OpenDocAssistant: Language-Driven Document Automation and Evaluation

Anonymous ACL submission

Abstract

Modern document processing tools remain inaccessible to non-technical users due to steep learning curves. This paper introduces OpenDocAssistant, a natural language-driven document automation system that addresses three core challenges: multi-step instruction decomposition, semantic-to-API mapping, and efficient execution under resource constraints. Our three-stage architecture—planning, API selection, and execution—uses large language models (LLMs) to translate free-form instructions into document operations. The novel RaAPI mechanism combines dense embedding retrieval with LLM reasoning to bridge natural language instructions to appropriate API calls. Ablation studies show RaAPI's critical role (performance drops from 74.53% to 12.40% without it) and robust handling of vague instructions (>0.86 consistency, >0.95 API similarity). We evaluate six LLMs on OpenDocEval (110 annotated sessions) using Achievement Rate (AR) and Average Number of APIs (ANA). Large models achieve 74.53% AR on complex tasks, while smaller models offer practical accuracy–efficiency trade-offs. This work demonstrates LLMs' potential to democratize document automation through natural language interfaces.

1 Introduction

Professional document processing software has grown increasingly complex, often burying essential features within layered menus and imposing a high cognitive load on non-expert users. While traditional interface optimizations can mitigate some usability issues, they do not fundamentally resolve the semantic gap between user intent and system actions (Nielsen, 2023). The emergence of large language models (LLMs) offers a transformative opportunity: natural language interfaces can bridge this gap by translating user instructions into precise, actionable commands. Empirical evidence highlights this potential, with Microsoft reporting a 42% productivity boost in LLM-augmented office tasks (Cambon et al., 2023) and MIT observing a 59% efficiency gain among business professionals (Noy and Zhang, 2023).

Despite these advances, leveraging LLMs for document automation remains challenging. First, **instruction understanding** is non-trivial: complex requests such as "merge cells and apply conditional formatting" require systems to decompose and sequence multiple atomic actions, a process where existing methods often falter, especially with nested or composite operations (Zou et al., 2024). Second, **semantic alignment** is difficult: mapping ambiguous or underspecified instructions (e.g., "make the table professional") to concrete API calls demands sophisticated inference and contextual reasoning, given the inherent mismatch between natural language and formal APIs. Third, **efficient execution** is essential: maintaining document state and ensuring reliable operation under resource constraints (such as sub-second latency on edge devices) calls for frameworks that are both lightweight and robust, which current solutions rarely achieve (McIntosh et al., 2024).

Moreover, the field lacks rigorous evaluation standards. Most existing benchmarks focus on single-turn tasks or use oversimplified metrics, failing to capture the complexity and continuity of real-world document workflows (Fodor, 2025). This results in inflated performance estimates that do not reflect practical deployment scenarios.

To address these gaps, we introduce **OpenDocAssistant**, a unified framework that advances both methodology and evaluation for language-driven document automation. Our system features a three-stage pipeline: (1) *planning*, which decomposes user instructions into executable steps; (2) *API selection*, where we propose the novel **RaAPI** (Retrieval-augmented API Selection) mechanism. RaAPI uniquely integrates dense vector retrieval of

relevant APIs with LLM-based contextual reasoning, enabling accurate and flexible mapping from natural language to formal API calls—a significant improvement over prior approaches that rely solely on retrieval or direct generation. (3) *execution*, which applies state-aware operations to the document, ensuring consistency and correctness throughout the workflow.

Complementing the system, we present **OpenDocEval**, a comprehensive benchmark designed to reflect real-world document automation challenges. OpenDocEval introduces dual metrics: *Achievement Rate (AR)*, measuring task success, and *Average Number of APIs (ANA)*, quantifying execution efficiency. This dual perspective enables fair comparison across both high-capacity LLMs and lightweight models, and more accurately reflects practical utility.

In summary, OpenDocAssistant and RaAPI together bridge the semantic and operational gaps in document automation, while OpenDocEval provides the first rigorous, workflow-oriented evaluation standard for this domain. Our experiments demonstrate that this approach not only improves accuracy and efficiency, but also broadens accessibility to advanced document automation for a wider range of users and deployment environments.

