PPTXEDIT-BENCH: A DIAGNOSTIC BENCHMARK FOR LANGUAGE-GUIDED POWERPOINT EDITING

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-guided editing can help users adapt existing presentations to different audiences and communication goals. Users often request edits without listing every PPTX object involved, requiring systems to use slide context to identify all objects to modify, use as references, or exclude. Existing benchmarks provide limited support for directly evaluating this process. We introduce **PPTXEdit-Bench**, a diagnostic benchmark for exact PPTX editing, comprising 22,100 editing instances from 4,420 generated slides across 27 actions and five request forms. Its annotations link instructions to complete object membership and editing roles for individual elements, composite entities, and selection sets. These annotations and executable outcome checks support separate evaluation of object grounding and editing correctness, enabling diagnosis of editing failures. Across four backbones, Gold interventions in our diagnostic baseline **SlideDiag** yield greater recovery from correcting object bindings than from correcting intent, identifying grounding as a key bottleneck.

1 INTRODUCTION

Language and multimodal models have advanced presentation automation, from content synthesis to slide organization and visual layout (Fu et al., 2022; Sun et al., 2021; Ge et al., 2025; Zheng et al., 2025). Yet creating a presentation does not end the authoring process. Users routinely refine existing slides to improve visual clarity, adjust emphasis, and adapt their presentations to different audiences. These revisions often involve repositioning content blocks, aligning related elements, or making image sizes and spacing more consistent. Much of a slide's content and design may already be suitable, so the goal is to make specific adjustments while preserving the work already completed. Reliable support for these revisions is an important part of making presentation automation useful in everyday workflows. Natural language offers a convenient way to specify such adjustments. A user may ask an assistant to move the leftmost card 12 pt to the right" or make the other images as wide as the leftmost image." We refer to this task as *language-guided exact PPTX editing* (Figure 1). The system edits native objects in an existing PPTX file, such as shapes, images, and text boxes. Here, *exact* means applying the specified changes to the intended objects while preserving unrelated document state. The resulting presentation remains editable for subsequent revisions.

A central challenge is that the visual units users describe do not always correspond to individual objects in the PPTX file. An information card" may appear as a single unit but consist of a background shape, an icon, and several separately stored text boxes, with no native group connecting them. Moving the card requires identifying all of these parts and moving them together. A description such as all blue icons" instead defines a set through a shared attribute, requiring the system to find every matching object. Objects can also play different roles: in the width-matching request, the leftmost image supplies the reference width, while the remaining images are the modification targets. An instruction need not name every participating object individually. The system must recover the complete object scope and editing roles from the language and the slide context. We refer to this process as *object grounding*.

Evaluating this task therefore calls for examining object grounding alongside the final edit. Existing presentation benchmarks assess instruction following, layout understanding, native-object editing, and content updates (Guo et al., 2024; Zhang et al., 2024; Huang et al., 2025; Ofengenden et al., 2025; Zhou et al., 2026). Recent evaluations also consider partial completion, unintended changes,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

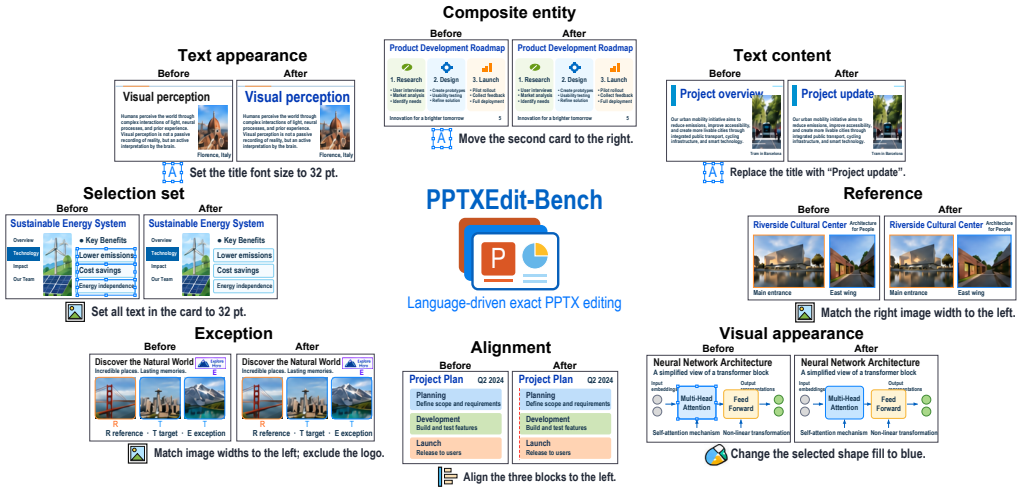

Figure 1: Examples of language-guided exact PPTX editing.

and intermediate interaction states (Gandhi et al., 2026; Ji et al., 2026). However, explicit annotations linking user descriptions to complete native-object membership and editing roles remain limited. The intended object scope cannot be inferred from changes alone: an image used as a width reference participates in a request without being modified. Knowing this intended scope is also insufficient to explain a failed edit. If only a card's background moves, the system may have omitted the other components or selected them correctly but failed to move them. Examining these possibilities requires comparing a system's intermediate object selections and role assignments against the annotated scope of the request.

To support this comparison, we introduce **PPTXEdit-Bench**. The benchmark is designed to evaluate whether models can resolve natural-language requests into complete sets of native objects, distinguish their editing roles, and apply the requested changes precisely. For each instruction, the benchmark records the complete set of objects involved, distinguishes their roles, and provides an expected edited PPTX. The annotations cover individual elements, composite entities such as cards, and selection sets defined by a condition. Human annotators review the membership of composite entities and selection sets, correcting errors and adding omitted composite entities. Executable checks assess whether the requested modifications are completed and whether unrelated object properties are preserved. The benchmark covers variation in both how edits are requested and how slides are structured. It contains 22,100 editing instances constructed from 4,420 slides and covers 27 actions involving layout, appearance, and content. Its five request forms cover edits to single targets and target sets, requirements specified directly or through reference objects, and relations among multiple objects. The source slides are generated by multiple systems across different application scenarios to vary both visual layouts and the organization of native objects.

To examine the decisions models make while carrying out these requests, we introduce **SlideDiag**, a diagnostic baseline that separates instruction interpretation, object grounding, and edit execution. It makes the predicted operation and the objects selected for each editing role explicit before execution. These intermediate predictions can then be checked against the benchmark annotations to help distinguish errors in interpretation and grounding from failures during execution. Our contributions are summarized as follows:

- We introduce PPTXEdit-Bench for language-guided exact editing of existing PPTX presentations, covering layout, appearance, and content changes across different object scopes and request forms.
- We provide annotations of complete object membership and editing roles linked to expected edits, supporting grounding assessment alongside executable checks of editing outcomes.

- We develop SlideDiag to make predictions of editing intent and object selection explicit before execution, enabling comparison with benchmark annotations for error analysis.

2 Related Work

2.1 Presentation Generation and Editing

Presentation automation covers document-to-slide generation (Li et al., 2021; Fu et al., 2022) and programmatic or edit-based slide creation (Ge et al., 2025; Zheng et al., 2025). Existing benchmarks evaluate instruction following and robustness (Guo et al., 2024; Zhang et al., 2024), layout and editing capabilities (Huang et al., 2025; Ofengenden et al., 2025; Jung et al., 2026), and data-driven content updates (Zhou et al., 2026). However, explicit ground truth for the complete native-object membership and editing roles implied by each instruction remains limited. PPTXEdit-Bench provides these annotations across individual elements, composite entities, and selection sets, making membership completeness and role assignment explicit evaluation targets.

2.2 Object Grounding and Diagnostic Evaluation

GUI grounding and interaction models connect instructions to interface elements and actions (Cheng et al., 2024; Gou et al., 2025; Hong et al., 2023; Lin et al., 2025), while broader benchmarks assess desktop, office, and web tasks (Xie et al., 2024; Wang et al., 2024; Deng et al., 2023; Zhou et al., 2024). Fine-grained evaluations further examine intermediate progress (Ma et al., 2024), GUI grounding and state correctness (Ji et al., 2026), and partial completion and extraneous changes (Gandhi et al., 2026). However, progress and state checks alone do not directly assess whether all required native document objects have been identified and assigned the correct editing roles. PPTXEdit-Bench links explicit membership and role annotations to executable outcome checks, enabling separate evaluation of grounding decisions and editing correctness. This allows grounding errors to be examined alongside editing outcomes, supporting more precise diagnosis of editing failures.

3 PPTXEdit-Bench

PPTXEdit-Bench supports evaluation of both object grounding and editing correctness. We construct the benchmark in three stages. First, we generate and filter native, editable source slides. Second, we annotate individual elements, composite entities, and selection sets, recording their complete native-object membership and generating corresponding referring expressions. Third, for each instance, we select an applicable action from the action library and specify participating objects, editing roles, and arguments. From this shared specification, we generate the instruction using a template and execute the edit to produce the expected edited PPTX (Gold PPTX). We validate annotations, instruction–edit consistency, and Gold edits, including preservation of unaffected state. Each validated instance contains a source PPTX, an editing instruction, a Gold PPTX, and structured annotations of the action, arguments, object membership, and roles.

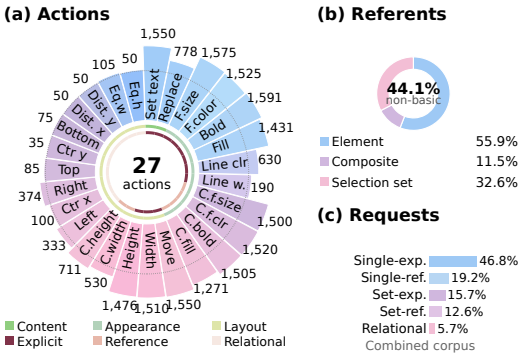

Figure 2: Dataset statistics.

3.1 Source Slide Construction

We prepare content briefs across eight application scenarios, including academic research, technical systems, teaching, and business decisions. Each brief specifies a topic, target audience, communication goal, and required content. We insert these fields into a shared prompt template that requires

native, editable PPTX objects and provides guidance on content organization, alignment, and visual consistency. We generate presentations from these prompts using PPTAgent (Zheng et al., 2025), AutoPresent (Ge et al., 2025), Presenton, Codex, and Claude Code, allowing each system to determine the specific layouts and object organization.

We then parse the generated PPTX files to check object accessibility and editability, and inspect rendered slides for missing assets, content overflow, and obstructive overlap. We also verify that visible content corresponds to native objects and screen for duplicate slides and highly repetitive structures. We prioritize slides with varied object types and structures such as information cards, text–image modules, and process diagrams, providing content units and object collections for subsequent annotation and editing tasks. This process yields 4,420 source slides from 997 presentations. The full prompt, scenario list, and presentation counts by system are provided in Appendix B.1.

3.2 Referent Annotation and Expression Generation

We annotate the complete native-object membership of three referent types and generate corresponding descriptions for use in editing instructions.

Referent Membership Annotation. *Individual elements* are independently accessible PPTX objects, e.g., text boxes, shapes, images, charts, and tables. We record their identifiers and treat charts and tables as whole objects. Appendix B.2 summarizes element types and editable properties.

Composite entities are coherent visual or semantic units composed of multiple elements, such as a card containing a background, an icon, and text boxes. We automatically generate candidates from native groups or infer them from background containment, proximity, and alignment when explicit grouping is absent.

