average time and cost are reported per slide, where cost is in units of 10^{-2} USD. Bold values indicate the best result, and underlined values indicate the second-best result among Overall results.

GUI control for structured slide edits (Jung et al., 2026). Static libraries such as `python-pptx` expose only a partial view of the application-resolved slide state, making conditional edits difficult. Direct OOXML editing[1] requires manipulating fragmented low-level XML structures, making LLM-based editing both costly and brittle. EDITPPT, by contrast, uses COM to query and modify the live PowerPoint application, recovering PowerPoint-resolved object state for parsing and applying bounded shape-level operations for editing within the same application context.

This distinction is especially important for Claude-style PPTX editing workflows that inspect or modify serialized OOXML directly. As shown in Figure 4, the EDITPPT parser is not merely a different serialization of the same file contents, but a compact application-resolved representation that removes package-level XML redundancy while preserving editable object state. Across 2,313 parsed slides, this reduces input size from 36.48M raw-XML tokens to 8.45M parser tokens, a 76.8% reduction, and lowers the estimated input cost by 4.32×(Table 5). The advantage is larger on complex slides because raw OOXML repeats style, coordinate, relationship, and object metadata across many XML parts, whereas EDITPPT exposes the objects that an editing agent can actually act on.

This representation and execution alignment reduces ambiguity throughout the pipeline. The parser exposes editable objects as PowerPoint resolves them, specialist agents select only type-compatible tools, and validators check both semantic completion and visual layout integrity. As a result, local editing failures are less likely to propagate across long, content-dense decks.

3.2 Planning and Targeted Parsing

EDITPPT first converts a user instruction into slide-level editing tasks and then parses only the slides needed to execute those tasks. The planner operates in two stages. First, it interprets the high-level scope of the instruction, such as whether the edit applies to a specific slide, a subset of slides, or the entire presentation. For example, instructions such as "change the title on slide 3" specify an explicit target slide, whereas instructions such as "remove the logo from all slides" induce a deck-wide scope. Second, after the relevant slides are parsed, the planner refines the request into shape-level subtasks that specify the target slide, target object, and intended action.

Structure-first grounding. For each target slide, our method uses a structure-first slide representation augmented with selective visual information. Vision-based GUI agents and screen parsers have made progress in grounding actions from screen-

[1] Office Open XML (OOXML) is the XML-based package format underlying modern Microsoft Office files, including `.pptx`. A PowerPoint deck is stored as a ZIP package containing XML parts, relationship files, and media assets. Because XML encodes hierarchy and metadata through explicit tags and attributes, serialized OOXML tends to be verbose and token-inefficient for LLM-based processing.

System	Slide-level Targeting			Object-level IF		
	Corr ↑	Miss ↓	Wrng ↓	Succ ↑	Part ↑	Fail ↓
PPTPilot	65.1%	24.2%	10.7%	39.0%	**21.4%**	39.6%
T2S	42.5%	50.0%	7.5%	24.6%	13.4%	62.1%
Claude Code	**94.7%**	**4.7%**	**0.6%**	**83.8%**	9.4%	**6.8%**
EditPPT	79.4%	13.5%	7.1%	65.7%	16.8%	17.5%

Table 3: Breakdown of modification accuracy. T2S denotes Talk-to-Your-Slides. Slide-level targeting is reported as Correct (Corr), Missing (Miss), and Wrong (Wrng) rates. Object-level instruction following (IF) is reported as Success (Succ), Partial (Part), and Fail rates. Bold values indicate the best value, and underlined values indicate the second-best value.

shots (Xie et al., 2024; Lu et al., 2024), but applying them across long decks is costly and does not expose the fine-grained editable state needed for precise modification, such as font attributes, shape colors, table structure, chart data, or object identifiers. We therefore use a COM-based slide parser as the primary representation. Visual signals are used selectively, with image captions extracted during parsing and rendered slide images generated only on demand for visual validation.

The parser produces a JSON object enumerating native PowerPoint objects and their properties, including object type, geometry, text and font attributes, table and chart contents, image metadata, and slide-level elements. Each parsed shape is associated with a `Shape_Id`, which serves as the addressing handle for downstream tool calls. For embedded images, the parser stores short captions as image-level metadata, allowing agents to identify images by visual content when users refer to them explicitly, e.g., "delete the dog photo." This also helps resolve non-textual slide content such as diagrams or table screenshots that would otherwise be invisible in the `.pptx` structure.

Although COM exposes PowerPoint-resolved object state, its raw interface is insufficient for reliable agentic editing. We observed three gaps: native COM properties and methods often lack granularity for rich-text editing, default paragraph segmentation often misaligns with instruction-level semantic units, and merged-cell structures in tables are exposed implicitly rather than as addressable logical cells. We therefore add reconstruction procedures for run-level formatting, paragraph-level segmentation, and merged table cells.

For text, the parser reconstructs formatting runs and logical paragraph units so that mixed-format text and paragraph-like spans can be edited without collapsing local style. For tables, it infers merged-cell structure geometrically, allowing downstream agents to address logical cells rather than raw grid positions. Full algorithmic details and edge cases are provided in Appendices E.1 and E.2.

The resulting slide-wise shape-level tasks are grounded in PowerPoint-resolved object state and passed to the dispatcher and specialist agents for bounded tool execution.

3.3 Specialist Agents and Tool Execution

After planning and targeted parsing, each slide-level editing task is paired with the JSON representation of its target slide. The dispatcher inspects the parsed shape inventory, selects the shapes relevant to the task, and routes each selected shape to a specialist agent according to its object type and intended edit. The specialist then receives the resulting shape-level task and target shape metadata and invokes the corresponding COM tool.

3.4 Dual-modal validation

After executing an edit, EDITPPT validates the result through dual-modal validation, separating instruction-level verification from visual-quality assessment for more robust evaluation. It re-parses the edited deck and compares it with the original state and edit plan to ensure that the intended slide object was correctly modified while unrelated elements remain unchanged; additionally, the edited slide is rendered and checked by a VLM-based validator to detect visual design issues such as overlap, clipping, or unintended layout shifts.

4 DECKEDIT-BENCH

To evaluate realistic slide editing over existing presentations, we introduce DECKEDIT-BENCH, a benchmark of 28 human-authored PowerPoint decks, 582 slides, and 183 natural-language editing prompts. Unlike slide generation or isolated slide modification, practical presentation editing requires a system to locate the correct target inside an existing deck, apply the requested change, and preserve unrelated content, formatting, and layout. The benchmark therefore keeps source decks as native `.pptx` files with heterogeneous layouts, grouped objects, embedded images, tables, charts, shapes, backgrounds, and slide-level properties.

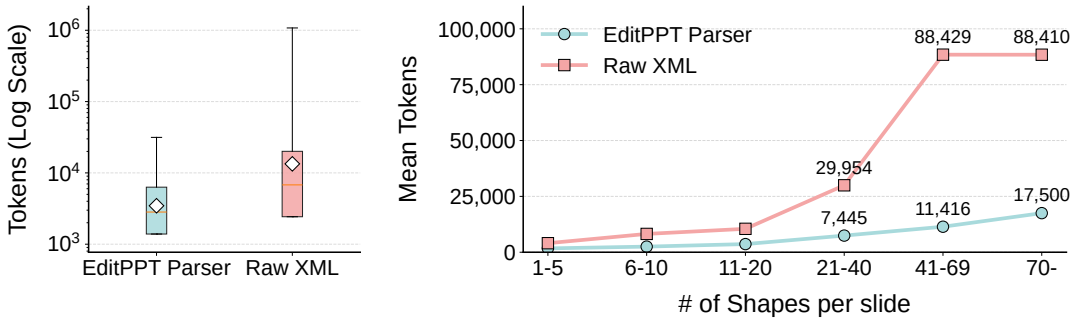

Figure 4: Token efficiency of the EditPPT XML parser. (a) The parser reduces the token footprint and removes the extreme long tail of raw XML, which is used in Claude code and Claude PPT. (b) Compression becomes more effective as slide complexity increases, using shape count as a proxy for complexity.

4.1 Deck Collection and Length Tiers

The deck corpus is curated to reflect presentations that people actually use, drawing from research, teaching, and professional materials in both Korean and English. Most decks were directly written and used by human authors, while a small number were first drafted with generation assistance and then manually edited into realistic, usable presentations.

We stratify the corpus by deck length into three tiers: short decks with at most 10 slides, medium decks with 11 to 30 slides, and long decks with more than 30 slides. The tiers are defined by length rather than difficulty, so short decks may still contain dense, visually rich slides, while long decks stress target selection, preservation, and execution over larger presentation contexts. Detailed tier statistics are provided in Appendix B.1.

4.2 Editing Prompt Design

For the 28 decks, we create 183 natural-language editing prompts that reflect user-facing presentation editing requests. The prompts do not expose internal shape identifiers, XML paths, object indices, or implementation-specific handles; therefore, a system must infer the target slide, target object, and intended operation from the instruction and the presentation content.

The prompts cover both local and presentation-scale edits, ranging from specified single-slide modifications to repeated-target edits across a deck. We vary both target scope and instruction complexity, yielding four prompt categories: Explicit-Simple, Explicit-Compound, Pattern-Simple, and Pattern-Compound. Explicit prompts specify the relevant slide directly, while pattern prompts require identifying recurring targets across multiple slides. Simple prompts involve a single target or operation, whereas compound prompts require multiple coordinated edits within a single instruction.

Detailed deck metadata, prompt templates, action labels, target categories, and capability groupings are provided in Appendix B.

5 Experiments

We evaluate reliable slide editing along three criteria: modification accuracy, preservation fidelity, and robustness to deck length. Accuracy and preservation are measured at both the slide level, where we evaluate whether the system edits only the requested slides, and the object level, where we evaluate whether the requested edit is fulfilled without changing out-of-scope objects. Length robustness is measured by reporting the same metrics across short, medium, and long decks.

5.1 Experimental Setup

All systems are evaluated on the full DECKEDITBENCH benchmark, consisting of 183 deck-prompt instances across 28 decks. Each instance starts from a fresh copy of the source deck, so edits do not accumulate across prompts. A run is considered executable if the system produces an edited `.pptx` file without runtime failure or unrecoverable file corruption. We record execution status, output files, logs, API cost, and runtime for each run.