2 Related Work

2.1 LLMs for Document Processing

Recent advancements in LLMs have significantly reshaped the landscape of document processing, a longstanding focus area in NLP.With the rapid advancement of LLMs, document-related applications have gained increasing attention. GPT-3 demonstrated strong few-shot capabilities in document generation and editing (Brown et al., 2020), while evaluations of ChatGPT revealed limitations in structured document manipulation, particularly in layout and formatting comprehension (Winn, 2023). Despite promising capabilities, general-purpose LLMs often lack deep understanding of document-specific structures and workflows. Structure-aware pretraining approaches such as LayoutLM (Xu et al., 2020) and its multimodal extension LayoutLMv2 (Xu et al., 2021) have been proposed to address this gap.

A growing direction involves integrating LLMs with external APIs to enhance document operation capabilities. ToolFormer introduced a self-supervised method for LLMs to acquire tool use abilities (Schick and Schütze, 2023), while ToolLLM developed a framework for mastering thousands of real-world APIs (Qin et al., 2023). Unlike these methods, OpenDocAssistant specifically addresses semantic-to-API mapping and execution challenges within complex document environments.

2.2 Document-Centric Evaluation Benchmarks for LLMs

While advancements in LLM-based document processing systems have been significant, a comprehensive evaluation framework is equally critical to measure their practical effectiveness. Therefore, we next review document-centric evaluation benchmarks. Effective evaluation of LLMs in document processing requires benchmarks that capture domain-specific operational characteristics. While multi-dimensional frameworks have assessed general aspects such as accuracy, robustness, and efficiency (McIntosh et al., 2024), broader evaluations highlight that no single dataset comprehensively measures model capabilities (Banerjee et al., 2024).

Among recent efforts, benchmarks such as DocBench (Zou et al., 2024) and PPTC (Guo et al., 2023) have been proposed to address these evaluation gaps. DocBench focuses on metadata extraction, layout parsing, and multimodal understanding across static documents, while PPTC pioneers multi-turn PowerPoint creation and editing. However, these benchmarks remain constrained: PPTC supports short interactions (2.3 turns per session) and employs basic state tracking, whereas OpenDocAssistant handles longer sessions (averaging 3.57 turns) with robust state management via `save_state()` and `load_state()` functions, enabling complex multi-turn manipulations. Moreover, PPTC emphasizes presentation-specific operations, while our framework supports a broader range of document types through modular API selection.

Additionally, critiques point out that many benchmarks rely on single-turn evaluations, failing to emulate real-world multi-step workflows (Fodor, 2025), and enforce single gold-standard outputs, overlooking the multiplicity of valid document solutions (McIntosh et al., 2024). These limitations motivate the need for dynamic evaluation frameworks like OpenDocEval that better reflect practical document processing challenges.

3 OpenDocAssistant

3.1 System Overview

Modern document processing software's complexity creates accessibility barriers for non-technical users, reducing productivity and hindering automation. OpenDocAssistant addresses this through a three-stage framework: planning, API selection, and execution. As shown in Figure 1, planning decomposes instructions, API selection maps them to operations via RaAPI (hybrid retrieval-and-reasoning), and execution performs validated actions. The system uses Qwen2-7B-Instruct for semantic interpretation and `python-docx` for operations, enabling real-time error correction through a closed-loop user interaction cycle (Ibrahimzada, 2024).

3.2 Planning Stage

The planning stage bridges user intent and executable operations by breaking down complex instructions into ordered sequences of basic actions. It uses structured prompt engineering and task-specific templates to guide the LLM through a three-step process: prompt construction, model inference, and post-processing (Algorithm 1). For example, "Create a document, add a title, then insert a table" is decomposed into distinct actions, leveraging the LLM's ability to infer logical dependencies (Ibrahimzada, 2024). The algorithm operates in $O(n+m)$, with LLM inference costing $O(n^2)$.

3.3 Retrieval-Augmented API Selection (RaAPI)

We propose **RaAPI**, a hybrid mechanism that bridges natural language instructions and formal API calls in OpenDocAssistant. RaAPI combines dense vector retrieval with LLM-based contextual reasoning to select and sequence APIs for user commands.