Selection sets contain all objects satisfying a condition within a scope, such as "all text boxes in the second card." We construct candidates using object type, parent-entity membership, or shared attributes, recording each set's scope, condition, and complete membership. A set may have the same membership as a composite entity.

Eight annotators inspect candidate memberships using rendered slides and native-object information. They accept, correct, or reject candidates for both composite entities and selection sets and add omitted composite entities. Expressions do not expose internal PPTX identifiers but remain linked to their native objects. Appendix B.3 details candidate construction and human review.

Referring Expression Generation. We use a predefined lexicon and templates for each referent type to combine names with visible text, appearance, position, or ordinal cues. Set expressions also specify scope and conditions. We check expressions against slides and membership annotations to ensure that they unambiguously identify the complete intended object set. Ambiguous expressions are revised or discarded; after deduplication, we retain at least three valid expressions per referent. Expressions do not expose internal PPTX identifiers but remain linked to their native objects. Appendix B.4 describes expression construction and verification.

3.3 Edit Instance Construction

We combine annotated referents with an action, editing roles, and arguments to specify each instance. Referring expressions describe the participants in the instruction, while membership annotations identify the native objects for execution. The instruction and Gold PPTX are generated from this shared specification.

3.3.1 Action and Argument Design

The action library contains 27 operations across content, appearance, and layout. Each specifies applicable objects, editing roles, arguments, and validity conditions. *Explicit-value* actions use values or displacements supplied in the instruction; *reference-based* actions obtain target values from reference objects; and *relational* actions establish alignment, spacing, or size relations. Explicit arguments are sampled from predefined pools, such as font-size and color candidates, or derived from

current object properties, subject to compatibility and validity checks. Relational edits follow shared policies; for example, left alignment uses the leftmost original edge among participating objects.

The explicit-value and reference-based modes each support single-target and set-level scopes. Together with multi-object relational editing, this yields five request forms: single-target explicit, single-target reference-based, set-level explicit, set-level reference-based, and multi-object relational. A single target may be an individual element or a composite entity: moving one card remains a single-target edit even when it contains multiple native objects. Request form and referent type therefore describe editing requirements and object identification, respectively. Appendix B.5 details object compatibility, argument generation, and relational execution policies.

3.3.2 Instruction Template Design

Instruction templates combine the referring expressions from the previous stage with actions and arguments to form complete editing requests. We design 4–16 canonical templates per action, totaling 166 across the five request forms. Templates contain slots for object descriptions and arguments, as in "Match the width of {target} to that of {reference}." An LLM generates three paraphrases per template, preserving action, role, and argument meanings. After checking semantic consistency and slot completeness and removing duplicates, we retain 498 paraphrases, yielding 664 variants. Appendix B.6 details template construction and paraphrase verification.

3.3.3 User Instruction and Gold PPTX Generation

We construct five editing instances per source slide, generating each instruction and Gold PPTX from a shared specification of participating objects, action, and arguments. For each instance, we randomly select an annotated referent—an individual element, composite entity, or selection set—and an applicable action. We select any additional objects required by the action from the same slide, assign editing roles, and sample valid arguments according to the parameter dictionaries and generation rules (Appendix B.5). We fill a compatible instruction template with referring expressions and arguments. In parallel, the execution function uses membership annotations to locate native objects and applies the edit to a copy of the source PPTX. When moving "the second metric card," the instruction combines this description with the sampled direction and distance, while execution moves its background, icon, and text boxes together.

Given the source slide and benchmark action definitions, each instruction is unambiguous and has unique, precise ground-truth labels for the action, role-specific object membership, and arguments. Verified referring expressions and fully instantiated templates ensure this correspondence; source-object properties and predefined relational execution policies determine the exact expected values of the edited properties. Each instance contains the source PPTX, instruction, Gold PPTX, and these annotations.

3.4 Data Validation

We verify each instance's object membership and editing roles, and check that its instruction matches the edit specification. For each Gold PPTX, we verify the requested changes, preservation of unaffected objects and properties, and successful rendering. Instances with identified issues are reviewed and corrected; those that cannot be unambiguously verified or reliably repaired are discarded. Appendices B.3, B.4, and B.6 detail annotation review, expression verification, and template checks, respectively; Appendix B.7 describes execution checks.

3.5 Dataset Statistics and Splits

PPTXEdit-Bench contains 22,100 editing instances from 4,420 slides across 997 presentations, covering 27 actions and five request forms. Its 94,096 referent annotations cover individual elements, composite entities, and selection sets; a native object may occur in multiple annotations. Figure 2 summarizes the distributions of actions, referent types, and request forms.

We partition the dataset at the presentation level, keeping all slides from each presentation and their derived instances in the same split. The training, validation, and test sets contain 15,470 (70%),

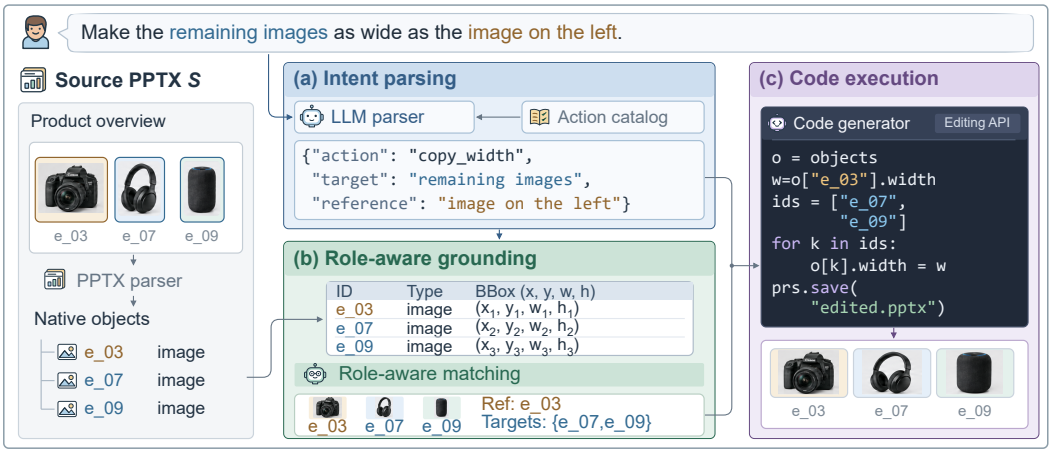

Figure 3: Overview of SlideDiag.

2,210 (10%), and 4,420 (20%) instances, respectively. Appendix B.8 provides detailed distributions and split statistics.

4 SLIDEDIAG

We introduce SlideDiag, a diagnostic baseline that separates PPTX editing into intent parsing, role-aware object grounding, and code execution. Given a user instruction u and a source PPTX S, it predicts a structured intent I, resolves its object descriptions into bindings B, and generates and executes editing code (Figure 3). Each stage uses a separate language-model call. The explicit intent and bindings support direct evaluation against benchmark annotations and Gold substitutions, allowing us to measure performance changes after correcting intent or object bindings.

4.1 EDIT-INTENT PARSING

The parser receives u and the action library (Appendix B.5) and produces a structured intent:

$$I = (a, \boldsymbol{v}, \mathcal{R}), \tag{1}$$

where a is the action, $\boldsymbol{v}$ records explicit arguments supplied in the instruction, and $\mathcal{R}$ maps each required editing role to one or more natural-language referring expressions. The roles distinguish targets, references, and relational participants.

For "Make the remaining images as wide as the image on the left," the parser predicts `copy_width`, assigning "the remaining images" to the target role and "the image on the left" to the reference role. No explicit width value is recorded because it is obtained from the reference during execution. Object descriptions remain in natural language for resolution in the grounding stage. Appendix C.1 provides the parsing prompt, output schema.

4.2 ROLE-AWARE OBJECT GROUNDING

The grounding stage resolves these descriptions to native objects. We extract an object inventory $\mathcal{O}(S)$ from the source PPTX, including identifiers, types, text, appearance attributes, positions, dimensions, and native grouping relations. Given I and $\mathcal{O}(S)$, the model predicts role-specific bindings:

$$B = \{r \mapsto B_r \mid r \in \mathrm{dom}(\mathcal{R})\}, \tag{2}$$

where B_r lists the editing units assigned to role r, each represented by a set of native-object identifiers. A composite entity is bound as one unit, while a selection set covers all matching units. Bindings must preserve complete membership and keep distinct units separate for subsequent layout edits. In the running example, the leftmost image `e_03` is bound as the reference, while `e_07`

and e_09 form two target units. The intent and bindings are then passed to code execution. Appendix C.2 details the source-object representation, grounding prompt, output schema.

4.3 Code Execution

Given I, B, and $\mathcal{O}(S)$, the model generates a Python editing function c using `python-pptx`. The runtime resolves the bound identifiers to native objects, executes the function on the source PPTX, and saves the edited result:

$$c = f_{\text{code}}(I, B, \mathcal{O}(S)), \qquad \widehat{S} = \text{Execute}(S, c). \tag{3}$$

For width matching, the function reads the width of e_03 and assigns it to e_07 and e_09. For composite movement and alignment, it computes displacements from each unit's overall bounds and applies them to its members. The prompt specifies the shared execution policies for relational actions (Appendix B.5) and requires preservation of unaffected state. Appendix C.3 provides the code-generation prompt, function interface, and execution checks.

5 Experiments

5.1 Experimental Setup

We evaluate nine backbones: Gemma-4-31B and Gemma-4-26B-A4B (Gemma Team, 2026), Qwen3.8-27B (Qwen Team, 2026), DeepSeek-V4-Flash (DeepSeek-AI, 2026), GPT-OSS-20B (OpenAI, 2025), GLM-5.3-Flash (Z.ai, 2026), Gemini-3.7-Flash (Google DeepMind, 2026), GPT-5.6-Terra (OpenAI, 2026b), and Qwen3.8-Flash (Alibaba Cloud, 2026). The editing systems comprise four general-purpose coding-agent harnesses—Codex (Bolin, 2026), Claude Code (Anthropic, 2025), DeepCode (Li et al., 2025), and OpenCode (Anomaly, 2025)—three one-shot editors (Direct Code (Ge et al., 2025; Guo et al., 2024), PPT-JSON (Zheng et al., 2025), and OOXML DOM (Ofengenden et al., 2025)), and SlideDiag.

Configurations are selected on the validation set and evaluated on the test split of PPTXEdit-Bench. All systems receive the same source PPTX files and editing instructions. To balance coverage and evaluation cost, we conduct skill comparisons on five backbones: three open-weight and two proprietary models. For each selected backbone, we compare all four coding-agent harnesses with and without OpenAI's Google Slides skill (OpenAI, 2026a). Appendix D describes the software sources, evaluation protocol, and efficiency measurement procedures.

We report editing success rate (SR) for all systems and intent and object-binding accuracy for SlideDiag. SR requires a parseable output, correct completion of all requested changes, and preservation of the remaining checked properties of target and non-target objects. Intent accuracy requires exact agreement on the action and arguments; object-binding accuracy requires exact matches to the annotated native-object sets for every role. Each metric is the percentage of test instances satisfying its criterion.

5.2 Overall Editing Performance

Table 1 reports editing SR for base configurations in panel (a) and coding agents augmented with the Google Slides skill in panel (b). Parenthesized values indicate gains over the corresponding base configurations in percentage points.