Baselines. We compare our method with three baselines. **Talk-to-Your-Slides** is a COM-based code-generation baseline for natural-language presentation editing. **PPTPilot** is the editing agent from PPTArena, which alternates between `python-pptx` operations and direct XML manipulation depending on the required edit. **Claude Code + PPTX Skill** is a strong general-purpose coding-agent baseline equipped with PowerPoint

Metric	Claude Sonnet 4.6	Claude Opus 4.7	EditPPT
Cost (full deck)			
Avg. cost	$14.87	$43.18	**$4.91**
Cost SD	$11.08	$24.04	**$0.27**
Avg. cost / slide	$0.304	$0.881	**$0.100**
Cost / slide SD	$0.226	$0.491	**$0.006**
End-to-End Runtime (full deck)			
Avg. runtime	**34.3 min**	48.5 min	40.2 min
Runtime SD	9.2 min	24.3 min	**2.1 min**
Avg. runtime / slide	**42.0s**	59.4s	49.2s
Runtime / slide SD	11.3s	29.7s	**2.6s**

Table 4: Repeated-trial variability under identical conditions. Each system was run seven times on the same 49-page deck with the identical "Translate to English" instruction. **Bold** values indicate the best result for each metric, while underlined values indicate the second-best result.

editing capability. All systems receive the same source deck and user instruction and are required to produce an edited `.pptx` file.

Configuration. We use OpenAI GPT-4.1 as the backbone language model for all text-only components of our framework, including the planner, dispatcher, specialist agents, structural validator, and recovery planner (OpenAI, 2025). For image-related processing, we use Gemini 2.5 Flash-Lite to generate image captions during parsing and Gemini 2.5 Pro to judge visual layout defects during validation (Google, 2026). Talk-to-Your-Slides and PPTPilot are also evaluated with GPT-4.1 under their default tool configurations (Jung et al., 2026; Ofengenden et al., 2025). Claude Code + PPTX Skill is run with Claude Sonnet 4.6 and the official PPTX Skill (Anthropic, 2026b,c). For the repeated-trial variability study, we separately evaluate Claude Sonnet 4.6 and Claude Opus 4.7 on the same long-deck translation task (Anthropic, 2026b,a).

5.2 Evaluation

We evaluate edited decks in two stages to separate slide selection from object-level editing quality. All LLM and VLM judges use GPT-5.5 (OpenAI, 2026), OpenAI's latest frontier model.

Stage 1. Slide-level targeting. We first identify which slides were edited. Each slide is labeled as `correctly_targeted` if it was requested and edited, `missing_target` if it was requested but unchanged, and `wrong_target` if it was edited despite being outside the requested scope. Slides that are neither requested nor edited are treated as correctly unchanged and excluded from precision, recall, and F1. This stage measures slide-level modification accuracy and slide-level preservation by LLM Judge.

Stage 2. Object-level instruction following. On correctly targeted slides, we evaluate whether the requested edit was fulfilled and whether unrelated objects were preserved. The target edit is labeled as `success`, `partial`, or `fail`, where `partial` indicates incomplete fulfillment or minor formatting deviations. We separately mark a `preservation_error` when any object outside the requested edit scope is unintentionally modified. We also render the before-and-after decks and use a VLM judge to assess visual defects and layout changes.

Metrics. We report slide-targeting F1, Instruction Following, Object Preservation Rate, and defect-complement visual quality, with the main metrics broken down by deck length.

6 Results

Overall performance. Table 2 summarizes the overall and length-wise results on DECKEDITBENCH. Our method completes 99.5% of editing instances and achieves 88.7% Slide F1, 82.5% Instruction Following, and 91.5% object-level preservation overall. These results show that the system not only produces valid edited decks reliably, but also selects the intended slides, applies the requested edits, and preserves out-of-scope objects with high accuracy. Table 3 further decomposes these scores into slide-level targeting and object-level instruction following. Although Claude Code leads on both stages, our framework outperforms the other native-edit baselines.

Robustness to deck length. As shown in Table 2, performance remains stable across deck lengths. Long decks achieve 90.5% Slide F1, 86.6% Instruction Following, and 91.7% object-level preservation, improving over the corresponding short-deck instruction-following score while staying close to the short-deck preservation score. PPTPilot and Talk-to-Your-Slides, by contrast, show clear instruction-following degradation from short to long decks, suggesting constrained native execution helps reduce length-induced failure cascades.

Input representation	Tokens	Est. cost	Efficiency
EditPPT Parser	**8.45M**	**$16.89**	**4.3×**
Raw XML, full scope	36.48M	$72.97	1.0×

Table 5: Token and cost comparison between the EditPPT parser representation and raw XML over 2,313 parsed slides. Full scope includes both slide XML files and their associated relationship files. Estimated cost assumes GPT-4.1 input pricing without caching. Efficiency is measured relative to raw XML, where higher is better.

Metric	EditPPT	EditPPT + Single Agent	UnifiedVLM
Performance			
Exec. Rate	99.5%	**100.0%**	**100.0%**
Slide F1	88.7%	**90.6%**	82.6%
Instr. Follow.	82.5%	**86.1%**	44.7%
Preserv.	**91.5%**	**91.5%**	59.6%
Efficiency			
Avg. Cost ($\times 10^{-2}$\$/slide)	**7.74**	10.05	9.71
Avg. Time (s/slide)	**40.8**	46.1	70.9

Table 6: Ablation study comparing the multi-agent system with a single-agent variant and a unified VLM baseline using the same tool set.

Repeatability. Table 4 evaluates repeated runs on the same long-deck translation task. EDITPPT has the lowest average cost and the smallest cost variation, with $4.91 average cost and $0.27 standard deviation, compared with $14.87±$11.08 for Claude Sonnet 4.6 and $43.18±$24.04 for Claude Opus 4.7. It also has the lowest runtime variation, 2.1 minutes across trials, indicating more predictable resource usage under repeated execution.

Additional analyses. Appendix C reports additional result analyses, including modification and preservation breakdowns, qualitative examples, and validation details, while Appendix D reports parser token efficiency.

6.1 Ablation Study

We ablate two core design choices of our framework, the multi-agent system and the dual-modal validator. Detailed results are reported in Table 6.

Multi-agent dispatch. Replacing the type-specialized dispatcher with a single general-purpose agent yields modest accuracy gains, but at substantial efficiency cost: per-slide cost rises by 26.6% and runtime by 10.0%. The single agent must reason over the full tool inventory at every step, inflating prompt context and reasoning length. Object Preservation is unchanged (91.5%), indicating that locality is governed by tool design rather than agent decomposition. Multi-agent dispatch therefore offers a favorable accuracy–efficiency trade-off, with the efficiency advantage compounding on long, content-dense decks.

Dual-modal validation. Replacing the dual-modal validator with a single VLM judge — provided with both the rendered slide and the parsed structural state — causes severe regressions across every quality metric, with Instruction Following collapsing by over 37 points and Object Preservation by nearly 32 points. We attribute this collapse to two likely factors. First, cognitive overload, as the VLM must judge structure and visuals simultaneously. Second, modality grounding bias, as it tends to trust visual cues over the structural state. These findings suggest that relying on a VLM alone to validate slide edits is risky, and that separating structural verification via re-parsing from visual layout checking is a precondition for reliable validation.

The specialist architecture operationalizes the action-space reduction introduced in Section 3.1 by decomposing slide-level plans into type-compatible shape-level tool calls. This reduces cross-object tool misuse before validation and recovery.

7 Conclusion

EDITPPT reframes presentation editing as constrained tool selection over native PowerPoint objects, rather than as open-ended code generation or raw file-structure manipulation. This design aligns parsing, execution, and validation around the same COM-resolved object state, allowing agents to perform localized edits while preserving unrelated slide content. Our parser removes the redundancy of raw OOXML and yields more efficient, repeatable execution across runs, while a dual-modal validator separates structural verification from visual layout assessment, enabling more robust judgment along each modality. Through DECKEDIT-BENCH, we evaluate this setting on human-authored decks across length tiers and show that EDITPPT maintains strong execution reliability, slide targeting, instruction following, and object preservation, including on long decks where conventional baselines degrade. These findings suggest that robust document editing agents should expose less arbitrary executable freedom to LLMs and instead provide typed interfaces that match the native structure of the target application.

Limitations

EDITPPT currently relies on the PowerPoint COM interface, which is tied to the desktop PowerPoint runtime and is effectively single-threaded for many operations. This limits straightforward parallelism and makes deployment dependent on environments where PowerPoint automation is available. The framework is also optimized for concrete editing requests over existing objects, so highly abstract instructions that require extensive design reasoning, content rewriting, or global narrative restructuring may still require stronger planning modules or human review. Finally, while our validators reduce silent failures, visual judging remains imperfect, especially for subtle aesthetic regressions or domain-specific formatting conventions.

Ethical Considerations

This work aims to improve the reliability of presentation editing tools and can reduce repetitive manual labor in document workflows. Because the system modifies user-authored decks, deployments should preserve source files, expose edit logs, and avoid overwriting user content without confirmation. The benchmark includes human-authored presentation materials. All decks in DECKEDITBENCH were contributed by the authors or by colleagues who were fully informed of the research purpose and gave consent for their decks to be used in a publicly released benchmark. We removed or anonymized any sensitive or personally identifying content prior to release, respected source licenses, and avoided exposing private organizational content. More broadly, automated editing can be used to alter persuasive or educational material at scale, which makes provenance, review, and user control important safeguards.

Use of AI Assistants We used AI coding assistants for parts of the implementation and AI writing assistants for language polishing of this manuscript. All technical claims, experimental results, and conclusions were independently verified by the authors.

References

Anthropic. 2026a. Claude Opus 4.7. https://www.anthropic.com/claude/opus. Accessed: 2026-05-25.

Anthropic. 2026b. Claude Sonnet 4.6. https://www.anthropic.com/claude/sonnet. Accessed: 2026-05-25.

Anthropic. 2026c. PPTX Skill. https://github.com/anthropics/skills/blob/main/skills/pptx/SKILL.md. Accessed: 2026-05-25.