The process begins with a retrieval phase, where user instructions and API specifications are encoded into dense vectors using the `text-embedding-ada-002` model. This transforms both instructions and API documentation into a high-dimensional semantic space, where cosine similarity captures conceptual relationships beyond keyword overlap:

$$\text{sim}(I, A_i) = \frac{\vec{E}_I \cdot \vec{E}_{A_i}}{||\vec{E}_I|| \cdot ||\vec{E}_{A_i}||} \quad (1)$$

Algorithm 1: Hierarchical Instruction Decomposition

Input: Instruction `I`
Output: Ordered sequence of sub-instructions `S`

```
if NotPlanningEnabled() then
    return [I]; // No decomposition required
template ← FormatPrompt(I);
response ← QueryLanguageModel(template, model);
subInstructions ← ParseResponse(response);
if IsEmpty(subInstructions) then
    return [I]; // Fallback if model fails
S ← FilterAndNormalize(subInstructions);
S ← EnrichWithContext(S, I);
return S;
```

where $\vec{E}_I$ and $\vec{E}_{A_i}$ are the embeddings for instruction I and API candidate A_i. To reduce latency, we use a persistent embedding cache: pre-computed embeddings are loaded if available, otherwise computed and stored for future use.

Top-k API candidates are passed to an LLM reasoning module. Prompted as a document automation assistant, the LLM receives APIs (signatures, descriptions), strict output rules, checks, and examples. A carefully designed prompt guides the LLM to interpret complex requests, select relevant APIs, extract/format parameters per signatures, and produce a logically ordered, syntactically correct API call sequence. This prompt engineering constrains LLM output to valid calls and encourages reasoning about document structure, dependencies, and parameters, as seen in codebase templates (`src/api_selection.py`).

The LLM's multi-step, context-aware reasoning analyzes instructions and retrieved APIs, breaking down complex requests into atomic operations. It maps these to relevant APIs using prompt context and its knowledge. Parameters are extracted/formatted per API signatures, and the final API sequence is generated in a single, strictly formatted, logically ordered line. This handles ambiguous, multi-step, or underspecified instructions beyond retrieval's scope. While the LLM doesn't manage document state, it generates API sequences

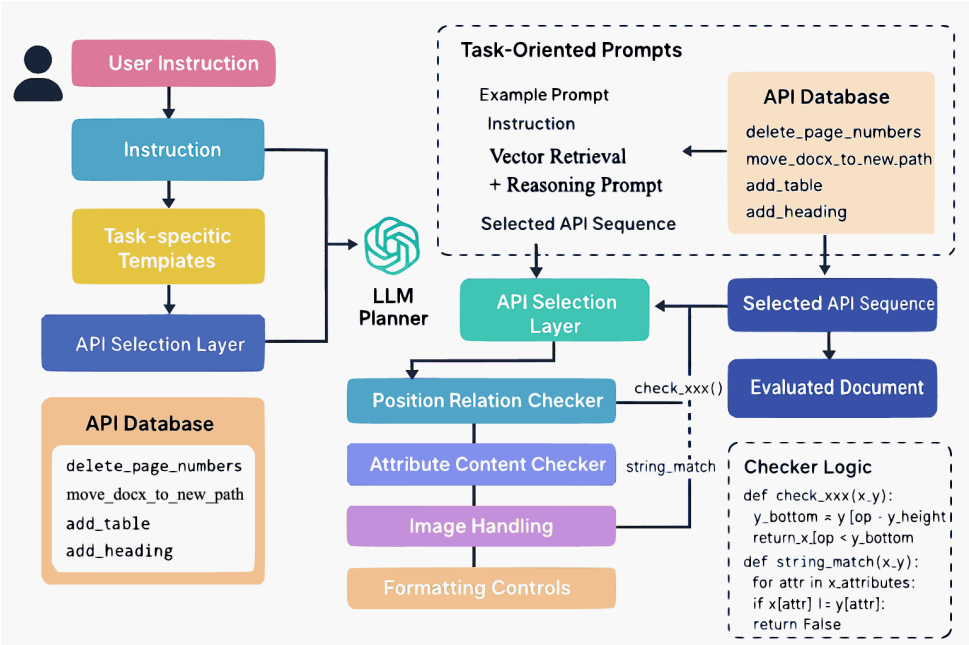

Figure 1: System framework of OpenDocAssistant showing the three-stage design. (1) The Planning stage processes instructions via task-specific templates. (2) The API selection stage combines vector retrieval and LLM reasoning. (3) The Execution stage enforces validation rules.

executed by a stateful backend (`word_executor`), ensuring correct document structure and order. This interplay is essential for robust automation, as LLM output is interpreted within the evolving document's context.