For all nine backbones, the best one-shot editor outperforms the best coding agent without skills, highlighting the importance of the editing system under a fixed backbone. Adding the Google Slides skill improves coding-agent SR by 3.67–23.19 percentage points across the evaluated backbone–harness pairs. SlideDiag achieves the highest SR for every evaluated backbone, ranging from 63.98% to 88.51%. Across the five backbones tested with the skill, it outperforms the best skill-augmented coding agent for each backbone by an average of 4.80 percentage points.

Despite these improvements, even the best configuration fails on more than one in ten requests. Overall SR alone cannot determine whether these failures arise from intent parsing, role-aware object grounding, or code execution. We therefore evaluate SlideDiag's intermediate predictions

Table 1: Editing success rates (%) on PPTXEdit-Bench.

(a) Base configurations

Model	Coding agents				One-shot editors			SlideDiag (Ours)
	Codex	Claude Code	DeepCode	OpenCode	Direct Code	PPT-JSON	OOXML DOM	
Open-weight models								
Gemma-4-31B	53.73	46.83	69.57	58.94	79.82	67.62	61.04	**82.19**
Gemma-4-26B-A4B	50.00	43.51	66.00	55.50	66.49	64.50	58.51	**69.50**
Qwen3.8-27B	60.00	64.00	62.29	65.95	83.26	83.62	82.76	**84.23**
DeepSeek-V4-Flash	63.48	62.90	74.55	72.17	81.20	82.74	78.78	**85.14**
GPT-OSS-20B	43.14	46.99	44.07	50.95	62.01	59.93	61.04	**63.98**
GLM-5.3-Flash	53.62	63.42	73.19	72.74	81.67	82.53	81.43	**85.36**
Proprietary models								
Gemini-3.7-Flash	79.86	79.00	78.51	76.99	85.14	84.73	84.00	**88.51**
GPT-5.6-Terra	76.49	75.00	74.55	71.38	83.03	83.60	82.51	**87.40**
Qwen3.8-Flash	76.02	78.62	75.68	75.54	84.43	83.21	83.87	**86.76**

(b) PPT-skill coding agents vs. SlideDiag

Model	Coding agents with PPT skill				SlideDiag (Ours)
	Codex	Claude Code	DeepCode	OpenCode	
Qwen3.8-27B	78.51 (+18.51)	78.01 (+14.01)	73.51 (+11.22)	75.00 (+9.05)	84.23
DeepSeek-V4-Flash	77.38 (+13.90)	79.12 (+16.22)	80.54 (+5.99)	78.96 (+6.79)	85.14
GLM-5.3-Flash	76.81 (+23.19)	79.75 (+16.33)	79.55 (+6.36)	80.25 (+7.51)	85.36
GPT-5.6-Terra	83.01 (+6.52)	81.99 (+6.99)	81.49 (+6.94)	79.00 (+7.62)	87.40
Qwen3.8-Flash	82.58 (+6.56)	82.29 (+3.67)	81.99 (+6.31)	79.98 (+4.44)	86.76

(a) Intermediate diagnostics

	GPT	Qwen	DeepSeek	GLM
Intent	95.1	94.6	93.8	94.2
Binding	88.4	87.7	86.3	85.9
Final SR	87.4	86.8	85.1	85.4

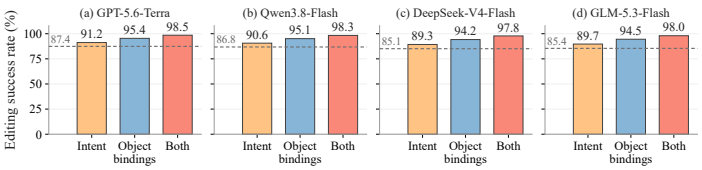

Figure 4: Diagnostic evaluation of language-guided PPTX editing. The table reports intermediate intent and object-binding accuracy, while the plots show editing success rates under Gold intent, Gold object bindings, and both interventions.

against the benchmark annotations and use Gold interventions to examine how these stages affect editing outcomes.

5.3 Stage-Wise Diagnosis with SlideDiag

Figure 4 provides complementary evidence from intermediate predictions and Gold interventions. Across four backbones, intent accuracy reaches 93.8–95.1%, whereas object-binding exact match is lower at 85.9–88.4%. Replacing predicted bindings with Gold bindings improves editing SR by 8.6 percentage points on average, compared with 4.0 points for Gold intent. The lower binding accuracy and greater recovery jointly support object grounding as a key bottleneck in SlideDiag: errors in recovering complete object membership and editing roles account for substantial performance loss beyond operation understanding. Providing both Gold intent and bindings raises SR to 97.8–98.5%, showing that most edits succeed when the operation and object scope are correctly specified. Residual failures show that execution errors remain even with correct intent and bindings. Thus, grounding is a major source of recoverable error, while code generation and execution contribute additional failures.

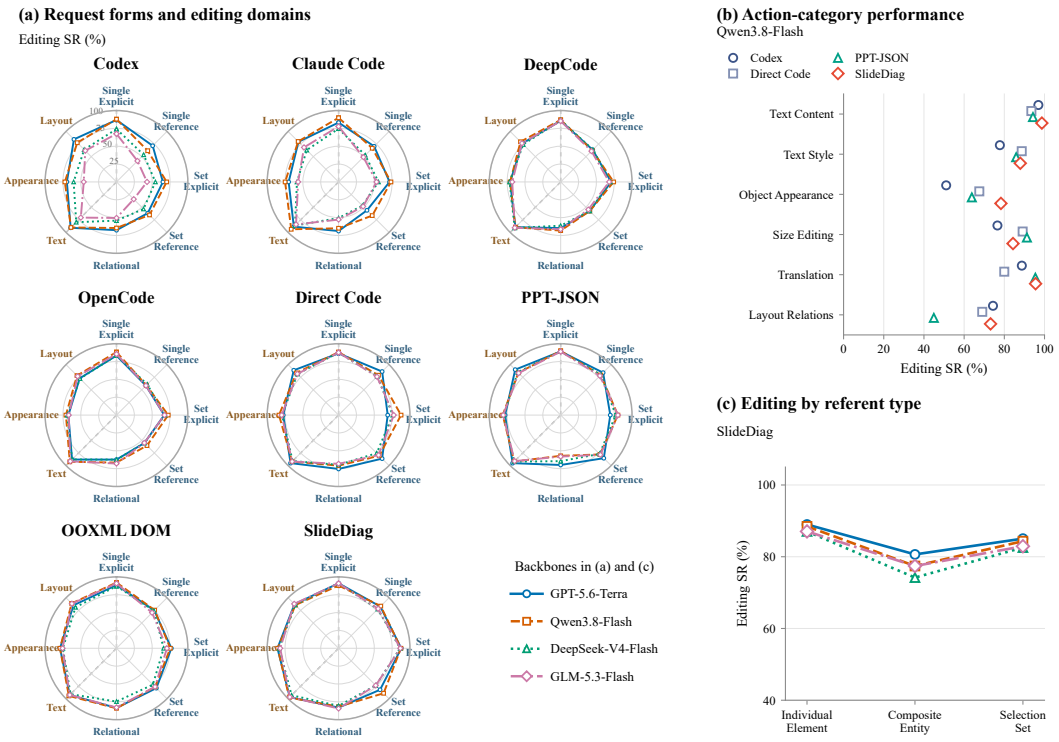

Figure 5: Fine-grained editing success rates across editing actions and request forms.

5.4 Performance across Request Forms and Referent Granularities

Figure 5 examines performance across request forms, editing domains, action categories, and referent types. Reference-based set editing is particularly difficult for coding agents: across the four agents and four backbones in panel (a), Set Reference trails Single Explicit by 20.46–33.20 percentage points. This consistent gap shows that strong performance on explicit single-target edits does not extend uniformly to requests combining target-set selection with reference-based modification. Action categories also differ in how strongly they distinguish editing systems. With Qwen3.8-Flash as the shared backbone, Layout Relations shows the largest spread across the four systems in panel (b), ranging from 44.92% to 74.36%, compared with 93.46–98.68% for Text Content. Geometric editing therefore reveals substantially greater system-level differences than text-content updates, suggesting that system design matters particularly for satisfying positional and geometric constraints. Across all four SlideDiag backbones in panel (c), SR is highest for Individual Element, followed by Selection Set and Composite Entity. Composite Entity trails Individual Element by 8.34–12.87 percentage points. Its consistently lower SR, including relative to Selection Set, shows that successful editing of condition-defined collections does not imply equally reliable editing of coherent content units. This further supports object grounding as a key bottleneck in SlideDiag.

6 Conclusion

In this paper, we introduce PPTXEdit-Bench, a diagnostic benchmark that links natural-language editing instructions to complete object membership, editing roles, and verifiable outcomes. By evaluating object grounding separately from editing correctness, it supports diagnosis of failures that final success rates alone cannot explain. To support this diagnosis, we develop SlideDiag, a baseline that separates intent parsing, object grounding, and code execution. Across four SlideDiag backbones, correcting object bindings yields greater performance recovery than correcting intent. Composite entities and reference-based set edits remain challenging. These findings highlight complete, role-aware object grounding as a priority for translating user requests into precise edits while preserving unaffected state.

REFERENCES

Alibaba Cloud. Qwen3.8-Flash. Model Studio documentation, 2026. URL https://www.alibabacloud.com/help/en/model-studio/qwen3-8-flash.

Anomaly. OpenCode. GitHub repository, 2025. URL https://github.com/anomalyco/opencode.

Anthropic. Claude Code. GitHub repository, 2025. URL https://github.com/anthropics/claude-code.

Michael Bolin. Unrolling the Codex agent loop. OpenAI Engineering, 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.505. URL https://aclanthology.org/2024.acl-long.505/.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. arXiv preprint arXiv:2606.19348, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2306.06070.

Tsu-Jui Fu, William Yang Wang, Daniel McDuff, and Yale Song. DOC2PPT: Automatic presentation slides generation from scientific documents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 634–642, 2022. doi: 10.1609/aaai.v36i1.19943. URL https://ojs.aaai.org/index.php/AAAI/article/view/19943.

Apurva Gandhi, Vishwas Suryanarayanan, Raja Hasnain Anwar, Firoz Shaik, Shubhang Desai, Thong Q. Nguyen, Muhammad Taqi Raza, Vishal Chowdhary, and Graham Neubig. PPT-Eval: A benchmark for computer-use agents on PowerPoint tasks. In *Forty-third International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026. URL https://microsoft.github.io/ppteval/.

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. AutoPresent: Designing structured visuals from scratch. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2902–2911, June 2025. doi: 10.1109/CVPR52734.2025.00276. URL https://openaccess.thecvf.com/content/CVPR2025/html/Ge_AutoPresent_Designing_Structured_Visuals_from_Scratch_CVPR_2025_paper.html.

Gemma Team. Gemma 4 technical report. arXiv preprint arXiv:2607.02770, 2026.

Google DeepMind. Gemini 3.7 Flash model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-7-flash/.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.05243.

Yiduo Guo, Zekai Zhang, Yaobo Liang, Dongyan Zhao, and Nan Duan. PPTC benchmark: Evaluating large language models for PowerPoint task completion. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 8682–8701, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.514. URL https://aclanthology.org/2024.findings-acl.514/.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. Cogagent: A visual language model for gui agents. *arXiv preprint arXiv:2312.08914*, 2023. URL https://arxiv.org/abs/2312.08914.