Anthropic. 2026d. Using Claude for PowerPoint. https://support.claude.com/ko/articles/13521390-powerpoint%EC%9A%A9-claude-%EC%82%AC%EC%9A%A9. Accessed: May 23, 2026.

Jiaxin Ge, Zora Zhiruo Wang, Xuhui Zhou, Yi-Hao Peng, Sanjay Subramanian, Qinyue Tan, Maarten Sap, Alane Suhr, Daniel Fried, Graham Neubig, and Trevor Darrell. 2025. AutoPresent: Designing structured visuals from scratch. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Google. 2026. Gemini API Models. https://ai.google.dev/gemini-api/docs/models. Accessed: 2026-05-25.

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

Daesik Jang, Morgan Lindsay Heisler, Linzi Xing, Yifei Li, Edward Wang, Ying Xiong, Yong Zhang, and Zhenan Fan. 2026. Deckbench: Benchmarking multi-agent frameworks for academic slide generation and editing. *Preprint*, arXiv:2602.13318.

Kyudan Jung, Hojun Cho, Jooyeol Yun, Soyoung Yang, Jaehyeok Jang, and Jaegul Choo. 2026. Talk to your slides: High-efficiency slide editing via language-driven structured data manipulation. *Preprint*, arXiv:2505.11604.

Hyeonsu Kang, Emily Bao, and Anjan Goswami. 2025. Vlm-slideeval: Evaluating vlms on structured comprehension and perturbation sensitivity in ppt. *Preprint*, arXiv:2510.22045.

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 *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. OmniParser for pure vision based gui agent. *Preprint*, arXiv:2408.00203.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative

refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Michael Ofengenden, Yunze Man, Ziqi Pang, and Yu-Xiong Wang. 2025. Pptarena: A benchmark for agentic powerpoint editing. *Preprint*, arXiv:2512.03042.

OpenAI. 2025. GPT-4.1 Model. https://developers.openai.com/api/docs/models/gpt-4.1. Accessed: 2026-05-25.

OpenAI. 2026. Gpt-5.5 technical report. https://www.openai.com. Accessed: 2026.

Yiming Pan, Chengwei Hu, Xuancheng Huang, Can Huang, Mingming Zhao, Yuean Bi, Xiaohan Zhang, Aohan Zeng, and Linmei Hu. 2026. Aeslides: Incentivizing aesthetic layout in llm-based slide generation via verifiable rewards. *Preprint*, arXiv:2604.22840.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *Preprint*, arXiv:2305.15334.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Rui Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations (ICLR)*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhengliang Shi, Yiqun Chen, Haitao Li, Weiwei Sun, Shiyu Ni, Yougang Lyu, Run-Ze Fan, Bowen Jin, Yixuan Weng, Minjun Zhu, Qiujie Xie, Xinyu Guo, Qu Yang, Jiayi Wu, Jujia Zhao, Xiaqiang Tang, Xinbei Ma, Cunxiang Wang, Jiaxin Mao, and 7 others. 2025. Deep research: A systematic survey. *Preprint*, arXiv:2512.02038.

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

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

Wei Wu, Ziyang Xu, Zeyu Zhang, Yang Zhao, and Hao Tang. 2026. Presentagent-2: Towards generalist multimodal presentation agents. *Preprint*, arXiv:2605.11363.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

Yuheng Yang, Wenjia Jiang, Yang Wang, Yiwei Wang, and Chi Zhang. 2025. Auto-Slides: An interactive multi-agent system for creating and customizing research presentations. *Preprint*, arXiv:2509.11062.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Wenzheng Zeng, Mingyu Ouyang, Langyuan Cui, and Hwee Tou Ng. 2025. SlideTailor: Personalized presentation slide generation for scientific papers. *Preprint*, arXiv:2512.20292.

Hao Zheng, Xinyan Guan, Hao Kong, Wenkai Zhang, Jia Zheng, Weixiang Zhou, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2025. PPTAgent: Generating and evaluating presentations beyond text-to-slides. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 14402–14418, Suzhou, China. Association for Computational Linguistics.

Hao Zheng, Guozhao Mo, Xinru Yan, Qianhao Yuan, Wenkai Zhang, Xuanang Chen, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. 2026. Deeppresenter: Environment-grounded reflection for agentic presentation generation. *Preprint*, arXiv:2602.22839.

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

A Related Works (continued)

Tool-augmented and multi-agent LLM systems. Tool-augmented LLMs extend language models by enabling them to invoke external tools, APIs, and executable functions during reasoning and task execution (Schick et al., 2023; Yao et al., 2023; Qin et al., 2024; Patil et al., 2023). Multi-agent frameworks further decompose complex tasks into specialized roles, improving modularity across planning, execution, and verification (Wu et al., 2023). Building on this paradigm, our solution adapts tool-augmented execution to native presentation editing through constrained specialist agents, each restricted to a narrow, type-consistent subset of deterministic shape-level tools. This restriction reduces the effective action space exposed to each agent and mitigates cross-object tool misuse, thereby improving the reliability of structured slide editing.

LLM- and VLM-based validation. LLM-as-judge methods have been widely used to evaluate open-ended model outputs, including instruction following, response quality, and task completion (Zheng et al., 2023; Liu et al., 2023). Recent work has also extended this paradigm to multimodal settings, where VLMs assess visual outputs or detect layout-level defects. In slide editing, such evaluators are useful because success requires both semantic instruction fulfillment and visual layout integrity, but most judge-based protocols are used primarily as post-hoc evaluation mechanisms. In contrast, EDITPPT incorporates modality-decoupled validation into the editing loop: a structural validator checks instruction fulfillment and unintended object-level changes from parsed slide states, while a visual validator detects newly introduced layout defects from rendered slides. These validators provide recovery signals for replanning and repair, rather than serving only as final evaluators.

B Additional Benchmark Details

DECKEDIT-BENCH includes prompt metadata for target scope, action type, object category, and instruction complexity. Explicit prompts name a target slide, while pattern prompts require locating recurring or conditional targets, and compound prompts require multiple coordinated object-level edits within a single instruction.

Tier	Slide range	# Decks	# Slides	# Prompts	# Shapes	Shapes / Slide
Short	$\leq$10	9	46	54	505	10.98
Medium	11 to 30	12	241	81	1,758	7.29
Long	>30	7	295	48	4,494	15.23
Total	All	**28**	**582**	**183**	**6,757**	**11.61**

Table 7: Statistics of DECKEDIT-BENCH. Tiers are defined by deck length, but all tiers contain structurally rich slides with native PowerPoint objects.

B.1 DECKEDIT-BENCH Statistics

Table 7 reports the detailed corpus statistics by deck-length tier. Tiers are defined by slide count rather than by intrinsic difficulty or slide simplicity, and all tiers contain native PowerPoint objects and heterogeneous layouts.

B.2 Examples from DECKEDIT-BENCH

Figure 5 shows representative examples from DECKEDIT-BENCH. Each example is paired with its pristine target slide.

C Additional Result Analyses

C.1 Modification Breakdown

Remaining failures are concentrated in target selection rather than local tool execution. Pattern-Compound tasks are the most difficult because they require conditional slide selection across a deck, while ADD operations are the easiest and reach near-perfect instruction following once the target is found.

C.2 Detailed Preservation Results

The detailed preservation breakdown shows that EDITPPT's strongest advantage is object preservation: it reaches 91.5% overall preservation and remains above 90% in every length tier. Its remaining preservation errors mainly arise from pattern-based instructions, where the system must infer recurring targets across multiple slides, rather than from explicit single-slide edits.

C.3 Qualitative Comparison

Figure 3 compares representative presentation-editing strategies. The qualitative examples show that open-ended code or structure-level editing can satisfy the requested operation while still introducing unintended changes to neighboring content, whereas EDITPPT localizes execution through typed shape-level tools and therefore better preserves unrelated slide elements.

System	Breakdown	Category	Inst.	Slide-level	Object-level
				Wrong Slide Edits ↓	Object Preserv. ↑
PPTPilot	Task type	Explicit-Single	13	61.8%	76.9%
		Explicit-Compound	39	63.4%	58.8%
		Pattern-Single	77	8.0%	61.1%
		Pattern-Compound	18	15.1%	87.6%
	Action type	ADD	15	15.2%	77.7%
		REPLACE	105	14.6%	57.6%
		DELETE	15	3.8%	73.2%
		SLIDE	7	0.0%	58.3%
Talk-to-Your-Slides	Task type	Explicit-Single	17	0.0%	76.5%
		Explicit-Compound	57	0.0%	60.0%
		Pattern-Single	65	10.7%	73.6%
		Pattern-Compound	21	15.0%	81.5%
	Action type	ADD	12	0.0%	89.4%
		REPLACE	121	11.6%	75.5%
		DELETE	13	33.3%	65.0%
		SLIDE	9	0.0%	56.2%
Claude Code + PPTX Skill	Task type	Explicit-Single	17	0.0%	93.8%
		Explicit-Compound	57	0.0%	92.6%
		Pattern-Single	87	0.4%	76.2%
		Pattern-Compound	22	1.5%	88.7%
	Action type	ADD	18	0.0%	84.1%
		REPLACE	132	0.5%	73.8%
		DELETE	17	0.9%	79.6%
		SLIDE	11	0.0%	100.0%
EditPPT	Task type	Explicit-Single	17	0.0%	93.8%
		Explicit-Compound	57	0.0%	84.0%
		Pattern-Single	86	6.2%	89.8%
		Pattern-Compound	22	15.1%	93.7%
	Action type	ADD	18	0.0%	87.5%
		REPLACE	131	7.5%	89.9%
		DELETE	17	20.1%	94.5%
		SLIDE	11	6.7%	92.4%

Table 8: Detailed preservation fidelity analysis by system, task type, and action type. Slide-level preservation is measured by Wrong Slide Edits, the proportion of slide-level targeting outcomes labeled as `wrong_target`, indicating edits to slides outside the requested scope. Object-level preservation is measured by Object Preserv., the proportion of correctly targeted slides without `preservation_error`.