Algorithm 2 outlines the vector retrieval process. The LLM then takes the sorted top-k APIs and the original instruction to generate the final API sequence. This hybrid design ensures both broad coverage and precise, context-aware API selection, which is essential for robust document automation.

3.4 Execution Stage

The execution stage converts API call sequences into document edits via a hierarchical interface covering text, tables, images, and formatting. A state manager tracks the current document and key elements, ensuring context-aware, sequential operations. Multi-level error handling catches exceptions per API call, normalizes error messages, and applies fallbacks for noncritical failures. Core principles—functional atomization, sensible defaults, composability, and explicit state feedback—underpin reliable execution and support higher-level modules (Fowler, 2002).

Algorithm 2: Vector-Based API Selection for Instruction Processing

Input: Instruction `I`, Candidate API set `A`
Output: Top-k most relevant APIs

```
embedding_I ← GetEmbedding(I, model="text-embedding-ada-002");
// Encode instruction
similarityList ← []; // Initialize similarity scores
foreach api in A do
    embedding_api ← GetAPIEmbedding(api);
    score ← CosineSim(embedding_I, embedding_api);
    similarityList.append((api, score));
sortedAPIs ← SortByScore(similarityList, descending=True);
return TopK(sortedAPIs, k=10);
```

4 OpenDocEval

The evaluation of LLMs in document processing faces challenges in handling ambiguous instructions, complex API sequences, and diverse execution paths. Traditional single-point metrics fail to capture model performance in knowledge-intensive scenarios. We propose OpenDocEval—a structured framework for assessing LLM capabilities in document processing.

4.1 Evaluation Framework

Figure 2 shows the framework's two-dimensional structure: independent session units horizontally and sequential interaction turns vertically. Each turn follows a pipeline: instruction input, operation generation, execution, result comparison, and metric computation. Formally, given instructions $I = \{i_1, i_2, \ldots, i_n\}$ and operations $A = \{a_1, a_2, \ldots, a_m\}$, the system learns a mapping function $f : I \times D_{j-1} \rightarrow S_j$ that maps instruction i_j and prior state D_{j-1} to operation sequence S_j, producing document state D_j that satisfies user intent.

4.2 Multidimensional Evaluation Metric Framework

We introduce a hierarchical metric system evaluating performance across functional accuracy and computational efficiency.

Functional Metrics Achievement Rate (AR) is the primary metric, assessing functional equivalence between predicted and reference documents. We report AR at two granularities:

- **Task-level AR** (AR_{task}): Measures success for individual tasks:

$$\text{AR}_{\text{task}} = \frac{\sum_{i=1}^{N_T} C(D_{p,i}, D_{l,i})}{N_T} \tag{2}$$

- **Session-level AR** ($\text{AR}_{\text{session}}$): Assesses success over entire sessions:

$$\text{AR}_{\text{session}} = \frac{\sum_{j=1}^{N_S} C(D_{p,j,\text{final}}, D_{l,j,\text{final}})}{N_S} \tag{3}$$

Efficiency Metrics We measure the Average Number of APIs (ANA) to quantify operational efficiency, with lower values indicating higher efficiency. We report ANA at both task and session granularities:

- **Task-level ANA** (ANA_{task}): Average API calls per task:

$$\text{ANA}_{\text{task}} = \frac{1}{N_T} \sum_{i=1}^{N_T} |A_i| \tag{4}$$

- **Session-level ANA** ($\text{ANA}_{\text{session}}$): Average total API calls per session:

$$\text{ANA}_{\text{session}} = \frac{1}{N_S} \sum_{j=1}^{N_S} \left(\sum_{k=1}^{T_j} |A_{j,k}| \right) \tag{5}$$

4.3 Test Set Design and Evaluation Methodology

To rigorously evaluate document processing capabilities, we constructed a comprehensive test set comprising 110 sessions (84 for document creation, 26 for template editing) with 750 instructions. The test set covers text, table, image, and hybrid operations at varying complexity levels, with each session maintaining contextual dependencies between instructions to evaluate model performance stability. As shown in Table 1, our test set includes diverse operation types and complexity levels, with text operations being the most common (45.2% in document creation, 61.5% in template editing) and advanced tasks (4+ operations) constituting 40.5% of document creation tasks.