Zheng Huang, Xukai Liu, Tianyu Hu, Kai Zhang, and Ye Liu. PPTBench: Towards holistic evaluation of large language models for PowerPoint layout and design understanding, 2025. URL https://arxiv.org/abs/2512.02624.

Fengxian Ji, Jingpu Yang, Zirui Song, Yuanxi Wang, Zhexuan Cui, Yuke Li, Qian Jiang, and Xiuying Chen. FineState-Bench: Benchmarking state-conditioned grounding for fine-grained GUI state setting. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 43073–43088, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.2136. URL https://aclanthology.org/2026.findings-acl.2136/.

Kyudan Jung, Hojun Cho, Jooyeol Yun, Soyoung Yang, Jaehyeok Jang, and Jaegul Choo. Talk to your slides: High-efficiency slide editing via language-driven structured data manipulation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 3370–3399, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.166. URL https://aclanthology.org/2026.findings-acl.166/.

Da-Wei Li, Danqing Huang, Tingting Ma, and Chin-Yew Lin. Towards Topic-Aware slide generation for academic papers with unsupervised mutual learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 13243–13251, 2021. doi: 10.1609/aaai.v35i15.17564. URL https://ojs.aaai.org/index.php/AAAI/article/view/17564.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. DeepCode: Open agentic coding. arXiv preprint arXiv:2512.07921, 2025.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for gui visual agent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025. URL https://arxiv.org/abs/2411.17465.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2365. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-Datasets_and_Benchmarks_Track.html.

Michael Ofengenden, Yunze Man, Ziqi Pang, Liang-Yan Gui, and Yu-Xiong Wang. PPTArena: A benchmark for PowerPoint editing, 2025. URL https://arxiv.org/abs/2512.03042.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. arXiv preprint arXiv:2508.10925, 2025.

OpenAI. Google Slides. OpenAI Plugins, Google Slides skill, 2026a. URL https://github.com/openai/plugins/blob/main/plugins/google-drive/skills/google-slides/SKILL.md.

OpenAI. GPT-5.6 system card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6.

Qwen Team. Qwen3.8-27B. Hugging Face model card, 2026. URL https://huggingface.co/Qwen/Qwen3.8-27B.

Edward Sun, Yufang Hou, Dakuo Wang, Yunfeng Zhang, and Nancy X. R. Wang. D2S: Document-to-slide generation via query-based text summarization. In *Proceedings of the 2021*

Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1405–1418. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.naacl-main.111. URL https://aclanthology.org/2021.naacl-main.111/.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. OfficeBench: Benchmarking language agents across multiple applications for office automation, 2024. URL https://arxiv.org/abs/2407.19056.

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. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1650. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Z.ai. GLM-5.3-Flash. Hugging Face model card, 2026. URL https://huggingface.co/zai-org/GLM-5.3-Flash.

Zekai Zhang, Yiduo Guo, Yaobo Liang, Dongyan Zhao, and Nan Duan. PPTC-R benchmark: Towards evaluating the robustness of large language models for PowerPoint task completion. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 12387–12402, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.722. URL https://aclanthology.org/2024.findings-emnlp.722/.

Hao Zheng, Xinyan Guan, Hao Kong, Wenkai Zhang, Jia Zheng, Weixiang Zhou, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. PPTAgent: Generating and evaluating presentations beyond text-to-slides. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 14402–14418, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.728. URL https://aclanthology.org/2025.emnlp-main.728/.

Kun Zhou, Jiakai He, Wenmian Yang, Zhensheng Wang, Yiquan Zhang, and Weijia Jia. Automatic slide updating with user-defined dynamic templates and natural language instructions. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 18263–18297, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.909. URL https://aclanthology.org/2026.findings-acl.909/.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2307.13854.

USAGE OF LLMS

LLMs were used to generate source presentations, paraphrase editing-instruction templates, and power the editing systems evaluated in this work. In SlideDiag, they performed intent parsing, role-aware object grounding, and editing-code generation. These uses are described in the corresponding methodology and experimental sections.

LLMs also assisted with manuscript preparation, including language refinement, terminology consistency, and clarification of technical descriptions. The authors are responsible for verifying LLM-assisted content and retain full responsibility for the research design, data quality, experimental analysis, conclusions, and final manuscript.

A LIMITATIONS

PPTXEdit-Bench is designed to evaluate whether editing systems can translate user instructions into accurate modifications of native PowerPoint objects. Its construction prioritizes precise annotations, verifiable outcomes, and common editing needs. The following limitations concern the coverage of source presentations and instructions, the scope of editing tasks, and the applicability of stage-wise diagnosis.

First, the source presentations are generated by AI systems rather than collected from human-authored decks. This setting follows the workflow in which users obtain an initial presentation from an AI system and refine it through subsequent editing requests. We use multiple generation systems across eight application scenarios to introduce variation in layouts and native-object organization. The evaluation focuses on editing these existing objects, including recovering complete membership, assigning editing roles, and preserving unaffected properties. Human-authored presentations offer a complementary source for extending this coverage.

Second, editing instructions are constructed from predefined templates and LLM-generated paraphrases rather than collected from user editing sessions. This choice enables precise alignment between natural-language requests and their action, argument, and object annotations. Such alignment supports both end-to-end evaluation and analysis of intermediate predictions: an unsuccessful edit can be examined in terms of the interpreted operation, selected objects, and executed changes. Paraphrases introduce lexical and syntactic variation while retaining the same editing requirements. The current instruction set therefore prioritizes controlled task coverage and verifiable ground truth; naturally occurring requests would provide an additional setting for evaluating linguistic robustness.

Third, the benchmark focuses on deterministic editing requirements. This scope allows the expected object-property values to be established in Gold PPTX files and supports reproducible evaluation across editing systems. It does not include subjective or underspecified requests such as "make this slide more professional," which may require clarification or admit multiple acceptable designs. Evaluating these requests would involve user preferences and judgments of visual quality in addition to editing correctness. We leave this broader setting to future work and focus here on the precise execution of specified changes to the intended objects.

Fourth, the benchmark primarily covers slide elements and combinations associated with common user editing needs, including text boxes, shapes, pictures, charts, tables, and composite units such as information cards and text–image blocks. This selection prioritizes practical editing scenarios rather than exhaustive coverage of every possible PowerPoint element combination. These elements support different editing scopes, from modifying individual objects to operating on complete content units and condition-defined object sets.

Finally, the stage-wise analysis is conducted through SlideDiag, which exposes intent, role-aware object bindings, and editing code as separate intermediate outputs. These interfaces support direct comparison with annotations and controlled replacement of predictions with Gold information. The resulting diagnosis characterizes the evaluated pipeline and backbones, while other editing systems may organize or represent these decisions differently. The final editing-success criterion applies across the evaluated systems; extending intermediate diagnosis to other architectures would require corresponding mappings between their internal decisions and the benchmark annotations.

B DATASET CONSTRUCTION DETAILS

This appendix details the construction of PPTXEdit-Bench, including source-slide generation, referent annotation, referring-expression construction, and the generation of editing instructions and Gold PPTX files. We also provide the action specifications, annotation procedures, and dataset statistics supporting the main text.

B.1 SOURCE SLIDE GENERATION

This section provides additional details on generation requests, generation systems, and slide filtering. We organize content briefs by application scenario, instantiate a shared prompt template with

Source Generation Prompt

Create a clear, professional, and editable PowerPoint presentation based on the following brief.

01 Presentation brief

Topic	`{topic}`
Target audience	`{audience}`
Communication goal	`{goal}`
Required content	`{required_content}`

02 Content and slide organization

Organize the slide sequence and layouts around the content, highlighting the main message of each slide. Use text, images, charts, tables, and diagrams as appropriate for the audience and communication goal.

03 Visual relationships and consistency

Group related information together. When presenting multiple items or viewpoints, consider side-by-side comparisons, categorized displays, or process illustrations. Use headings, alignment, spacing, and color to communicate relationships. Maintain a consistent visual style throughout the presentation while varying the layouts.

04 Editability and output

Keep text, shapes, images, tables, and charts editable. Do not flatten entire slides into images. Ensure that text is legible and layouts are complete, and output a PPTX file.

Figure 6: Presentation-generation prompt template. Placeholders are populated with the topic, target audience, communication goal, and required content from each generation request.

each brief, and use different systems to generate native PPTX files. We then select pages suitable for object annotation and edit-instance construction to form the source-slide collection.

Generation Requests. We design generation requests across eight application scenarios: academic research, technical systems, teaching, project progress, product proposals, business decisions, data analysis, and organizational operations. Each request specifies a topic, a target audience, a communication goal, and required content, turning an application scenario into a concrete presentation-generation task. For example, technical-system requests cover architecture design, security controls, and platform migration, while data-analysis requests cover retention analysis, demand forecasting, and experimental effect analysis. We insert the request fields into a generation prompt template (Figure 6) that requires editable native objects, including text, shapes, images, tables, and charts as appropriate, rather than slides flattened into images. The prompt provides requirements for content organization and visual quality, while leaving specific page layouts and object organization to the generation system.

Generation Systems. We use PPTAgent (Zheng et al., 2025), AutoPresent (Ge et al., 2025), Presenton, Codex, and Claude Code to generate source presentations, introducing variation in page layouts and object organization across systems. Codex and Claude Code each use a fixed PPTX-generation toolchain. The same generation request can be used by different systems, so request counts and presentation counts are reported separately. Table 2 summarizes the presentation counts for each system.

Slide Filtering. We inspect the generated presentations for file validity, visual presentation, and native-object structure. First, we parse each PPTX to check whether its slides and native objects can be accessed. We then render the pages to inspect missing resources, content overflow, and occlusion that impairs comprehension, and verify the correspondence between visible content and native objects. We also inspect object types and accessible properties to assess whether the pages support subsequent native-object editing.

Table 2: Presentation counts by generation system.

Generation system	Presentations
PPTAgent	133
AutoPresent	132
Presenton	132
Codex	300
Claude Code	300
Total	**997**

Table 3: Individual elements and editable properties.

Element type	Editable properties
Text boxes and text placeholders	Text content, font name, font size, font color, bold, italic, underline, paragraph alignment, position, and size.
Text-bearing shapes	Text content, font name, font size, font color, bold, italic, underline, paragraph alignment, fill color, border color, border width, position, size, and rotation.
Non-text shapes	Fill color, border color, border width, position, size, and rotation.
Pictures	Position and size.
Lines and arrows	Position, size, line color, line width, and dash style.
Charts	Whole-object position and size.
Tables	Whole-object position and size.

We prioritize slides with diverse element types and richer object organization, including combinations of text, images, charts, and tables, as well as information cards, parallel text–image modules, and process diagrams composed of multiple elements. This selection provides suitable objects for different editing tasks, including moving complete content units, modifying properties across object sets, matching dimensions to reference objects, and aligning multiple objects. The applicability of a specific editing action is determined during subsequent referent annotation and edit-instance construction.

We additionally check for duplicate pages and highly repetitive page structures to reduce redundancy in the source slides. We retain 4,420 source slides, each linked to its original presentation and slide index, for subsequent referent annotation and edit-instance construction.

B.2 Basic Elements and Editable Properties

Individual elements are native objects that can be independently identified and accessed in a PPTX file, including text boxes, text placeholders, shapes, pictures, lines, charts, and tables. We record each element's object identifier to link it to natural-language descriptions. A shape containing text is treated as a single element, with its text content, formatting, and shape appearance belonging to the same object.