C.4 Validation and Recovery Details

A successful COM call does not necessarily imply a successful edit: the operation may execute without error while failing to satisfy the user instruction or introducing a layout defect. This mirrors recent findings that external feedback and critic-in-the-loop refinement can improve agent reliability, but that VLM-based slide assessment remains imperfect and benefits from calibration (Madaan et al., 2023; Gou et al., 2024; Kang et al., 2025). EDITPPT therefore validates each edited slide with two complementary modules. A structural validator checks instruction fulfillment from the before/after parsed slide state, while a visual validator inspects the rendered slide for layout errors introduced during editing.

Structural validation. The structural validator receives the user instruction, the slide-level task, and the difference between the parsed slide JSON before and after editing. It checks whether the intended edit was applied to the correct target without unintended changes to unrelated objects or parsed properties. For severe failures, such as incorrect target selection or a missing edit, EDITPPT restores the slide-level checkpoint and retries the specialist with validator feedback, following verbal self-reflection and iterative feedback-based refinement (Shinn et al., 2023; Madaan et al., 2023; Gou et al., 2024). For partially correct edits, the validator instead produces an incremental correction plan so that the specialist can complete only the remaining local modification.

Stage	Input	Output
Planning	User instruction	Slide-level tasks
Parsing	Target slides	Shape JSON states
Dispatch	Tasks and shapes	Specialist calls
Execution	Typed arguments	COM edits
Validation	Before/after states	Accept or recover

Table 9: The EDITPPT pipeline keeps each step grounded in PowerPoint-resolved objects, from planning to validation.

COM gap	Reconstruction	Editing benefit
Mixed text style	Run-level spans	Preserves formatting
Paragraph mismatch	Logical segments	Targets bullets and subtitles
Merged table cells	Geometric grouping	Edits logical cells
Image-only content	Caption metadata	Grounds visual references

Table 10: Parser augmentations used to convert raw COM state into editable units that better match natural-language slide-editing instructions.

D Parser Efficiency

D.1 XML-Parser Token Efficiency

To quantify the benefit of our parser, we compare the token footprint of EditPPT parser output against raw XML input. As shown in Table 5, the parser reduces total input tokens from 36.48M to 8.45M, a 76.8% reduction. This also lowers the estimated input cost from \$72.97 to \$16.89, corresponding to a 4.32$\times$ reduction.

Figure 4(a) further shows that the reduction is not limited to aggregate cost. Raw XML exhibits a heavy-tailed token distribution, reaching up to 1.1M tokens for a single slide, while the parser output is capped at 31K tokens in this set. Thus, the parser improves both efficiency and robustness by reducing average input size as well as extreme context-length outliers.

We also find that the parser becomes more effective as slide complexity increases. Figure 4(b) plots compression efficiency against the number of shapes per slide, using shape count as a proxy for slide complexity. Slides with more shapes tend to contain more repeated style, coordinate, relationship, and object-level metadata in raw XML. The parser removes much of this redundant low-level structure and represents the slide in a compact semantic form, leading to larger relative token savings on more complex slides.

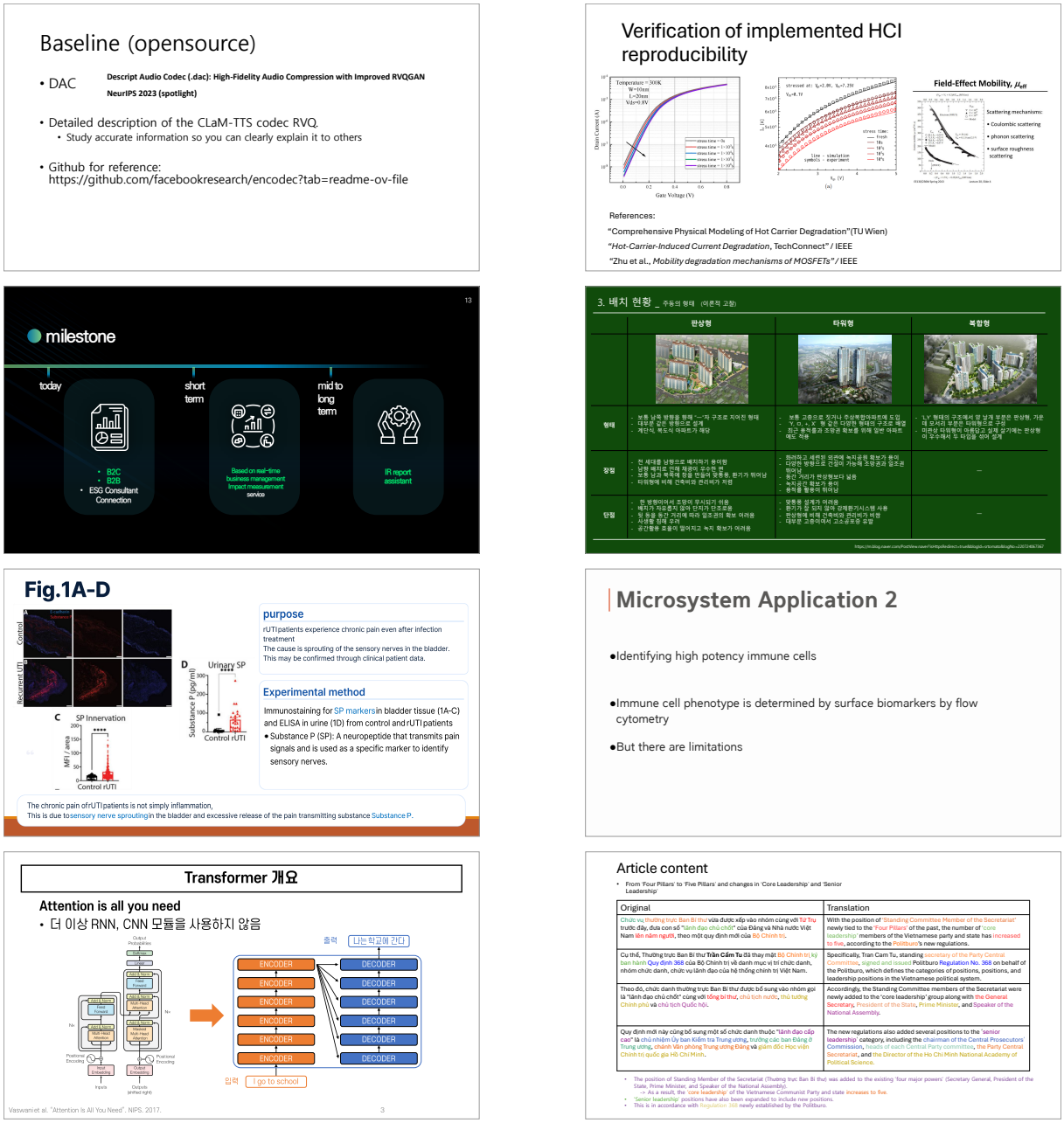

Figure 5: Representative editing instructions from DECKEDIT-BENCH, each paired with its pristine target slide. Explicit prompts directly specify the target slide or object, whereas pattern prompts require the system to infer recurring targets across the deck. Simple prompts involve a single operation, while compound prompts require multiple coordinated edits. Pattern examples show one representative slide containing the recurring target.

E Technical Implementation Details

Parser. The Parser converts all objects on each slide, including text, tables, charts, images, and shapes, into a JSON structure via the PowerPoint COM API. To avoid redundant work, it caches per-slide parse results in `database[page_number]`. When a retry is needed after an edit, the parser bypasses the cache with `force=True` so that the latest slide state is parsed, and edit history is stored as snapshots in `edit_history`.

JSON structure:

```json
{
  "Objects_Detail": [
    {
      "Shape_Id": 2,
      "Name": "Title 1",
      "Type": "TextBox",
      "More_detail": {
        "TextFrame": {
          "Runs": [
            { "Text": "Hello", "Font": { "Name": "Malgun Gothic", "Size": 24, "Bold": true } }
          ]
        }
      }
    },
    {
      "Shape_Id": 5,
      "Name": "Table 1",
      "Type": "Table",
      "More_detail": {
        "Table": {
          "Cells": {
            "1,1": { "Text": "Item", "Runs": [...] },
            "1,2": { "Text": "Value", "Runs": [...] }
          }
        }
      }
    }
  ]
}
```

Table 11: Planner Modes

Mode	Description	Example Usage
explicit	Specifies individual tasks for each slide	"Change the title on page 1 to red, and highlight the table header on slide 4 in blue."
pattern	Template-based repetitive tasks (page number substituted via `{i}`)	"Change the body font to Arial on all slides"

Planner.

Task plan JSON:

```json
{
  "understanding": "The user wants to change the title on slide 1 to red and highlight the table header on slide 4 in blue.",
  "tasks": [
    {
      "page_number": 1,
      "description": "Change the font color of the title text to red",
      "action": "change font color to red",
      "target": "title",
      "contents": { "color": "#FF0000" }
    },
    {
      "page_number": 4,
      "description": "Change the font color of the table header to blue",
      "action": "change table header cell to blue",
      "target": "table header",
      "contents": { "color": "#0000FF" }
    }
  ]
}
```

Dispatcher. DispatcherAgent routes tasks to specialized agents. When parsed slide data is available, the LLM performs multi-dispatch by classifying relevant shapes and grouping them by agent type. Slide-level operations such as adding or deleting slides are routed directly to the slide agent, while classification failures fall back to the `text_style` agent.

```
FALLBACK_AGENT = "text_style"

class DispatcherAgent:
    def dispatch(self, task, slide_objects=None):
        if not slide_objects:
            return self._dispatch_single(task)
        shape_summary = [
            {"Shape_Id": obj["Shape_Id"], "Name": obj["Name"], "Type": obj["Type"]}
            for obj in slide_objects
        ]
        response = call_llm(
            model=self.model,
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user",   "content": user_prompt},
            ],
        )
```

Table 12: Agent Registry

Agent	Description	No. of Tools	Tool List
text_style	Text content and formatting edits	7	set_text_style, edit_text_insert, edit_text_delete, edit_text_replace, edit_text_rewrite, set_paragraph_alignment, manage_bullet_points
table	Table structure and cell editing	3	cell_text_style, replace_table_text, table_layout_style
chart	Chart data and style modification	5	update_chart_categories, update_chart_series, update_chart_structure, update_chart_axes, update_chart_colors
shape_layout	Shape placement, creation, and image insertion	13	adjust_layout, distribute_shapes, align_shapes, create_textbox, create_placeholder, create_shape, delete_shape, duplicate_shape, duplicate_shape_within_slide, apply_visual_style, apply_gradient_fill, insert_image, edit_image
slide	Slide-level operations	5	add_slide, delete_slide, duplicate_slide, set_slide_transition, set_slide_background