Our evaluation process follows a three-stage pipeline: (1) preparation, where we load test sessions and generate reference documents; (2) execution, conducting both isolated and continuous interactions; and (3) error analysis, categorizing failures into selection, parameter, and execution errors. This structured approach ensures reproducibility and supports automated evaluation of model performance.

5 Experiment

This section presents a focused evaluation of OpenDocAssistant, highlighting how model architecture, the RaAPI mechanism, and task characteristics jointly affect system performance. We detail the experimental setup and present the primary performance metrics.

5.1 Model Capability Evaluation

We benchmarked a range of models, both commercial and open-source, with varying parameter scales. The task-level and session-level results, including Achievement Rate (AR) and Average Number of APIs (ANA), are summarized in Table 2.

5.2 Ablation and Robustness Study

We conducted comprehensive ablation studies and targeted robustness tests using the `deepseek-v3` model. These experiments assess the contribution of key components, especially the Retrieval-augmented API Selection (RaAPI) mechanism, and evaluate resilience to imperfect inputs. Performance was measured across three primary scenarios: the standard system evaluated on the robust test set, the ablated system (without RaAPI) on a

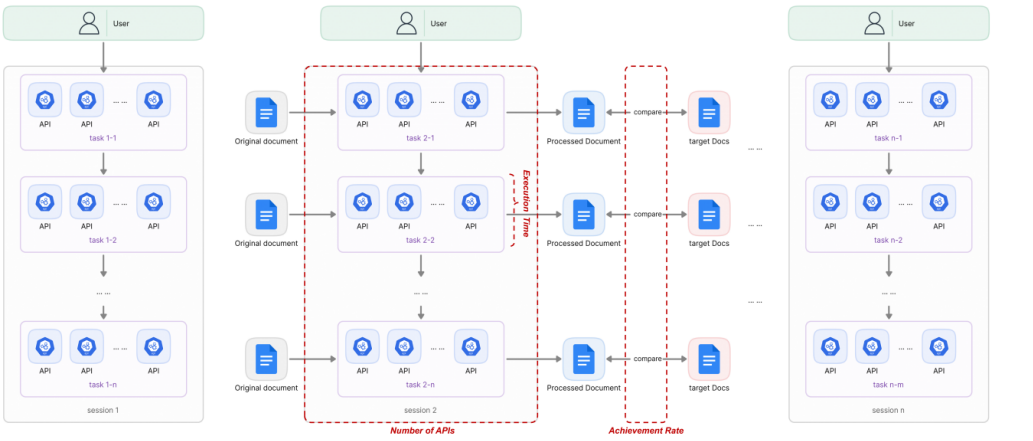

Figure 2: OpenDocEval evaluation framework architecture. The framework is organized horizontally into three independent session units (Sessions 1, 2, and 3) and vertically into sequential interaction turns (Tasks 1-1, 1-2, etc.). Each column represents a different session evaluation approach. The middle section (Session 2) illustrates the complete pipeline, with original documents being processed through API sequences to produce processed documents. These are then compared against target documents to calculate both ANA and AR metrics. Blue circular icons represent API operations executed for each task.

Characteristics	**Create_new_docs**	**Edit_Word_template**
Sessions	84	26
Avg. instructions/session	3.57	2.15
Document types	New documents	Existing templates
Operation type distribution (%)		
Text operations	45.2	61.5
Table operations	32.1	23.1
Image operations	15.5	7.7
Hybrid operations	7.2	7.7
Complexity level distribution (%)		
Basic (single operations)	21.4	30.8
Intermediate (2–3 operations)	38.1	53.8
Advanced (4+ operations)	40.5	15.4

Table 1: Dataset characteristics for document creation and editing.

standard test set, and the ablated system also on the robust test set. Performance metrics (Task AR, Session AR, Task ANA, Session ANA) are presented in Table 3.