Table 3 summarizes the editable properties of each element type. Text boxes and text-bearing shapes support changes to text content and formatting, while shapes additionally support fill and border changes. Position and size properties support movement, size matching, alignment, and spacing operations. The specific operations used in editing instances are defined in the action library (Appendix B.5).

This benchmark treats charts and tables as whole objects whose position and size can be modified. Arrows are categorized according to their native representation in the PPTX file: arrow-shaped AutoShapes follow the shape properties, whereas arrows stored as connectors follow the line properties. During edit-instance construction, we select applicable actions based on object types and their supported properties.

Table 4: Selection-set construction rules and examples.

Selection basis	Membership criterion	Example
Object type	All objects of a specified type on the slide.	All images on the slide.
Parent entity	All members of an accepted entity, or all its members of a specified type.	All text boxes in the second card.
Shared attribute	All objects of the same type with a specified attribute value.	All blue-filled shapes on the slide.

B.3 Annotation Protocol

We combine automatic annotation with human review to organize native slide objects into composite entities and selection sets. The program generates candidates from native grouping, slide layouts, and object attributes. Annotators then assess the candidates, check membership completeness, and add missing entities. Each annotation records its referent type, native-object members, and the structural evidence or selection condition used to identify them. These records support subsequent referring-expression generation and edit-instance construction.

Automatic Annotation of Composite Entities. We generate composite-entity candidates using native grouping, background containment, and local layout relations. For explicitly grouped objects, the program extracts the constituent individual elements as candidate members without counting the group container as an additional member.

For content units without explicit grouping, the program examines background shapes, distances between objects, and alignment to identify elements that may belong to the same unit. For example, icons and text boxes on a shared background can form an information-card candidate, while an image and nearby aligned text can form an image–text candidate. The background shape is retained as a member so that the complete unit can be edited together. The program also checks existing background assignments to reduce associations across different content units.

Candidates with identical membership are merged and submitted for human review. Both native groups and inferred entities require annotator confirmation, and annotators can add content units missed by the program.

Automatic Annotation of Selection Sets. We construct selection sets using object type, parent-entity membership, and shared attributes, recording each selection condition and all matching objects. Table 4 summarizes the construction rules.

Shared attributes include fill color, font size, font color, and bold formatting. Matching uses explicit attribute values extracted from native objects. Color strings are normalized to the same case, and font sizes are rounded to two decimal places. If a text attribute varies within an object, that object is excluded from set construction based on that attribute.

Each set retains the complete membership matching its condition, without removing individual matches to narrow its scope. Candidates with identical membership, entity dependencies, and review status are merged, while equivalent selection conditions are retained. Sets derived from composite entities are updated when the entity membership is corrected.

Annotation Platform and Human Review. We develop a browser-based annotation platform to support the construction and quality control of PPTXEdit-Bench (Figure 7). The platform links PowerPoint-rendered slides with native PPTX objects and supports slide-structure inspection, object-membership review, annotation-state management, and result validation. The left panel displays the slide queue and review progress, the central panel shows the rendered slide with interactive native-object bounding boxes, and the right panel presents candidate annotations, selection conditions, review states, and member lists.

Annotators can inspect a candidate structure by highlighting all of its members on the rendered slide. They can add or remove individual objects by clicking their bounding boxes, select multiple objects by dragging a region, or directly modify membership through the object list. The interface

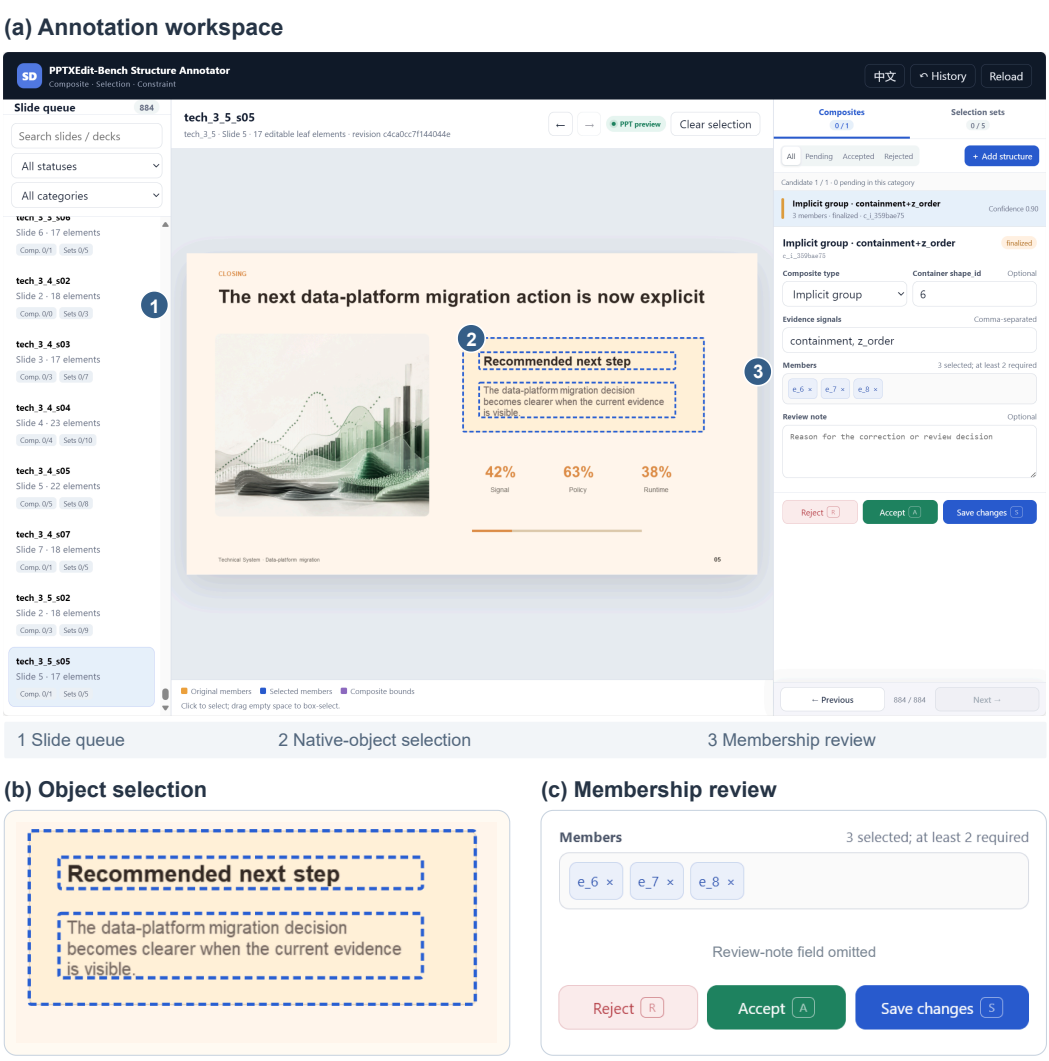

Figure 7: Annotation interface for PPTXEdit-Bench. (a) Full workspace with the slide queue (1), native-object selection (2), and membership review (3). Enlarged views show (b) object selection and (c) membership and review controls; the blank review-note field is omitted.

supports four review operations: accepting a correct candidate, correcting its membership, rejecting an invalid candidate, and adding an omitted composite entity. Distinct visual indicators distinguish the original candidate membership from the current selection, while every selected object remains linked to its native PPTX identifier.

The tool enforces dependencies between composite entities and selection sets. Composite entities must be reviewed first because their members may subsequently appear in selection sets. When the membership or review state of a composite entity changes, the tool regenerates the selection sets that depend on it. A selection set can be accepted only after all composite entities referenced by it have been accepted. To prevent inconsistent updates, each modification is validated before being written, saved atomically, and accompanied by a recoverable version snapshot. The tool also prevents an outdated browser session from overwriting a newer data version.

Eight annotators with experience in presentation authoring and PowerPoint editing used the tool to review all 4,420 source slides. Before formal annotation, all annotators received the same written annotation guidelines, which defined composite entities and selection sets, and completed a pilot round on the same 30 representative slides. The pilot examples covered correct candidates, incom-

Table 5: Descriptive cues and example referring expressions for the three referent types.

Referent type	Descriptive cues	Example
Individual element	Name and position	The image on the right.
Individual element	Name and visible text	The text box containing "Revenue."
Individual element	Name and containing unit	The icon in the second card.
Composite entity	Whole-unit name and order	The second metric card from the left.
Composite entity	Whole-unit name and member content	The information card containing "Revenue."
Selection set	Member type and scope	All images on the slide.
Selection set	Member type and containing unit	All text boxes in the second metric card.
Selection set	Member type, attribute, and scope	All blue-filled shapes on the slide.

plete memberships, invalid groupings, and omitted composite entities. The annotators discussed disagreements arising during the pilot round to align their annotation criteria.

All slides were divided approximately evenly among the eight annotators. Each slide underwent two complete review passes conducted by different annotators: a primary annotation pass and a full secondary review. Each annotator therefore reviewed approximately 552–553 slides in each role, corresponding to approximately 1,105 slide-level review assignments in total. For composite entities, annotators determined whether the members formed a complete visual or semantic content unit. For selection sets, they verified whether the membership included all objects satisfying the specified condition within the stated scope. During the second review pass, the reviewer re-examined each candidate structure, its review decision, and its complete native-object membership against the rendered slide.

Disagreements between the two review passes were referred to the authors for adjudication and resolved before the annotations were finalized. After human review, automated consistency checks verified native-object identifiers, membership validity, dependencies between composite entities and selection sets, and duplicate records. Annotations that failed any check were returned for correction. Reviewer information and intermediate review states were excluded from the released data; only accepted or corrected annotations that passed both rounds of human review and automated validation were used for subsequent referring-expression generation and edit-instance construction.

B.4 Referring Expression Construction and Verification

We construct natural-language expressions for verified referents, linking user descriptions to their corresponding native objects. The process consists of choosing object names, incorporating slide information into specific descriptions, and checking whether the descriptions accurately identify the intended objects. We retain at least three distinct expressions per referent for subsequent editing-instruction generation.

Object Names and Descriptive Cues. Object names indicate what the referent is, while additional descriptive information identifies which object or objects are intended. We design names and expression templates according to the characteristics of the three referent types, with examples in Table 5.

For individual elements, we use type names such as "text box," "image," and "rectangle," or function names such as "title" when supported by the slide content. These names are qualified by visible text, appearance, position, or the containing content unit. For composite entities, we use whole-unit names such as "card," "panel," and "image–text block," or functional names such as "metric card" and "process step." Member content, position, and ordinal order distinguish the intended entity. For selection sets, we use plural member names and specify the selection scope and conditions.

Names are drawn from a predefined lexicon, and templates organize the names and their qualifying information. Name specificity depends on the slide content: a semantic name is used when the function can be identified; otherwise, a generic name is used. For example, "metric card" is used when the content supports that function, and "card" otherwise.

Table 6: Action categories and their object-role requirements.

Category	Count	Modification and object roles
Explicit-value	11	Modify targets using supplied text, property values, or displacement arguments.
Reference-based	6	Read a property value from a reference object and apply it to the targets.
Relational	10	Adjust participants to establish alignment, equal spacing, or equal dimensions.

Expression Generation from Slide Information. For each annotated referent, we select applicable names and templates based on its type, member composition, and slide information. We then populate the templates with visible text, appearance, position, ordinal order, or the containing content unit. Spatial descriptions follow the rendered slide, while ordinal descriptions specify the comparison scope and ordering direction, as in "the second metric card from the left."