Specialist Agents : 5 types.

```
@dataclass
class AgentSpec:
    agent_type: str
    tool_names: List[str]
    system_prompt_builder: Callable[[str], str]
    description: str

AGENT_REGISTRY: dict[str, AgentSpec] = {}

register_agent(AgentSpec(
    agent_type="text_style",
    tool_names=[
        "set_text_style", "edit_text_insert", "edit_text_delete",
        "edit_text_replace", "edit_text_rewrite",
        "set_paragraph_alignment", "manage_bullet_points",
    ],
    system_prompt_builder=create_text_style_agent_system_prompt,
    description="Text content and formatting ...",
))
```

BaseEditAgent : retry + rollback. BaseEditAgent executes each edit as a bounded retry-and-rollback loop. It first parses the slide with `parser.process()`, using `force=True` on retries to bypass cached state, and filters the parsed JSON to the `shape_ids` selected by the dispatcher. The agent then constructs a specialist prompt from the system prompt, target slide JSON, and user request, calls the LLM to select a tool and arguments through function calling, and injects required runtime fields such

as `slide_json` and `agent_request` according to `ToolMeta`. After COM execution, text tools call `clamp_text_to_slide()` and other tools call `clamp_shapes_to_slide()`. The result is checked by the text validator and, when enabled, the vision validator, after which the runtime either saves the edit, restores a checkpoint and retries, or keeps the current state and applies an incremental correction.

Tools : 33 tools.

Text Editing (7 tools): set_text_style, edit_text_insert, edit_text_delete, edit_text_replace, edit_text_rewrite, set_paragraph_alignment, manage_bullet_points

Table Editing (3 tools): cell_text_style, replace_table_text, table_layout_style

Chart Editing (5 tools): update_chart_categories, update_chart_series, update_chart_structure, update_chart_axes, update_chart_colors

Shape/Layout (13 tools): adjust_layout, distribute_shapes, align_shapes, create_textbox, create_placeholder, create_shape, delete_shape, duplicate_shape, duplicate_shape_within_slide, apply_visual_style, apply_gradient_fill, insert_image, edit_image

Slide Management (5 tools): add_slide, delete_slide, duplicate_slide, set_slide_transition, set_slide_background

```
@dataclass
class ToolMeta:
    needs_slide_json: bool = False
    needs_agent_request: bool = False
    needs_container: bool = False
    cleanup_false_args: list | None = None

meta = TOOL_METADATA.get(function_name)
if meta:
    if meta.needs_slide_json:
        function_args["slide_json"] = contents
    if meta.needs_agent_request:
        function_args["agent_request"] = agent_request
```

Validator.

Text Validator (required): Validates editing results by comparing old_parse and new_parse using an LLM.

```
def update_after_edit(self, text_validation, model, page_number,
                      description, action, detailed_contents, used_tools):
    old_parse = self.database[page_number]
    new_parse = parse_active_slide_objects(
        page_number, self.container.prs, self.container.ppt_app
    )
    response = call_llm(
        model=model,
        messages=[
            {"role": "system", "content": create_text_validator_agent_system_prompt(...)},
            {"role": "user",   "content": create_text_validator_agent_user_prompt(
                old_parse, new_parse, used_tools
            )},
        ],
    )
    return (valid, reason, strategy, new_parse)
```

Vision Validator (in development): Exports the slide as a PNG and performs visual validation using Gemini 2.5 Pro.

```
class VisionValidatorAgent:
    @classmethod
    def create(cls, activate_valid=False, container=None, model=None):
        return cls(activate_valid, container, model)

    def process(self, page_number, agent_request, parsed_contents, used_tools):
        slide.Export(str(screenshot_path), "PNG")
        response = call_llm(model=self.model, messages=[...])
        return (valid, reason)
```

Table 13: Bounds Clamping Targets

Target Tools	Clamping Function	Behavior
Text tools (edit_text_rewrite, edit_text_replace, edit_text_insert, replace_table_text, set_text_style, manage_bullet_points)	`clamp_text_to_slide()`	Proportionally reduces font size at the Run level to fit text within the shape (up to 5 passes)
Other tools	`clamp_shapes_to_slide()`	Repositions or resizes shapes that have moved outside the slide boundary

Bounds Clamping.

```
_TEXT_CLAMP_TOOLS = {
    "edit_text_rewrite", "edit_text_replace", "edit_text_insert",
    "replace_table_text", "set_text_style", "manage_bullet_points",
}

def _execute_tool(self, name, args):
    result = FUNCTION_MAP[name](**args)
    if name in _TEXT_CLAMP_TOOLS:
        clamp_text_to_slide(prs, slide_number, shape_id)
    else:
        clamp_shapes_to_slide(prs, slide_number)
```

Table 14: Pipeline Summary

Step	Component	Role	On Failure
1	Planner	Natural language → structured task list	Up to 3 LLM retries
2	Parser	Slide → JSON database	Invalidate cache and re-parse
3	Dispatcher	Task → specialized agent routing	Fallback to text_style
4	Specialist Agent	LLM Tool Calling → COM execution	Rollback + up to 3 retries
5	Validator	Validate editing results (text/vision)	Strategy-based branching (incremental/rollback)
6	Bounds Clamping	Prevent shapes from exceeding slide bounds	Adaptive font reduction / position adjustment

Pipeline Summary.

E.1 In-House Algorithms

Run-Level Parsing.

Problem: The PowerPoint COM `TextFrame2.TextRange.Runs` API does not return Run boundaries accurately in a dynamic dispatch environment.

Solution: Character-level font snapshot comparison

Step 1 : snap().

```
def snap(font):
    if font is None:
        return (None, 0.0, False, False, False, None, False, False, False)
    return (
        safe(font, "Name"),
        round(float(safe(font, "Size", 0)), 1),
        bool(safe(font, "Bold", 0)),
        bool(safe(font, "Italic", 0)),
        bool(safe(font, "Underline", 0)),
        rgb_of(font),
        bool(safe(font, "Strikethrough", 0)),
        bool(safe(font, "Subscript", 0)),
        bool(safe(font, "Superscript", 0)),
    )
```

Step 2 : parse_text_frame_debug().

```
def parse_text_frame_debug(text_frame):
    tr = text_frame.TextRange
    full = tr.Text
    runs = []
    n = safe(tr, "Length", len(full))
    cur_idx = 1
    cur_snap = snap(safe(tr.Characters(cur_idx, 1), "Font"))
```

```
    for i in range(2, n + 1):
        nxt_snap = snap(safe(tr.Characters(i, 1), "Font"))
        if nxt_snap != cur_snap:
            seg_len = i - cur_idx
            runs.append(make_run_dict(tr.Characters(cur_idx, seg_len)))
            cur_idx = i
            cur_snap = nxt_snap
    runs.append(make_run_dict(tr.Characters(cur_idx, n - cur_idx + 1)))
    cp_offset = 0
    for run in runs:
        run["Run_Start_Index"] = cp_offset
        cp_offset += len(run.get("Text", ""))
    return {"Has Text": True, "Text": full, "Runs": runs,
            "Paragraphs": parse_paragraph_bullets(text_frame)}
```

Step 3 : make_run_dict().

```
def make_run_dict(text_range_segment):
    text = safe(text_range_segment, "Text", "")
    run = {"Text": text}
    f = safe(text_range_segment, "Font")
    if not f:
        return run
    font_dict = {}
    name = safe(f, "Name")
    if name is not None:
        font_dict["Name"] = name
    size = safe(f, "Size")
    if size is not None:
        font_dict["Size"] = size
    if safe(f, "Bold", 0):          font_dict["Bold"] = True
    if safe(f, "Italic", 0):        font_dict["Italic"] = True
    if safe(f, "Underline", 0):     font_dict["Underline"] = True
    if safe(f, "Strikethrough", 0): font_dict["Strikethrough"] = True
    if safe(f, "Subscript", 0):     font_dict["Subscript"] = True
    if safe(f, "Superscript", 0):   font_dict["Superscript"] = True
    rgb = rgb_of(f)
    if rgb is not None:
        font_dict["Color"] = rgb
    if font_dict:
        run["Font"] = font_dict
    return run
```

Table 15: Run-Level Parsing Issues

Issue	Resolution
COM uses UTF-16 indexing while Python uses code point indexing	Iterate using `tr.Length` (UTF-16); store `Run_Start_Index` in code point units
`hasattr()` misbehaves under COM dynamic dispatch	Wrap all attribute access with a `safe()` helper using `try/except`
COM RGB uses BGR byte order	In `rgb_of()`: `bgr & 0xFF` → R, `(bgr » 8) & 0xFF` → G, `(bgr » 16) & 0xFF` → B

Key Issues and Resolutions:

Table Merged Cell Tracking.

Problem: The PowerPoint COM API does not directly expose merged cell information for tables.