To further analyze robustness against input variations, we evaluated the system's performance on specific types of fuzzy instruction variants: keyword-only, vague terms, and truncated inputs. Table 4 presents the Document Consistency Rate and High API Similarity Rate for both the standard system (with RaAPI) and the ablated system (without RaAPI) when subjected to these fuzzy tests.

6 Analysis and Discussion

This section analyzes the experimental results presented in Section 5, discussing model capabilities, the impact of the RaAPI mechanism, system robustness, and error patterns.

6.1 Model Capability Analysis

6.1.1 Performance Overview

The results from Table 2 highlight significant performance variations across different models. `deepseek-v3` achieves the highest task-level achievement rate (AR) at 74.53% and the best session-level AR at 36.36%, while also maintaining a low average number of APIs per task (ANA) of 4.37, indicating efficient planning. In

Table 2: OpenDocEval results across different model variants. AR: Achievement Rate (%), ANA: Average Number of APIs.

Models and Methods	Task AR (%)	Task ANA	Session AR (%)	Session ANA
gpt-4.1	**70.00**	**4.21**	**31.82**	**34.66**
qwen3-plus	60.93	4.64	30.91	35.52
claude-3-7-sonnet	63.73	6.60	0.00	50.48
gemini-2.0-flash	59.87	5.45	12.73	41.73
deepseek-v3	**74.53**	**4.37**	**36.36**	**33.44**
lama3-70b	63.60	7.59	13.64	58.11
qwen3-14b	45.47	4.39	0.00	33.59
deepseek-r1-llama-8b	37.33	11.25	0.00	86.07
qwen2.5-instruct-7b	33.60	13.22	0.00	101.09
deepseek-qwen2.5-math-7b	11.87	27.84	0.00	213.09

Table 3: Overall Performance Metrics for **deepseek-v3** under Ablation and Robustness Scenarios.

Scenario	Task AR (%)	Task ANA	Session AR (%)	Session ANA
Standard System on Robust Test	60.93	4.54	21.00	17.46
Ablated System on Standard Test	12.40	1.60	0.00	12.37
Ablated System on Robust Test	10.40	1.60	0.00	14.27

contrast, deepseek-qwen2.5-math-7b performs worst, with only 11.87% task AR and 0% session AR, and requires an average of 27.84 APIs per task, reflecting inefficient decomposition and execution. gpt-4.1 also demonstrates strong performance, with a 70.00% task AR and the lowest ANA (4.21), but its session AR drops to 31.82%. qwen3-plus and claude-3-7-sonnet achieve moderate task ARs (60.93% and 63.73%, respectively), but claude-3-7-sonnet fails to generalize to session-level tasks (0.00% session AR), and qwen3-plus achieves 30.91% session AR. gemini-2.0-flash shows a task AR of 59.87% and session AR of 12.73%. A notable trend across all models is the significant decline from task-level to session-level AR, underscoring the challenge of maintaining state and context across multiple turns.

6.1.2 Impact of Task Complexity and Model Specialization

Task complexity critically impacts performance. Simpler tasks show similar model performance, but complexity widens gaps. For instance, deepseek-v3 and gpt-4.1 maintain high ARs on complex tasks, unlike qwen2.5-instruct-7b and deepseek-qwen2.5-math-7b whose ARs drop below 35%. Text operations generally outperform table or image tasks, likely due to models' language-centric training. Editing, needing state recognition and contextual changes, remains challenging. In multi-turn editing, models often struggle with consistency (e.g., updating charts post-table changes), showing limits in current planning and state management.

While larger models often perform better, domain adaptation and planning are vital. Top models generate concise API sequences for complex tasks, shown by low ANA values (e.g., deepseek-v3: 4.37, gpt-4.1: 4.21 from Table 2), whereas smaller models produce longer, inefficient chains (e.g., deepseek-qwen2.5-math-7b: 27.84). For example, deepseek-v3 merges tables in one pass, while weaker models need multiple steps or fail entirely. This underscores the need for both model capability and robust API selection for reliable document automation.

6.1.3 Error Analysis

To pinpoint system weaknesses, we analyze errors across the three-stage pipeline. Figure 3 shows the distribution: planning (42%), API selection (31%), and execution (27%).

Planning errors often stem from difficulties in decomposing nested instructions. API selection errors arise when parameter mappings are ambiguous or necessary context is missing. Execution

Table 4: Granular robustness test results on different types of fuzzy instructions for Full RaAPI and Ablated RaAPI configurations.