Expressions can also describe objects within a content unit. For example, "the second metric card from the left" refers to the complete card membership. Using this expression to specify the selection scope produces "all text boxes in the second metric card from the left," which refers to every text box within that card. The two expressions share descriptive information but identify different sets of native objects.

Expression Verification and Retention. We check candidate expressions against the source slide and membership annotations to confirm that their descriptive cues are visible, that no alternative reasonable referent exists, and that they cover the complete intended membership. An individual-element expression must identify the corresponding element, a composite-entity expression must identify the complete content unit, and a selection-set expression must include all objects satisfying the condition within the specified scope.

Candidates with insufficient descriptive information are revised using additional or alternative visible cues and checked again. Expressions that remain ambiguous are discarded. If verification reveals a problem in the membership annotation itself, the annotation is returned to the review procedure (Appendix B.3).

After deduplication, we retain at least three valid expressions per referent, favoring variations in descriptive cues or wording. Expressions do not expose internal PPTX object identifiers but remain linked to the complete native-object membership. During subsequent edit-instance construction, these expressions describe the participating objects, while the associated membership annotations identify the native objects involved.

B.5 Edit Action Library

We design an edit action library that specifies the participating objects, required arguments, validity conditions, and property updates for each operation. The complete set of 27 actions and their instance counts is shown in Figure 2(a). This appendix provides the operational specifications shared by instruction generation, Gold PPTX construction, and evaluation.

Action Categories and Object Roles. We organize actions into explicit-value, reference-based, and relational categories according to how the modification is specified. An action can apply to different object types that support the required properties. For example, both pictures and shapes can participate in width matching without requiring separate operations.

During instance construction, we select compatible referents according to the action requirements and assign them roles as modification targets, reference objects, or relational participants.

Parameter Generation and Filtering. We define parameter dictionaries for actions requiring explicit values and select applicable candidates based on the objects' current state. Text and font-size candidates are organized by function, including titles, subtitles, body text, and labels. Color names are paired with exact RGB values for text, fills, and lines. Dimension and line-width candidates are

Table 7: Parameter pools and candidate-generation rules. Font sizes, distances, dimensions, and line widths are expressed in points (pt).

Parameter	Candidate pool or generation rule
Replacement text	Phrase pools for titles, subtitles, section labels, ordinary labels, and body text; examples include "Key findings," "Updated analysis," and "Recommended action."
Text-span replacement	Identify a year or percentage in the target text and increment it by one year or one percentage point. For multiple targets, the original span must occur in every target.
Title font size	28, 32, 36.
Subtitle font size	18, 20, 22, 24.
Body font size	14, 16, 18, 20.
Label font size	10, 11, 12, 14, 16; also used for section labels.
Generic font size	12, 14, 16, 18, 20, 24, 28, 32, 36.
Color	Eight font-color candidates, ten fill-color candidates, and five line-color candidates, each paired with an RGB value.
Bold formatting	`true` or `false`.
Displacement	Left, right, up, or down by 6, 8, or 12.
Width and height	Multiply the original dimension by 0.85 or 1.15, round to the nearest 0.5-point increment, and require a change of at least 4 points.
Line width	Generate a final value by increasing the original width by 0.75, 1.5, or 2.25; require a final width of at most 8 and a change of at least 0.5.

derived from current values so that the modification accounts for the object's initial state. Table 7 summarizes the candidate pools and generation rules.

We filter generated candidates using object properties and slide boundaries, excluding candidates that produce no effective change, move objects beyond the slide boundaries, or fail the approximate text-fit check. The line-width increase action records the final width: for example, increasing a line from 1 pt to 2.5 pt uses an argument value of 2.5 pt.

Action–Object Compatibility. Action applicability depends on whether the participating objects support the required property updates (Appendix B.2). For individual elements, we check whether the relevant properties are accessible and editable. For set-level property changes, all members must support the operation. Moving a composite entity applies the same displacement to its complete membership, moving its background, icons, and text boxes together. Reference-based actions require a reference object that provides a definite property value, while relational actions require the relevant position and dimension information.

Full-text replacement and direct width or height assignment apply to individual elements. Other text and appearance actions apply to individual elements and selection sets where the required properties are supported. Alignment and equal-size operations require at least two participants, while equal-spacing operations require at least three. Equal-width and equal-height operations adjust individual native elements.

Execution Policies for Relational Actions. For alignment, equal-spacing, and equal-size actions, we define deterministic execution policies (Table 8). These policies compute the resulting positions or dimensions from the participating objects' original geometry and specify which geometric properties remain unchanged. We apply these policies when constructing the Gold PPTX files.

After compatibility and parameter checks, we record the selected action, participating objects and their roles, and arguments. Instruction templates express this information in natural language, while execution functions apply the corresponding modification to a copy of the source file to produce the Gold PPTX.

Table 8: Execution policies used to construct Gold PPTX files for relational actions.

Operation	Target position or dimension	Preserved geometry
Left/right alignment	Use the minimum left edge or maximum right edge among participants, respectively.	Vertical position, width, and height.
Top/bottom alignment	Use the minimum top edge or maximum bottom edge among participants, respectively.	Horizontal position, width, and height.
Horizontal/vertical center alignment	Use the arithmetic mean of the original center coordinates on the corresponding axis.	Position on the other axis, width, and height.
Horizontal/vertical equal spacing	Sort by original position, fix the endpoint objects, and adjust intermediate positions to establish equal edge-to-edge gaps.	Object order, dimensions, and position on the other axis.
Equal width/height	Use the largest original width or height among participants, respectively.	Top-left position and the other dimension.

Table 9: Actions, request forms, and representative instruction templates. Each paraphrase preserves the editing requirements of its corresponding action.

Action	Request form	Canonical template	Paraphrase
Set font size	Single-target explicit	Set the font size of {target} to {value} pt.	Format {target} with a font size of {value} pt.
Match width	Single-target reference-based	Match the width of {target} to that of {reference}.	Copy the width from {reference} to {target}.
Set fill color	Set-level explicit	Change the fill color of {target} to {value}.	Apply {value} as the fill color for {target}.
Match font color	Set-level reference-based	Set the font color of {target} to match that of {reference}.	Copy the font color from {reference} to {target}.
Align left	Multi-object relational	Align the left edges of {target}.	Arrange {target} so their left edges coincide.

B.6 Instruction Templates and Paraphrases

We construct instruction templates from the definitions in the action library, expressing each action, its required object roles, and its arguments in natural language. For each action, we design multiple canonical templates and introduce lexical and syntactic variation through paraphrasing. During edit-instance construction, these templates are populated with object descriptions and argument values, allowing different expressions to convey the same editing requirement.

Template Construction from Action Definitions. Each action specifies the property to modify, the participating objects, and the required arguments. These definitions determine the template structure. For explicit-value actions, templates identify the targets and supplied arguments. For example, `set_font_size` requires target objects and a font-size value, yielding "Set the font size of {target} to {value} pt." For reference-based actions, templates specify which object supplies a property value and which targets receive it. For example, `copy_width` yields "Match the width of {target} to that of {reference}," with the width obtained from the reference object. For relational actions, templates describe the participants and the relation they should satisfy, as in "Align the left edges of {target}" for `align_left`.

Explicit-value and reference-based actions use single-target or set-level expressions according to their applicability, while relational actions use multi-object expressions. Together, these templates cover the five request forms. Table 9 illustrates the correspondence between actions, request forms, and templates.

The {target} and {reference} slots are populated with referring expressions for the corresponding objects, while {value} holds an argument value. In relational templates, {target} describes the objects participating in the operation.

Table 10: An illustrative width-matching instance linking the instruction, participating objects, and Gold modification.

Component	Example
Source state	Left image: 120 pt wide; right image: 90 pt wide.
Action	Width matching (`copy_width`).
Participating objects	Target: right image; reference: left image.
Instruction	Match the width of the right image to that of the left image.
Gold modification	Set the right image's width to 120 pt.
Preserved state	The left image, the right image's position and height, and unrelated content.

Paraphrasing with Preserved Action Semantics. We design 4–16 canonical templates per action, totaling 166 templates, and use a large language model to generate three paraphrases for each template. Paraphrases vary wording and sentence structure while preserving the object roles, arguments, and direction of modification required by the action. For example, a `copy_width` template may mention either the target or the reference first, but both formulations require applying the reference object's width to the target.

We check templates and paraphrases against their corresponding action definitions. Text replacement must distinguish the original and replacement strings; movement must preserve direction and distance; reference-based operations must preserve the direction of property transfer; and set-level operations must preserve the target scope. Automated checks additionally reject templates with missing or additional argument or object-description slots and remove case-insensitive duplicates within each action and request form. The resulting library contains 166 canonical templates and 498 paraphrases, totaling 664 template variants.

Template Instantiation. During edit-instance construction, we select a template according to the chosen action and request form, then populate it with role-specific object descriptions and argument values. For example, for `set_font_size`, inserting "the title on the right" and "32" produces "Set the font size of the title on the right to 32 pt." The instruction and Gold PPTX share the same action, native target objects, and arguments; template variation changes only how the editing requirement is expressed.

B.7 Instruction and Gold PPTX Generation

We generate editing instructions and Gold PPTX files from the same actions, participating objects, and arguments, then check the resulting modifications. Five independent instances are constructed per source slide, with each edit starting from a copy of the original presentation. Each completed instance contains the source PPTX, a natural-language instruction, action and object annotations, and a Gold PPTX.

Editing Instruction Generation. We select an instruction template according to the action and request form, then populate it with natural-language descriptions of the participating objects and argument values. These descriptions are linked to native objects through existing membership annotations, identifying the modification targets, reference objects, and other participants in the source file.

For example, consider a slide containing two images: a left image with a width of 120 pt and a right image with a width of 90 pt. Selecting `copy_width`, with the right image as the target and the left image as the reference, produces "Match the width of the right image to that of the left image." Table 10 shows the instance components and corresponding modification.

Different templates can express the same instance. For example, "Copy the width from the left image to the right image" corresponds to the same action, participating objects, and modification.

Gold PPTX Generation. The execution function uses the same editing information as the instruction to locate participating objects in a copy of the source file and update the relevant properties. Explicit-value actions apply supplied arguments, reference-based actions read property values from

Table 11: Overview of PPTXEdit-Bench.

Statistic	Count
Source presentations	997
Source slides	4,420
Application scenarios	8
Annotated referents	94,096
Editing actions	27
Request forms	5
Editing instances	22,100
Instances per source slide	5

Table 12: Dataset splits. All slides and derived instances from the same presentation belong to a single split.

Split	Source slides	Editing instances	Instance share
Training	3,094	15,470	70%
Validation	442	2,210	10%
Test	884	4,420	20%
Total	**4,420**	**22,100**	**100%**

reference objects, and relational actions compute target positions or dimensions using shared execution policies (Table 8). In the width-matching example, the function reads the left image's width and sets the right image's width to 120 pt.

For composite entities, the complete membership annotation determines the execution scope. For example, "Move the second metric card 6 pt to the right" refers to a background shape, an icon, and two text boxes. Execution applies the same horizontal displacement to all four members, moving the card as a whole while preserving its internal arrangement.