Solution: Geometric bounding box comparison

```
def parse_table(table):
    rows = table.Rows.Count
    cols = table.Columns.Count
    result = {"Dimensions": {"Rows": rows, "Columns": cols}, "Cells": {}}
    visited = {}
    for r in range(1, rows + 1):
        for c in range(1, cols + 1):
            cell = table.Cell(r, c)
            shape = cell.Shape
            geom_key = (
                round(shape.Left, 2),
```

```
                round(shape.Top, 2),
                round(shape.Width, 2),
                round(shape.Height, 2),
            )
            if geom_key in visited:
                anchor_r, anchor_c = visited[geom_key]
                key = f"{anchor_r},{anchor_c}"
                anchor_cell = result["Cells"][key]
                anchor_cell["_RowSpan"] = max(
                    anchor_cell["_RowSpan"], r - anchor_r + 1
                )
                anchor_cell["_ColSpan"] = max(
                    anchor_cell["_ColSpan"], c - anchor_c + 1
                )
                continue
            visited[geom_key] = (r, c)
            tf_detail = clean_cell_detail(
                parse_text_frame_debug(shape.TextFrame)
            )
            cell_detail = {
                "Text": tf_detail.get("Text", ""),
                "Runs": tf_detail.get("Runs", []),
                "_RowSpan": 1,
                "_ColSpan": 1,
            }
            v_anchor = safe(shape.TextFrame, "VerticalAnchor", None)
            if v_anchor is not None and v_anchor != 1:
                cell_detail["VerticalAlign"] = {3: "middle", 4: "bottom"}.get(v_anchor)
            if "Paragraphs" in tf_detail:
                cell_detail["Paragraphs"] = tf_detail["Paragraphs"]
            bg = get_cell_bg_color_hex(shape)
            if bg:
                cell_detail["BgColor"] = bg
            result["Cells"][f"{r},{c}"] = cell_detail
    for key, cell in list(result["Cells"].items()):
        rs = cell.pop("_RowSpan")
        cs = cell.pop("_ColSpan")
        if rs > 1 or cs > 1:
            cell["Merged"] = True
            cell["RowSpan"] = rs
            cell["ColSpan"] = cs
    return result
```

Output Example:

```
{
  "Dimensions": { "Rows": 3, "Columns": 3 },
  "Cells": {
    "1,1": {
      "Text": "Merged Header",
      "Runs": [{"Text": "Merged Header", "Font": {"Name": "Malgun Gothic", "Size": 14, "Bold": true}}],
      "Merged": true,
      "RowSpan": 2,
      "ColSpan": 2
    },
    "1,3": { "Text": "Regular Cell", "Runs": [...] },
    "2,3": { "Text": "Regular Cell", "Runs": [...] },
    "3,1": { "Text": "Bottom", "Runs": [...] }
  }
}
```

Textbox Autosizing.

Problem: After editing text, if the content becomes longer, it may overflow outside the shape or extend beyond the slide boundary.

Solution: Two-tier adaptive font reduction system

Tier 1 : Inside edit_text_rewrite.

```
if not is_auto_size and auto_resize:
    new_tr = tf.TextRange
    shape.Width = original_width
    current_size = new_tr.Font.Size if new_tr.Font.Size else 12.0
    if old_base_font_size:
        current_size = min(current_size, old_base_font_size)
    while new_tr.BoundHeight > original_height and current_size > 9.0:
        current_size -= 0.5
        new_tr.Font.Size = current_size
    if new_tr.Font.Size and current_size != new_tr.Font.Size:
        scale = current_size / new_tr.Font.Size
        for i in range(1, new_tr.Length + 1):
            ch = new_tr.Characters(i, 1)
            if ch.Font.Size:
                ch.Font.Size *= scale
```

Table 16: Tier 1 Behavior

Phase	Target	Behavior	Lower Bound
Uniform reduction	Entire text	Repeat `Font.Size -= 0.5`	9.0pt
Proportional scaling	Per character	`ch.Font.Size *= scale`	Preserves original ratio

Tier 2 : clamp_text_to_slide.

```python
def clamp_text_to_slide(prs, slide_number, shape_id):
    slide_w = prs.PageSetup.SlideWidth
    slide_h = prs.PageSetup.SlideHeight
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    if not shape.HasTextFrame:
        return
    def _overflow():
        over_r = (shape.Left + shape.Width) - slide_w
        over_b = (shape.Top + shape.Height) - slide_h
        return max(0, over_r, over_b)
    if _overflow() <= 1:
        return
    run_info = []
    tr = shape.TextFrame.TextRange
    for pi in range(1, tr.Paragraphs().Count + 1):
        para = tr.Paragraphs(pi)
        rc = para.Runs().Count
        if rc == 0:
            size = para.Font.Size if para.Font.Size and para.Font.Size > 0 else 12.0
            run_info.append((para, size))
        else:
            for ri in range(1, rc + 1):
                run = para.Runs(ri)
                size = run.Font.Size if run.Font.Size and run.Font.Size > 0 else 12.0
                run_info.append((run, size))
    max_orig = max(s for _, s in run_info)
    min_allowed = max(6.0, max_orig * 0.5)
    def _apply_scale(scale_factor):
        for run_obj, orig in run_info:
            new_s = max(min_allowed, round(orig * scale_factor * 2) / 2)
            run_obj.Font.Size = new_s
    shape_dim = max(shape.Width, shape.Height, 1)
    scale = max(0.5, 1.0 - _overflow() / shape_dim)
    _apply_scale(scale)
    for _ in range(4):
        ovf = _overflow()
        if ovf <= 1:
            break
        scale *= max(0.7, 1.0 - ovf / shape_dim)
        scale = max(0.5, scale)
        _apply_scale(scale)
```

Table 17: Textbox Autosizing Algorithm Properties

Property	Detail
Run-level processing	Significantly fewer COM calls compared to per-character access
Adaptive initial estimate	First scale computed as `1.0 - overflow / shape_dimension`
Iterative fine-tuning	Up to 4 additional correction passes after initial estimate (based on residual overflow)
Proportional reduction	Each Run is reduced proportionally to its original size → preserves mixed size ratios
0.5pt snapping	Aligned to 0.5pt increments via `round(size * 2) / 2` (rendering consistency)
Lower bound	Minimum `max(6.0, max_original_size × 0.5)` → ensures readability
Convergence condition	Stops when overflow $\leq$ 1pt

Algorithm Properties:

E.2 Win32COM Limitations

Win32COM provides the most faithful access to PowerPoint's application-resolved object state, but it also introduces practical constraints. It does not natively support parallel execution, so processing many slides can become a runtime bottleneck. The API also exposes overlapping `TextFrame` and `TextFrame2` interfaces whose behavior is not always interchangeable, and equation objects require special handling because Office Math structures are not exposed in the same way as ordinary text ranges. Figure 6 and Figure 7 illustrate representative equation-handling cases.

Figure 6: Equation Handling Issue

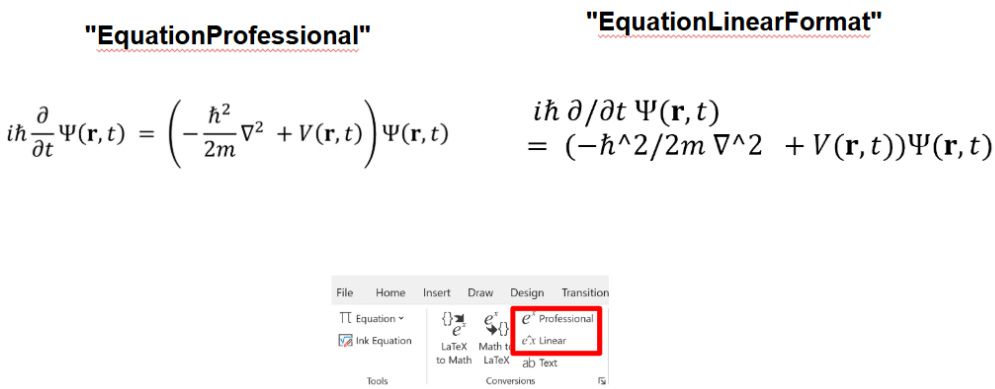

Figure 7: Equation Form

PowerPoint paragraph ranges can become unreliable when a text box contains carriage returns, soft line breaks, or mixed run-level formatting, so paragraph-level deletion and insertion may break the intended grouping or drop style information. The implementation avoids relying on visual paragraph boundaries and instead reconstructs the full paragraph text from mapped runs before writing the resolved text back.

The current paragraph update strategy replaces direct paragraph mutation with full-text reconstruction:

```
# Previous approach
paragraph.Text = ""
paragraph.InsertAfter("new text")
# Current approach
full_text = "".join(mapped_run_texts)
paragraph.Text = full_text
```

E.3 Claude PPT Editing Limitations

The current Claude PPT workflow reads and modifies OOXML directly across slides, which increases token cost and processing time even for simple repetitive edits. This full-package representation is especially inefficient when the intended operation touches only a small number of objects. It can also

make run-level text editing brittle because finely separated formatting spans are exposed verbatim, which increases the chance of hallucinated edits and observed artifacts such as broken spacing.

E.4 Full Tool Reference

A. Common Helpers.

_find_shape_by_id.

```python
def _find_shape_by_id(prs, slide_number, shape_id):
    slide = prs.Slides(slide_number)
    for shape in slide.Shapes:
        if shape.Id == shape_id:
            return shape
    raise ValueError(f"Shape {shape_id} not found on slide {slide_number}")
```

_hex_to_rgb_int.

```python
def _hex_to_rgb_int(hex_color):
    hex_color = hex_color.lstrip("#")
    r, g, b = int(hex_color[0:2], 16), int(hex_color[2:4], 16), int(hex_color[4:6], 16)
    return r + (g << 8) + (b << 16)
```

_apply_font_snapshot.

```python
def _apply_font_snapshot(font, snapshot):
    if snapshot.get("Name"):   font.Name = snapshot["Name"]
    if snapshot.get("Size"):   font.Size = snapshot["Size"]
    if snapshot.get("Bold") is not None:      font.Bold = snapshot["Bold"]
    if snapshot.get("Italic") is not None:    font.Italic = snapshot["Italic"]
    if snapshot.get("Underline") is not None: font.Underline = snapshot["Underline"]
    if snapshot.get("Color"):
        font.Color.RGB = _hex_to_rgb_int(snapshot["Color"])
```

_apply_overrides.

```python
def _apply_overrides(font, overrides):
    for key, value in overrides.items():
        if key == "Name":      font.Name = value
        elif key == "Size":    font.Size = value
        elif key == "Bold":    font.Bold = value
        elif key == "Italic":  font.Italic = value
        elif key == "Underline": font.Underline = value
        elif key == "Color":   font.Color.RGB = _hex_to_rgb_int(value)
        elif key == "Strikethrough": font.Strikethrough = value
```

_normalize_char_range.

```python
def _normalize_char_range(text_range, start, end):
    total = text_range.Length
    if start is None: start = 0
    if end is None:   end = total
    start = max(0, min(start, total))
    end   = max(start, min(end, total))
    return start, end
```