Configuration	Instruction Type	Document Consistency (%)	API Similarity (%)
Full RaAPI (under Fuzzy Test)	keywords_only	2.4	0.8
	vague_terms	86.4	95.1
	truncated	10.4	2.8
Ablated RaAPI (under Fuzzy Test)	keywords_only	3.2	0.8
	vague_terms	87.6	96.4
	truncated	10.1	1.9

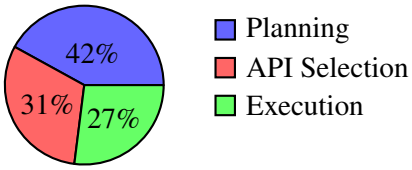

Figure 3: Error distribution across pipeline stages.

errors typically occur due to invalid parameters or incorrect operation ordering. These findings suggest three primary directions for improvement: (1) refining decomposition templates to better capture dependencies between sub-tasks, (2) enforcing stricter type–context checks during API matching to reduce ambiguity, and (3) strengthening state validation at execution time to prevent cascading errors.

6.2 Ablation and Robustness Analysis

6.2.1 Impact of RaAPI and System Robustness

The ablation studies (Table 3) demonstrate RaAPI's critical role in system performance. The standard `deepseek-v3` system achieves 60.93% Task AR on the robust test set, showing resilience despite a notable drop from its peak performance (74.53% Task AR, 36.74% Session AR on standard tests). Ablating the RaAPI mechanism leads to severe degradation, with Task AR dropping to 12.40% on standard tests and 10.40% on robust tests. This contrast highlights RaAPI's essential role in enabling accurate document manipulation. The low ANA values (1.60) in ablated configurations indicate an inability to generate complete API sequences.

6.2.2 Performance on Fuzzy Instructions

Robustness tests on fuzzy instructions (Table 4) reveal that both full and ablated RaAPI configurations handle vague terms well (Document Consistency >86%) but struggle with keyword-only and truncated instructions (Consistency <11%). The similar performance patterns suggest that while RaAPI is crucial for overall task success, neither it nor the LLM alone can reliably handle highly fragmented inputs. This indicates that improving robustness against underspecified instructions remains a key challenge, requiring enhanced understanding mechanisms beyond current implementations.

7 Conclusion

This paper presents OpenDocAssistant, a language-driven document automation system that transforms complex processing into natural language interactions through a three-stage framework. Evaluations across 110 test sessions demonstrate its effectiveness, achieving 74.53% Achievement Rate with only 4.37 API calls per task. The plan-select-execute paradigm represents a fundamental shift in human-software interaction, with RaAPI proving essential (performance drops from 74.53% to 12.40% without it) and showing robust handling of vague instructions (>0.86 consistency, >0.95 API similarity). Future work will focus on multimodal interactions, personalization, and broader software applications.

Limitations

The current system has several limitations: reliance on a single language model for planning and execution, performance constraints from underlying APIs, challenges with ambiguous instructions (as shown in robustness tests), dependency on high-quality training data, and computational resource requirements for inference.

References

Soumya Banerjee, Ananya Agarwal, and Eshan Singh. 2024. The vulnerability of language model benchmarks: Do they accurately reflect true llm performance? *arXiv preprint*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, and 1 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33.

A. Cambon, B. Hecht, B. Edelman, D. Ngwe, S. Jaffe, A. Heger, and J. Teevan. 2023. Early llm-based tools for enterprise information workers likely provide meaningful boosts to productivity. Technical Report MSR-TR-2023-01, Microsoft Research.

J. Fodor. 2025. Line goes up? inherent limitations of benchmarks for evaluating large language models. *arXiv preprint*.

Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, MA.

Yuwei Guo, Zhenzhen Zhang, Yihan Liang, Dongdong Zhao, and Nan Duan. 2023. Pptc benchmark: Evaluating large language models for powerpoint task completion. *arXiv preprint*.

A. R. Ibrahimzada. 2024. Program decomposition and translation with static analysis. *Preprint*, arXiv:2401.12412.

T. R. McIntosh, T. Susnjak, N. Arachchilage, T. Liu, P. Watters, and M. N. Halgamuge. 2024. Inadequacies of large language model benchmarks in the era of generative artificial intelligence. *arXiv preprint*.