Relational edits follow the policy associated with each action. For example, if three images have original left-edge coordinates of 100, 110, and 120 pt, left alignment places all three left edges at 100 pt while preserving their vertical positions and dimensions. After execution, the modified presentation is saved as the Gold PPTX.

B.8 Dataset Statistics

PPTXEdit-Bench contains 4,420 source slides from 997 presentations across eight application scenarios. The dataset includes 94,096 annotated referents and 22,100 editing instances covering 27 actions, with five instances per source slide. Table 11 summarizes the overall scale.

Dataset Splits. We partition the dataset at the presentation level, assigning all slides from a presentation and their derived editing instances to the same split. The training, validation, and test sets contain 15,470, 2,210, and 4,420 editing instances, respectively, accounting for 70%, 10%, and 20% of the dataset. Table 12 reports the corresponding counts.

Referent Distribution. Annotated referents comprise individual elements, composite entities, and selection sets, with counts and proportions reported in Table 13. Individual elements account for 55.92%, while composite entities and selection sets together account for 44.08%. These categories cover references to individual elements, complete content units, and objects selected through conditions.

These statistics count referent annotations. A native object may occur in multiple annotations: for example, a text box may be annotated individually, belong to a metric card, and also belong to the selection set described as "all text boxes in that card."

Request-Form Distribution. Editing instances cover five request forms, as shown in Table 14. Single-target explicit editing accounts for 46.77% of instances. The remaining four forms account

Table 13: Distribution of the 94,096 annotated referents.

Referent type	Annotations	Share
Individual element	52,619	55.92%
Composite entity	10,787	11.46%
Selection set	30,690	32.62%
Total	**94,096**	**100%**

Table 14: Distribution of the 22,100 editing instances across request forms.

Request form	Editing instances	Share
Single-target explicit	10,337	46.77%
Single-target reference-based	4,244	19.20%
Set-level explicit	3,469	15.70%
Set-level reference-based	2,793	12.64%
Multi-object relational	1,257	5.69%
Total	**22,100**	**100%**

for 53.23% and involve reference-property retrieval, set-level modification, or relational adjustment among multiple objects.

Request forms are counted over editing instances, while referent types are counted over object annotations. For example, moving one information card composed of multiple native objects is a single-target edit whose referent type is composite entity.

C Method Implementation Details

SlideDiag performs editing through three modules: edit-intent parsing, role-aware object grounding, and code generation and execution. Each module makes a separate language-model call and passes its output to the next module. We use the instruction "Make the remaining images as wide as the image on the left" to illustrate their inputs, outputs, and prompt design.

C.1 Edit-Intent Parsing

This module extracts the editing action, arguments, and role-specific object descriptions from the user instruction. Its inputs are the instruction and an action catalog specifying action names, argument definitions, and role requirements. The output retains natural-language object descriptions, which are resolved to native objects by the grounding module.

For the running example, the expected output is:

```
{
  "action": "copy_width",
  "arguments": {},
  "roles": {
    "targets": ["the remaining images"],
    "reference": ["the image on the left"]
  }
}
```

The action `copy_width` reads the width from the reference object, so no explicit width argument is required. The `targets` and `reference` fields contain descriptions of the targets and reference object, respectively. Relational actions, including alignment, distribution, and equal-size operations, use `participants` to record the participating object descriptions.

Intent Parsing Prompt

01 Output format

Return exactly one JSON object with three fields:

action: the selected action name from the supplied catalog.
arguments: an object following the action schema; use {} when no arguments are required.
roles: only the roles required by the selected action, each mapped to an array of natural-language object descriptions.

Output no prose, Markdown, or additional fields.

02 Action selection and object roles

Select the action matching the instruction from the supplied catalog. Use targets for explicit edits, targets and reference for copy actions, and participants for relational actions such as alignment, equal spacing, and equal dimensions.

Preserve object descriptions, scope, ordering, relative relations, and exclusions. Keep collection descriptions intact; do not enumerate implicit members or invent object IDs. Grounding resolves the objects and their membership later.

03 Argument extraction

Follow the selected action schema. Extract only arguments explicitly specified in the instruction; do not guess missing values.

Express dimensions, displacements, line widths, and font sizes in points (pt). Preserve literal strings and the boolean value false.

Use old/new for replace_text_span and dx_pt/dy_pt for move_by (right/down positive).

Copy and relational actions have empty arguments ({}). Reference properties and layout values are obtained or computed during execution.

For increase_line_width, value denotes the final line width, not an increment.

For explicit colors, use the exact RGB value from the supplied benchmark color dictionary. For color-copy actions, read the corresponding reference property during execution rather than using the dictionary.

Figure 8: System prompt for edit-intent parsing.

The intent-parsing prompt (Figure 8) specifies action selection, argument extraction, role assignment, and the JSON output format. The output is checked for valid action names, required fields, and argument types before being passed to the grounding module.

C.2 Role-Aware Object Grounding

This module combines the parsed intent with source-slide information to resolve each role's descriptions to native objects. Source-object information is extracted directly from the PPTX and includes identifiers, types, text, appearance, positions, dimensions, and native grouping relations.

Suppose the slide contains three images: the leftmost image has identifier `e_3`, and the other two have identifiers `e_7` and `e_9`. The expected grounding output is:

```
{
  "targets": [
    {"element_ids": ["e_7"]},
    {"element_ids": ["e_9"]}
  ],
  "reference": [
    {"element_ids": ["e_3"]}
  ]
}
```

The description "the remaining images" resolves to two targets, while the leftmost image serves as the reference. Each `element_ids` list records the native members of one editing unit. For

Grounding Prompt

01 Output contract

Resolve the intent's descriptions using only the supplied Source slide. Return exactly one JSON object whose top-level keys are precisely the intent role names. Each top-level value is an array. Each array item has ONLY one key: element_ids. One logical object looks like {"element_ids":["e_7"]}; it never contains role names.

02 Complete response examples

Complete explicit-edit response:

```
{"targets":[{"element_ids":["e_7"]}]}.
```

Complete copy response:

```
{"targets":[{"element_ids":["e_7"]}],
"reference":[{"element_ids":["e_3"]}]}.
```

Complete alignment response:

```
{"participants":[{"element_ids":["e_7"]},
{"element_ids":["e_9"]}]}.
```

Do not wrap these complete responses inside another role or add an absent reference role.

03 Object membership and grouping

Use exactly the provided IDs. A logical object may contain several native elements. Keep different cards or other composite participants as separate objects for layout edits; do not flatten them together. A native group can be bound by its group ID. Do not select a group together with its descendants. Resolve implicit composites and selection sets from visible properties and geometry, not from invented annotations.

04 References and attribute interpretation

Exclude a reference from a set if the description says remaining/other. The reference is one logical object. Unknown/inherited attributes are not false or black. Return no prose.

Figure 9: System prompt for role-aware object grounding.

a composite entity, the list contains its complete membership. Two cards, for example, are represented by separate member lists so that subsequent code can compute their positions and move them independently.

The grounding prompt (Figure 9) specifies role assignment, membership organization, and the output format. The module checks object identifiers, duplicate membership, and the participant counts required by the action before passing the result to the code module.

C.3 Code Generation and Execution

This module generates a Python editing function from the parsed intent, object-grounding output, and relevant source-object information. Its inputs also include the slide index and canvas dimensions. The output is a JSON object with a `code` field containing a function with the following interface:

```
def edit(prs, slide, roles, arguments):
    ...
```

Here, `prs` and `slide` denote the presentation and designated slide, `roles` provides the native objects associated with each role, and `arguments` contains the parsed arguments. The model generates the modification logic, while the runtime loads the source file, resolves object identifiers, invokes the function, and saves the edited PPTX.

Code Generation Prompt

01 Output contract

Generate exactly one JSON object {"code":"..."}, containing Python source that defines def edit(prs, slide, roles, arguments). The runtime will call it and save output.pptx.

02 Runtime inputs and edit scope

prs is a python-pptx Presentation, slide is the designated native Slide. roles maps each role to a list of logical objects. Each object is a dict with element_ids, shapes (native python-pptx shape instances), and bounds_pt (page-space x,y,width,height). arguments is the parsed intent arguments. Use only bound objects and their descendants as edit targets.

03 Coordinates and preservation

Group shapes expose .shapes. Native shape coordinates are EMU (12700 EMU per point); child coordinates are in group space. Supplied parent_transform maps local point coordinates to page points and must be respected when editing grouped objects. Preserve all other slides, objects, text and formatting, including inherited values. Do not rebuild the deck.

04 Execution constraints

Use python-pptx, lxml or zipfile if required. Do not read any files besides the provided workspace inputs, use network, environment variables or external directories. The runtime loads/saves the deck; do not call prs.save yourself. Do not invent helper editing APIs. Generate the actual action logic; no deterministic action executor exists.

05 Action semantics

For explicit colors use the exact provided value. For copy actions read the represented reference property. Keep logical composite members together in geometric operations. Alignment: left/top=min edge, right/bottom=max edge, center=mean selected center. Equal width/height=max selected size. Distribution spaces logical bounding boxes evenly between the fixed outer bounds. Never resize objects merely to align them.

Figure 10: System prompt for code generation and execution.

For the width-matching example, the generated code reads the reference image's width and assigns it to both target images. For composite-entity movement and alignment, the code computes displacements from the overall bounds of each editing unit and applies the corresponding changes to its members.

The code-generation prompt (Figure 10) specifies the function interface, available tools, coordinate conventions, and action execution requirements. Generated code is checked for syntax and the required function interface before execution. The system outputs the edited PPTX and retains the intent and grounding predictions for subsequent analysis.

D Experimental Setup and Efficiency Measurement

This appendix describes the experimental setup for PPTXEdit-Bench and the measurement of API calls and processing time.

D.1 Experimental Setup and Evaluation Metric

Software Sources. The official repositories are Codex (`https://github.com/openai/codex`), Claude Code (`https://github.com/anthropics/claude-code`), DeepCode (`https://github.com/HKUDS/DeepCode`), and OpenCode (`https://github.com/anomalyco/opencode`).

Evaluation Data. We evaluate on the test split of PPTXEdit-Bench and use the validation split to select configurations. The test split contains 4,420 editing instances derived from 884 source slides. The dataset is partitioned at the presentation level, so all slides from a presentation and their derived instances belong to the same split. Each instance provides a source PPTX and a natural-language

editing instruction. Each run starts from the original source file and produces an independent output file.

Compared Methods. We compare general-purpose coding agents, one-shot editors, and SlideDiag. The coding agents include Codex, Claude Code, DeepCode, and OpenCode; the one-shot editors include Direct Code, PPT-JSON, and OOXML DOM. SlideDiag sequentially performs edit-intent parsing, role-aware object grounding, and code generation and execution. Its module interfaces and prompts are provided in Appendix C.

Each experimental configuration pairs a model with an editing system. All systems receive the same source PPTX files and editing instructions and produce outputs through their respective processing pipelines. Configurations augmented with PPT skills are reported separately and compared with the corresponding base configurations using the same model and editing system.

Editing Success Rate. Each editing instance has a Gold PPTX that specifies the expected object-property values after the requested edit. We use python-pptx to extract native-object properties from the source, predicted, and Gold presentations. For each instance, the evaluator compares the requested properties of all target objects against their corresponding values in the Gold PPTX. It also checks that the remaining evaluated properties of target and non-target objects are unchanged from the source PPTX. For composite entities, these checks cover every annotated native member. An instance is successful only if the output is parseable and all modification and preservation checks pass. Missing, corrupted, or unparseable outputs are counted as failures. SR is the percentage of evaluated instances that satisfy this criterion.