_resolve_insert_position.

```python
def _resolve_insert_position(text_range, position):
    if position == "start": return 0
    if position == "end":   return text_range.Length
    if isinstance(position, int):
        return max(0, min(position, text_range.Length))
    return text_range.Length
```

B. Text Editing (7 tools).

B.1 set_text_style.

```python
def set_text_style(prs, slide_number, shape_id, style_changes,
                   char_start=None, char_end=None, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    start, end = _normalize_char_range(tr, char_start, char_end)
    target = tr.Characters(start + 1, end - start)
    _apply_overrides(target.Font, style_changes)
    return f"Applied style to characters {start}-{end}"
```

B.2 edit_text_insert.

```python
def edit_text_insert(prs, slide_number, shape_id, text, position="end",
                     font_overrides=None, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    pos = _resolve_insert_position(tr, position)
    tr.Characters(pos + 1, 0).InsertBefore(text)
    if font_overrides:
        inserted = tr.Characters(pos + 1, len(text))
        _apply_overrides(inserted.Font, font_overrides)
    return f"Inserted text at position {pos}"
```

B.3 edit_text_delete.

```python
def edit_text_delete(prs, slide_number, shape_id,
                     char_start, char_end, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    start, end = _normalize_char_range(tr, char_start, char_end)
    tr.Characters(start + 1, end - start).Delete()
    return f"Deleted characters {start}-{end}"
```

B.4 edit_text_replace.

```python
def edit_text_replace(prs, slide_number, shape_id,
                      old_text, new_text, font_overrides=None,
                      slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    full = tr.Text
    idx = full.find(old_text)
    if idx == -1:
        return f"Text '{old_text}' not found"
    start = idx + 1
    length = len(old_text)
    target = tr.Characters(start, length)
    target.Text = new_text
    if font_overrides:
        replaced = tr.Characters(start, len(new_text))
        _apply_overrides(replaced.Font, font_overrides)
    return f"Replaced '{old_text}' with '{new_text}'"
```

B.5 edit_text_rewrite.

```python
def edit_text_rewrite(prs, slide_number, shape_id, runs,
                      auto_resize=True, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tf = shape.TextFrame
    original_width = shape.Width
    original_height = shape.Height
    is_auto_size = tf.AutoSize != 0
    old_base_font_size = None
    try:
        old_base_font_size = tf.TextRange.Font.Size
    except:
        pass
    tf.TextRange.Text = ""
    tr = tf.TextRange
    offset = 0
    for run in runs:
        text = run["Text"]
        tr.InsertAfter(text)
        segment = tr.Characters(offset + 1, len(text))
        if "Font" in run:
            _apply_font_snapshot(segment.Font, run["Font"])
        offset += len(text)
    if not is_auto_size and auto_resize:
        new_tr = tf.TextRange
        shape.Width = original_width
        current_size = new_tr.Font.Size if new_tr.Font.Size else 12.0
        if old_base_font_size:
            current_size = min(current_size, old_base_font_size)
        while new_tr.BoundHeight > original_height and current_size > 9.0:
            current_size -= 0.5
            new_tr.Font.Size = current_size
        if new_tr.Font.Size and current_size != new_tr.Font.Size:
            scale = current_size / new_tr.Font.Size
            for i in range(1, new_tr.Length + 1):
                ch = new_tr.Characters(i, 1)
                if ch.Font.Size:
                    ch.Font.Size *= scale
    return "Text rewritten"
```

B.6 set_paragraph_alignment.

```python
def set_paragraph_alignment(prs, slide_number, shape_id,
                            alignment, paragraph_index=None,
                            slide_json=None):
    ALIGN_MAP = {"left": 1, "center": 2, "right": 3, "justify": 4}
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    if paragraph_index is not None:
        para = tr.Paragraphs(paragraph_index + 1)
        para.ParagraphFormat.Alignment = ALIGN_MAP.get(alignment, 1)
    else:
        for i in range(1, tr.Paragraphs().Count + 1):
            tr.Paragraphs(i).ParagraphFormat.Alignment = ALIGN_MAP.get(alignment, 1)
    return f"Set alignment to {alignment}"
```

B.7 manage_bullet_points.

```python
def manage_bullet_points(prs, slide_number, shape_id,
                         action, paragraph_index=None,
                         bullet_char=None, indent_level=0,
                         slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    tr = shape.TextFrame.TextRange
    if paragraph_index is not None:
        paragraphs = [tr.Paragraphs(paragraph_index + 1)]
    else:
        paragraphs = [tr.Paragraphs(i) for i in range(1, tr.Paragraphs().Count + 1)]
    for para in paragraphs:
        pf = para.ParagraphFormat
        if action == "add":
            pf.Bullet.Type = 1
            if bullet_char:
                pf.Bullet.Character = ord(bullet_char)
            pf.IndentLevel = indent_level
        elif action == "remove":
            pf.Bullet.Type = 0
        elif action == "set_level":
            pf.IndentLevel = indent_level
    return f"Bullet {action} applied"
```

C. Table Editing (3 tools).

C.1 cell_text_style.

```python
def cell_text_style(prs, slide_number, shape_id,
                    row, col, style_changes,
                    char_start=None, char_end=None,
                    slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    cell = shape.Table.Cell(row, col)
    tr = cell.Shape.TextFrame.TextRange
    start, end = _normalize_char_range(tr, char_start, char_end)
    target = tr.Characters(start + 1, end - start)
    _apply_overrides(target.Font, style_changes)
    return f"Applied style to cell ({row},{col})"
```

C.2 replace_table_text.

```python
def replace_table_text(prs, slide_number, shape_id,
                       row, col, runs, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    cell = shape.Table.Cell(row, col)
    tf = cell.Shape.TextFrame
    tf.TextRange.Text = ""
    tr = tf.TextRange
    offset = 0
    for run in runs:
        text = run["Text"]
        tr.InsertAfter(text)
        segment = tr.Characters(offset + 1, len(text))
        if "Font" in run:
            _apply_font_snapshot(segment.Font, run["Font"])
        offset += len(text)
    return f"Replaced text in cell ({row},{col})"
```

C.3 table_layout_style.

```python
def table_layout_style(prs, slide_number, shape_id,
                       row=None, col=None,
                       width=None, height=None,
                       bg_color=None, border_color=None,
                       border_width=None,
                       vertical_align=None,
                       slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
```

```
    table = shape.Table
    if width and col:
        table.Columns(col).Width = width
    if height and row:
        table.Rows(row).Height = height
    if row and col:
        cell = table.Cell(row, col)
        if bg_color:
            cell.Shape.Fill.ForeColor.RGB = _hex_to_rgb_int(bg_color)
        if vertical_align:
            VA_MAP = {"top": 1, "middle": 3, "bottom": 4}
            cell.Shape.TextFrame.VerticalAnchor = VA_MAP.get(vertical_align, 1)
    return "Table layout updated"
```

D. Chart Editing (5 tools).

D.1 update_chart_categories.

```
def update_chart_categories(prs, slide_number, shape_id,
                            categories, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    chart = shape.Chart
    wb = chart.ChartData.Workbook
    ws = wb.Worksheets(1)
    for i, cat in enumerate(categories):
        ws.Cells(i + 2, 1).Value = cat
    chart.Refresh()
    wb.Close(False)
    return f"Updated {len(categories)} categories"
```

D.2 update_chart_series.

```
def update_chart_series(prs, slide_number, shape_id,
                        series_index, values, name=None,
                        slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    chart = shape.Chart
    wb = chart.ChartData.Workbook
    ws = wb.Worksheets(1)
    col = series_index + 1
    if name:
        ws.Cells(1, col).Value = name
    for i, val in enumerate(values):
        ws.Cells(i + 2, col).Value = val
    chart.Refresh()
    wb.Close(False)
    return f"Updated series {series_index}"
```

D.3 update_chart_structure.

```
def update_chart_structure(prs, slide_number, shape_id,
                           chart_type=None, has_legend=None,
                           has_title=None, title_text=None,
                           slide_json=None):
    CHART_TYPES = {
        "column": 51, "bar": 57, "line": 4, "pie": 5,
        "area": 1, "scatter": -4169, "doughnut": -4120,
    }
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    chart = shape.Chart
    if chart_type and chart_type in CHART_TYPES:
        chart.ChartType = CHART_TYPES[chart_type]
    if has_legend is not None:
        chart.HasLegend = has_legend
    if has_title is not None:
        chart.HasTitle = has_title
    if title_text and chart.HasTitle:
        chart.ChartTitle.Text = title_text
    return "Chart structure updated"
```

D.4 update_chart_axes.

```
def update_chart_axes(prs, slide_number, shape_id,
                      axis_type="value",
                      min_value=None, max_value=None,
                      title=None, number_format=None,
                      slide_json=None):
    AXIS_MAP = {"category": 1, "value": 2}
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    chart = shape.Chart
    axis = chart.Axes(AXIS_MAP.get(axis_type, 2))
    if min_value is not None: axis.MinimumScale = min_value
    if max_value is not None: axis.MaximumScale = max_value
    if title:
        axis.HasTitle = True
```

```
        axis.AxisTitle.Text = title
    if number_format:
        axis.TickLabels.NumberFormat = number_format
    return f"Updated {axis_type} axis"
```

D.5 update_chart_colors.

```
def update_chart_colors(prs, slide_number, shape_id,
                        series_colors=None, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    chart = shape.Chart
    if series_colors:
        for idx, color in series_colors.items():
            series = chart.SeriesCollection(int(idx))
            series.Format.Fill.ForeColor.RGB = _hex_to_rgb_int(color)
    return "Chart colors updated"
```

E. Shape/Layout (13 tools).

E.1 adjust_layout.

```
def adjust_layout(prs, slide_number, shape_id,
                  left=None, top=None, width=None, height=None,
                  rotation=None, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    if left is not None:     shape.Left = left
    if top is not None:      shape.Top = top
    if width is not None:    shape.Width = width
    if height is not None:   shape.Height = height
    if rotation is not None: shape.Rotation = rotation
    return f"Layout adjusted for shape {shape_id}"
```