Jakob Nielsen. 2023. Chatgpt lifts business professionals' productivity and improves work quality. Nielsen Norman Group.

Shakked Noy and Whitney Zhang. 2023. Experimental evidence on the productivity effects of generative artificial intelligence. MIT Economics Working Paper.

Yujia Qin, Shuming Liang, Yujia Ye, Kun Zhu, Linjie Yan, Yujie Lu, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint*.

Timo Schick and Hinrich Schütze. 2023. Toolformer: Language models can teach themselves to use tools. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages 293–304.

Zach Winn. 2023. Study finds chatgpt boosts worker productivity for some writing tasks. MIT News.

Yiheng Xu, Zihan Dai, Zhengyuan Li, Yang Gao, Jianfeng Li, Bing Qin, and Tie-Yan Liu. 2021. Layoutlmv2: Multi-modal pre-training for document image understanding. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, pages 2579–2591.

Yiheng Xu, Minghao Yang, Lei Liu, Yujie Wang, Furu Cao, and Yizhou Li. 2020. Layoutlm: Pre-training of text and layout for document image understanding. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1192–1200.

A. Zou, W. Yu, H. Zhang, and 1 others. 2024. Docbench: A benchmark for evaluating llm-based document reading systems. *arXiv preprint*.

A Dataset Description

Our dataset consists of 100 document creation sessions, each containing multiple tasks for creating and editing Word documents. The dataset is organized into two versions: a standard version and an API-lack version, allowing for comprehensive evaluation of document automation capabilities.

Each session is stored in a JSON file (e.g., `session_X.json`) containing a sequence of tasks. Each task includes a sequential identifier, natural language command for document manipulation, corresponding sequence of API calls to execute the instruction, paths to initial and expected final document states, and alternative versions for API-lack scenarios.

The dataset encompasses a comprehensive range of document operations, including document structure management (headers, footers, page numbers, table of contents), content formatting (font styles, colors, paragraph formatting), table operations (creation, cell content, headers), list management (ordered and unordered lists), hyperlink and reference handling, and various document elements (watermarks, line breaks, spacing).

The dataset comprises 100 sessions with an average of 7-8 tasks per session, resulting in over 700 unique instructions. It covers more than 20 different types of API operations and includes various document types such as meeting minutes, reports, and forms.

The dataset was created through a systematic process involving task design for realistic document creation scenarios, instruction writing in Chinese, API sequence generation for executable calls, document state tracking for intermediate and final states, and quality control for instruction-API mapping accuracy.

The dataset supports comprehensive evaluation scenarios, including task-level evaluation for individual instruction execution accuracy, session-level evaluation for complete document creation workflows, API-lack scenarios for testing robustness

with limited API availability, and cross-version comparison between standard and API-lack performance.

B RaAPI and Planning Mechanisms

The RaAPI (Robust API Planning and Integration) mechanism employs a hierarchical architecture for document automation, consisting of three key components. The API Selection Module utilizes a fine-tuned LLM to map natural language instructions to executable API sequences through a two-stage process of instruction analysis and API mapping with parameter validation. The State Management component maintains document state through a structured representation, tracking hierarchical elements, monitoring formatting and content modifications, and recording operation history for rollback and validation. The Error Recovery system implements robust error handling through API availability checking, fallback mechanisms for failed operations, and state recovery for maintaining document consistency.

The planning phase orchestrates document creation through a multi-step process. Task Decomposition breaks down complex document creation tasks into atomic operations, identifies dependencies, and generates optimal execution order. The LLM Prompting Strategy incorporates document state and history through context-aware prompting, utilizes few-shot learning with example API sequences, and employs chain-of-thought reasoning for complex operations. Execution Planning generates validated API call sequences, handles conditional operations based on document state, and optimizes operation order for efficiency.

The system maintains document structure through a comprehensive approach to hierarchical representation, state tracking, and format consistency. The hierarchical representation employs a tree-based structure for document elements, manages parent-child relationships for nested elements, and implements position tracking for content insertion. State tracking provides real-time updates of document modifications, validates structural integrity, and resolves conflicts in concurrent operations. Format consistency is maintained through style inheritance and propagation, format validation and correction, and cross-element reference management.