D.2 API Calls and Processing Time

We measure API calls and processing time for each complete run of an editing instance and aggregate them separately for each model, editing system, and skill configuration. Efficiency measurements are reported alongside SR on the same set of instances.

API Calls. An API call is one model inference request issued during editing. The count includes requests for intermediate predictions and editing code, as well as additional requests for repair and retries. Local file access, object-property extraction, and code execution are not counted as model API calls. A model response containing multiple tool calls still counts as one model request; a subsequent request after tool execution adds another call.

A complete SlideDiag run without retries requires three model requests, corresponding to intent parsing, object grounding, and code generation. Actual call counts are obtained from run records to capture early termination and additional requests. The mean number of API calls is the total number of model requests across all evaluated runs divided by the number of runs.

Processing Time. Processing time is measured from when the system begins processing the source PPTX and editing instruction until the final output file is saved or the run terminates. It includes input parsing, waiting for model responses, tool execution, code repair, and output saving. Slide rendering and output checks performed by the editing method are included; comparisons against the Gold PPTX performed by the independent evaluator after the run are excluded.

For example, if the system requests a code repair after an execution failure, the additional request and its elapsed time are included in the same run. Under concurrent execution, processing time is measured using the start and end timestamps of each individual run.

Aggregation. Mean API calls and mean processing time include both successful and failed runs. Failed runs retain their actual call counts and elapsed time, while timed-out runs are measured until termination. For each configuration, mean processing time is the sum of processing times across all evaluated runs divided by the number of runs, reflecting the actual overhead of processing the same set of editing requests.

E Additional Experimental Results

E.1 API Calls and Processing Time

We report the mean number of API calls and mean processing time per editing instance for base configurations in Table 15 and for coding agents with PPT-specific skills in Table 16. Both metrics follow the measurement procedure in Appendix D.2 and are averaged over all runs, including failures. In the tables, Calls denotes the mean number of model API requests, and Time denotes the mean processing time in seconds.

Table 15: Mean API calls and processing time per editing instance for base configurations.

(a) Mean API calls

Model	Codex	Claude Code	DeepCode	OpenCode	Direct Code	PPT-JSON	OOXML DOM	SlideDiag
Gemma-4-31B	13.49	2.15	5.22	6.11	1.00	1.00	1.00	3.00
Gemma-4-26B-A4B	15.41	16.20	9.30	14.60	1.00	1.00	1.00	3.00
Qwen3.8-27B	14.90	19.50	12.90	18.36	1.00	1.00	1.00	3.00
DeepSeek-V4-Flash	16.23	17.70	7.99	15.17	1.00	1.00	1.00	3.00
GLM-5.3-Flash	6.58	12.50	10.88	11.75	1.00	1.00	1.00	3.00
Gemini-3.7-Flash	4.85	6.30	4.20	6.00	1.00	1.00	1.00	3.00
GPT-5.6-Terra	11.00	14.30	9.40	13.50	1.00	1.00	1.00	3.00
Qwen3.8-Flash	10.96	14.33	17.95	13.50	1.00	1.00	1.00	3.00

(b) Mean processing time (s)

Model	Codex	Claude Code	DeepCode	OpenCode	Direct Code	PPT-JSON	OOXML DOM	SlideDiag
Gemma-4-31B	60.54	15.72	36.29	33.18	10.89	12.58	27.49	11.00
Gemma-4-26B-A4B	101.25	83.52	95.36	78.00	16.00	13.20	21.20	21.30
Qwen3.8-27B	91.70	76.30	90.37	61.92	18.40	14.10	18.90	18.70
DeepSeek-V4-Flash	94.61	102.84	63.87	70.11	21.75	19.65	11.43	19.25
GLM-5.3-Flash	98.81	78.80	126.75	90.43	16.33	14.57	10.11	7.86
Gemini-3.7-Flash	15.03	29.60	33.90	24.80	7.10	8.62	7.30	7.20
GPT-5.6-Terra	94.60	78.80	90.40	66.00	19.00	14.60	19.50	19.30
Qwen3.8-Flash	101.55	78.79	101.97	74.50	22.75	15.38	27.90	22.94

Table 16: Mean API calls and processing time per editing instance for coding agents with PPT-specific skills.

Model	**Codex**		**Claude Code**		**DeepCode**		**OpenCode**	
	Calls	Time (s)	Calls	Time (s)	Calls	Time (s)	Calls	Time (s)
Qwen3.8-27B	19.60	61.90	12.00	43.80	11.90	61.80	13.30	40.90
DeepSeek-V4-Flash	15.67	64.18	10.84	61.01	7.37	49.33	11.01	46.33
GLM-5.3-Flash	10.90	66.33	7.70	51.37	10.00	98.70	8.50	59.80
GPT-5.6-Terra	14.40	63.80	8.80	47.80	8.70	68.80	9.80	43.60
Qwen3.8-Flash	14.40	68.50	8.80	45.25	16.60	73.30	7.98	47.07

More model interaction does not necessarily translate into more reliable editing. Across the eight backbones in the efficiency comparison, the best base coding agent uses more model requests and has lower SR than the best one-shot editor under the same backbone. Thus, additional interaction alone does not close the performance gap on exact PPTX editing. Adding PPT-specific skills improves SR while reducing mean processing time for every evaluated backbone–harness pair, showing that better editing outcomes do not necessarily require longer runs. Together, these results highlight the importance of task-specific guidance and editing-system design, rather than interaction frequency alone, in achieving reliable edits efficiently.

E.2 Fine-Grained Performance by Model

Figures 11–13 show editing success rates across five request forms and three edit categories for the three model backbones.

The three-backbone breakdown shows a consistent difficulty gap between explicit single-target editing and reference-based set editing. Across all four base coding agents, Set Reference trails Single Explicit by 15.94–34.11 percentage points. These requests combine target-set selection with reference-based modification, requiring the system to recover both the complete target set and the object supplying the reference property. One-shot editors narrow this gap: their best Set Reference SR exceeds that of the best base coding agent by 15.02–26.42 points across the three backbones. Nevertheless, the highest Set Reference SR among all evaluated systems remains 63.54–77.58%. The persistent shortfall shows that requests combining object selection and reference constraints remain challenging across editing pipelines.

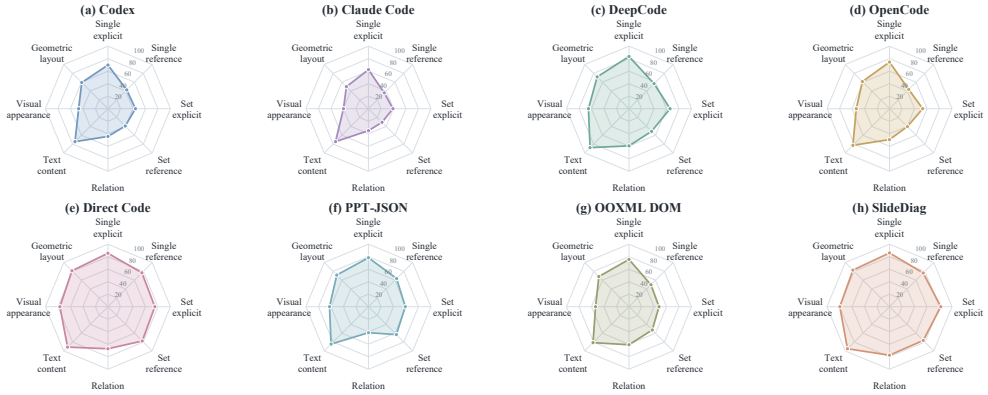

Figure 11: Fine-grained editing success rates (%) with Gemma-4-31B.

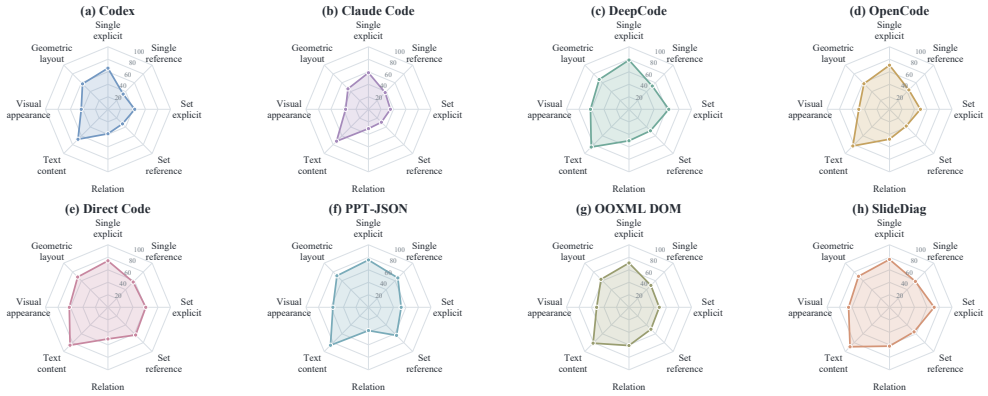

Figure 12: Fine-grained editing success rates (%) with Gemma-4-26B-A4B.

E.3 Performance by Object Type

Figures 14 and 15 report success rates across seven object types for base configurations and PPT-skill coding agents, respectively.

The object-type breakdown reveals recurring difficulties and differences in the benefits of domain-specific guidance. AutoShapes without text have lower SR than those with text in most evaluated configurations, including skill-augmented agents, suggesting that text-free shapes remain a common challenge across editing pipelines. Table editing benefits consistently from PPT-specific skills: across Qwen3.8-Flash, DeepSeek-V4-Flash, and GLM-5.3-Flash, the four coding agents gain 20.33–38.54 percentage points. These results suggest that domain guidance can alleviate some object-specific weaknesses, while difficulties with text-free shapes remain widespread.

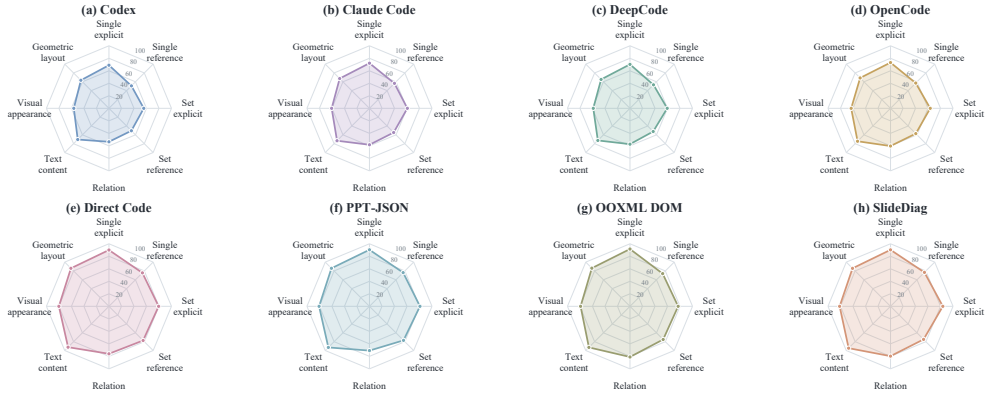

Figure 13: Fine-grained editing success rates (%) with Qwen3.8-27B.



Figure 14: Success rates (%) by object type for base configurations across four models.



Figure 15: Success rates (%) by object type for coding agents with PPT skill.