E.2 distribute_shapes.

```
def distribute_shapes(prs, slide_number, shape_ids,
                      direction="horizontal", spacing=None,
                      slide_json=None):
    shapes = [_find_shape_by_id(prs, slide_number, sid) for sid in shape_ids]
    if direction == "horizontal":
        shapes.sort(key=lambda s: s.Left)
        if spacing is None:
            total_w = sum(s.Width for s in shapes)
            slide_w = prs.PageSetup.SlideWidth
            spacing = (slide_w - total_w) / (len(shapes) + 1)
        x = spacing
        for s in shapes:
            s.Left = x
            x += s.Width + spacing
    else:
        shapes.sort(key=lambda s: s.Top)
        if spacing is None:
            total_h = sum(s.Height for s in shapes)
            slide_h = prs.PageSetup.SlideHeight
            spacing = (slide_h - total_h) / (len(shapes) + 1)
        y = spacing
        for s in shapes:
            s.Top = y
            y += s.Height + spacing
    return f"Distributed {len(shapes)} shapes {direction}ly"
```

E.3 align_shapes.

```
def align_shapes(prs, slide_number, shape_ids,
                 alignment="center", slide_json=None):
    shapes = [_find_shape_by_id(prs, slide_number, sid) for sid in shape_ids]
    if alignment == "left":
        min_left = min(s.Left for s in shapes)
        for s in shapes: s.Left = min_left
    elif alignment == "right":
        max_right = max(s.Left + s.Width for s in shapes)
        for s in shapes: s.Left = max_right - s.Width
    elif alignment == "center":
        cx = sum(s.Left + s.Width / 2 for s in shapes) / len(shapes)
        for s in shapes: s.Left = cx - s.Width / 2
    elif alignment == "top":
        min_top = min(s.Top for s in shapes)
        for s in shapes: s.Top = min_top
    elif alignment == "bottom":
        max_bottom = max(s.Top + s.Height for s in shapes)
        for s in shapes: s.Top = max_bottom - s.Height
    elif alignment == "middle":
        cy = sum(s.Top + s.Height / 2 for s in shapes) / len(shapes)
        for s in shapes: s.Top = cy - s.Height / 2
    return f"Aligned {len(shapes)} shapes to {alignment}"
```

E.4 create_textbox.

```python
def create_textbox(prs, slide_number, left, top, width, height,
                   text="", font_overrides=None, slide_json=None):
    slide = prs.Slides(slide_number)
    shape = slide.Shapes.AddTextbox(1, left, top, width, height)
    shape.TextFrame.TextRange.Text = text
    if font_overrides:
        _apply_overrides(shape.TextFrame.TextRange.Font, font_overrides)
    return f"Created textbox on slide {slide_number}"
```

E.5 create_placeholder.

```python
def create_placeholder(prs, slide_number, layout_index,
                       placeholder_type, left, top, width, height,
                       slide_json=None):
    slide = prs.Slides(slide_number)
    layout = prs.SlideMaster.CustomLayouts(layout_index)
    ph = layout.Placeholders.Add(placeholder_type, left, top, width, height)
    return f"Created placeholder on slide {slide_number}"
```

E.6 create_shape.

```python
def create_shape(prs, slide_number, shape_type,
                 left, top, width, height,
                 fill_color=None, line_color=None,
                 text=None, slide_json=None):
    SHAPE_TYPES = {
        "rectangle": 1, "rounded_rectangle": 5,
        "ellipse": 9, "triangle": 7,
        "right_arrow": 33, "star": 92,
    }
    slide = prs.Slides(slide_number)
    auto_type = SHAPE_TYPES.get(shape_type, 1)
    shape = slide.Shapes.AddShape(auto_type, left, top, width, height)
    if fill_color:
        shape.Fill.ForeColor.RGB = _hex_to_rgb_int(fill_color)
    if line_color:
        shape.Line.ForeColor.RGB = _hex_to_rgb_int(line_color)
    if text:
        shape.TextFrame.TextRange.Text = text
    return f"Created {shape_type} on slide {slide_number}"
```

E.7 delete_shape.

```python
def delete_shape(prs, slide_number, shape_id, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    shape.Delete()
    return f"Deleted shape {shape_id} from slide {slide_number}"
```

E.8 duplicate_shape.

```python
def duplicate_shape(prs, slide_number, shape_id,
                    target_slide, offset_left=0, offset_top=0,
                    slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    shape.Copy()
    target = prs.Slides(target_slide)
    target.Shapes.Paste()
    new_shape = target.Shapes(target.Shapes.Count)
    new_shape.Left = shape.Left + offset_left
    new_shape.Top = shape.Top + offset_top
    return f"Duplicated shape {shape_id} to slide {target_slide}"
```

E.9 duplicate_shape_within_slide.

```python
def duplicate_shape_within_slide(prs, slide_number, shape_id,
                                 offset_left=72, offset_top=0,
                                 slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    shape.Copy()
    slide = prs.Slides(slide_number)
    slide.Shapes.Paste()
    new_shape = slide.Shapes(slide.Shapes.Count)
    new_shape.Left = shape.Left + offset_left
    new_shape.Top = shape.Top + offset_top
    return f"Duplicated shape {shape_id} within slide {slide_number}"
```

E.10 apply_visual_style.

```python
def apply_visual_style(prs, slide_number, shape_id,
                       fill_color=None, line_color=None,
                       line_width=None, opacity=None,
                       shadow=None, slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
```

```
    if fill_color:
        shape.Fill.Solid()
        shape.Fill.ForeColor.RGB = _hex_to_rgb_int(fill_color)
    if line_color:
        shape.Line.ForeColor.RGB = _hex_to_rgb_int(line_color)
    if line_width is not None:
        shape.Line.Weight = line_width
    if opacity is not None:
        shape.Fill.Transparency = 1.0 - opacity
    if shadow is not None:
        shape.Shadow.Visible = shadow
    return f"Visual style applied to shape {shape_id}"
```

E.11 apply_gradient_fill.

```
def apply_gradient_fill(prs, slide_number, shape_id,
                        color1, color2, angle=0,
                        slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    shape.Fill.TwoColorGradient(1, 1)
    shape.Fill.GradientAngle = angle
    shape.Fill.ForeColor.RGB = _hex_to_rgb_int(color1)
    shape.Fill.BackColor.RGB = _hex_to_rgb_int(color2)
    return f"Gradient fill applied to shape {shape_id}"
```

E.12 insert_image.

```
def insert_image(prs, slide_number, image_path,
                 left, top, width=None, height=None,
                 slide_json=None):
    slide = prs.Slides(slide_number)
    if width and height:
        shape = slide.Shapes.AddPicture(
            image_path, False, True, left, top, width, height
        )
    else:
        shape = slide.Shapes.AddPicture(
            image_path, False, True, left, top
        )
    return f"Inserted image on slide {slide_number}"
```

E.13 edit_image.

```
def edit_image(prs, slide_number, shape_id,
               new_image_path=None, crop=None,
               brightness=None, contrast=None,
               slide_json=None):
    shape = _find_shape_by_id(prs, slide_number, shape_id)
    if new_image_path:
        left, top, w, h = shape.Left, shape.Top, shape.Width, shape.Height
        shape.Delete()
        slide = prs.Slides(slide_number)
        new_shape = slide.Shapes.AddPicture(
            new_image_path, False, True, left, top, w, h
        )
    else:
        if crop:
            pf = shape.PictureFormat
            if "left" in crop:   pf.CropLeft = crop["left"]
            if "right" in crop:  pf.CropRight = crop["right"]
            if "top" in crop:    pf.CropTop = crop["top"]
            if "bottom" in crop: pf.CropBottom = crop["bottom"]
        if brightness is not None:
            shape.PictureFormat.Brightness = brightness
        if contrast is not None:
            shape.PictureFormat.Contrast = contrast
    return f"Edited image shape {shape_id}"
```

F. Slide Management (5 tools).

F.1 add_slide.

```
def add_slide(prs, layout_index=1, position=None,
              slide_json=None, agent_request=None):
    layout = prs.SlideMaster.CustomLayouts(layout_index)
    if position:
        slide = prs.Slides.AddSlide(position, layout)
    else:
        slide = prs.Slides.AddSlide(prs.Slides.Count + 1, layout)
    return f"Added slide at position {slide.SlideIndex}"
```

F.2 delete_slide.

```
def delete_slide(prs, slide_number,
                 slide_json=None, agent_request=None):
    prs.Slides(slide_number).Delete()
    return f"Deleted slide {slide_number}"
```

F.3 duplicate_slide.

```
def duplicate_slide(prs, slide_number, target_position=None,
                    slide_json=None, agent_request=None):
    prs.Slides(slide_number).Duplicate()
    if target_position:
        new_slide = prs.Slides(slide_number + 1)
        new_slide.MoveTo(target_position)
    return f"Duplicated slide {slide_number}"
```

F.4 set_slide_transition.

```
def set_slide_transition(prs, slide_number,
                         transition_type=None, duration=None,
                         advance_on_click=None, advance_time=None,
                         slide_json=None, agent_request=None):
    slide = prs.Slides(slide_number)
    sst = slide.SlideShowTransition
    if transition_type is not None:
        sst.EntryEffect = transition_type
    if duration is not None:
        sst.Duration = duration
    if advance_on_click is not None:
        sst.AdvanceOnClick = advance_on_click
    if advance_time is not None:
        sst.AdvanceOnTime = True
        sst.AdvanceTime = advance_time
    return f"Transition set for slide {slide_number}"
```

F.5 set_slide_background.

```
def set_slide_background(prs, slide_number,
                         bg_color=None, image_path=None,
                         gradient=None,
                         slide_json=None, agent_request=None):
    slide = prs.Slides(slide_number)
    bg = slide.Background
    fill = bg.Fill
    if bg_color:
        fill.Solid()
        fill.ForeColor.RGB = _hex_to_rgb_int(bg_color)
    elif image_path:
        fill.UserPicture(image_path)
    elif gradient:
        fill.TwoColorGradient(1, 1)
        fill.ForeColor.RGB = _hex_to_rgb_int(gradient["color1"])
        fill.BackColor.RGB = _hex_to_rgb_int(gradient["color2"])
        if "angle" in gradient:
            fill.GradientAngle = gradient["angle"]
    return f"Background set for slide {slide_number}"
```