
DrafterBench: Benchmarking Large Language Models for Tasks Automation in Civil Engineering

Yinsheng Li

Department of Civil Engineering
McGill University
yinsheng.li@mail.mcgill.ca

Zhen Dong

UC Santa Barbara and NVIDIA
zhendong@berkeley.edu

Yi Shao

Department of Civil Engineering
McGill University
yi.shao2@mcgill.ca

Abstract

Large Language Model (LLM) agents have shown great potential for solving real-world problems and promise to be a solution for tasks automation in industry. However, more benchmarks are needed to systematically evaluate automation agents from an industrial perspective, for example, in Civil Engineering. Therefore, we propose DrafterBench for the comprehensive evaluation of LLM agents in the context of technical drawing revision, a representation task in civil engineering. DrafterBench contains twelve types of tasks summarized from real-world drawing files, with 46 customized functions/tools and 1920 tasks in total. DrafterBench is an open-source benchmark to rigorously test AI agents' proficiency in interpreting intricate and long-context instructions, leveraging prior knowledge, and adapting to dynamic instruction quality via implicit policy awareness. The toolkit comprehensively assesses distinct capabilities in structured data comprehension, function execution, instruction following, and critical reasoning. DrafterBench offers detailed analysis of task accuracy and error statistics, aiming to provide deeper insight into agent capabilities and identify improvement targets for integrating LLMs in engineering applications. Our benchmark is available at Github-DrafterBench, with the test set hosted at Huggingface.

1 Introduction

Recently, Large Language Models(LLMs) have been demonstrated with remarkable capabilities in planning (Wu et al., 2023a), problem-solving (Kim et al., 2024), tool calling (Kim et al., 2023), programming (Li et al., 2022), etc. There is a growing trend for the integration of general LLMs in real scenarios, where one of the most promising fields is automating tasks by LLM agents in the industry (Ahn et al., 2022; Song et al., 2023). Such automation solutions are urgently needed in **Civil Engineering**, as there are bunches of monotonous, low-tech, and high-labor-intensity tasks from the construction stage (Liu et al., 2019) to the design stage (EvolveLab, 2025). They benefit users by helping them focus on more complex and skill-intensive work and create more value in the same work time. Systematic evaluation and comprehensive analysis from the perspective of industrial applications are critical to gain a deeper understanding of the capabilities of models and to identify targeted improvements. However, few benchmarks put them in the shoes of real-world industrial tasks, especially those of Civil Engineering.

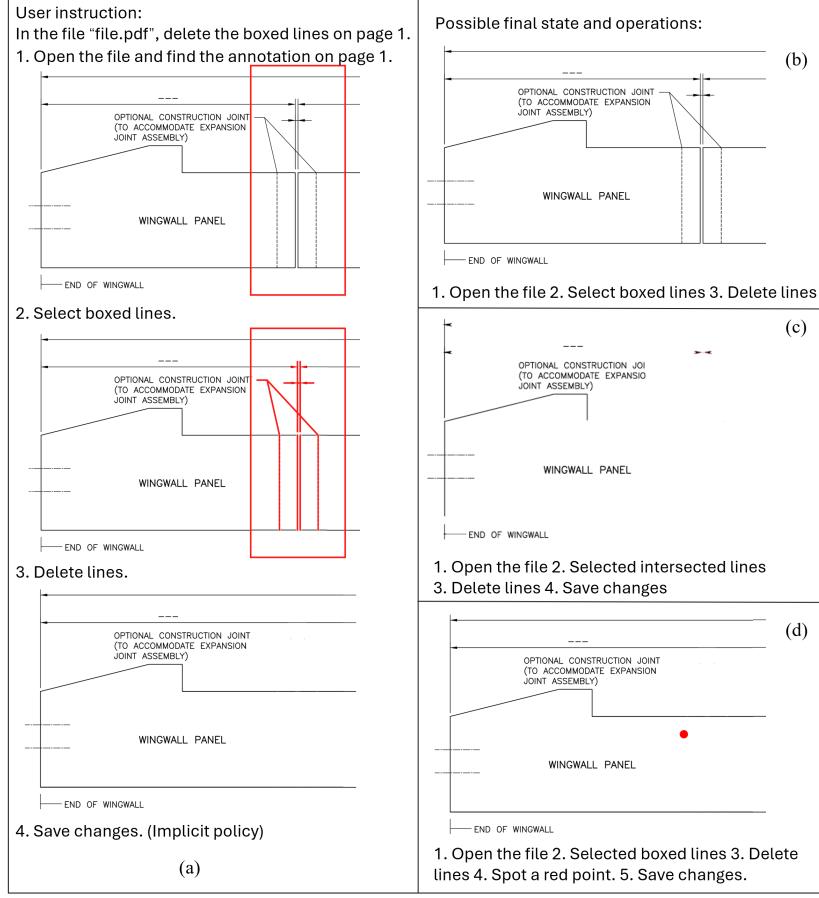


Figure 1: The workflow and possible results of a simple drawing revision task in civil engineering. (a) The whole process involves a series of operations, starting from opening the file, locating the annotation, selecting the expected lines, executing deletion, and ending with saving the changes. (b)(c)(d) illustrate the final outcomes of different WRONG operations.

The innateness of industrial tasks brings unique challenges for both AI agents and benchmarks compared to those developed for general tasks. To illustrate them, we take the drawing revision as an example, which is one of the most labor-intensive and low-tech tasks that needs automation in Civil Engineering (according to our interviews with more than ten construction companies in North America). First, industrial tasks require a skilled worker to provide a complete solution by integrating available tools, prior knowledge, and implicit policies rather than simply calling functions following instructions. Figure 1a illustrates the complete process of performing a simple revision task. It is worth noting that after deleting the lines as instructed, the changes should be saved as a new file named according to the company or the user's policies. Usually, the prior knowledge and policies are common sense for a skilled human practitioner and are not mentioned in the instructions. But it is a challenge for LLMs who lack them to know them and act on them like a skilled worker. Second, high robustness is crucial in industrial tasks, which means that an exact drawing is expected even if an instruction is expressed in varying language styles and expressions from different individuals. Third, it is essential to ensure the accuracy of every detail in the workflow. A task fails even if a simple operation is omitted (forget to save changes, Figure 1b), a parameter value is in error (delete all lines rather than only boxed lines, Figure 1c), or an unexpected parameter is specified (an extra point appears, Figure 1d). Fourth, it is difficult to assess the quality of the LLMs' performance directly from the revised drawings. This is because not all steps in a solution make any visible changes, resulting in 'what can be seen is not what has been done'. It is possible that an unclean solution with invisible extra operations outputs a drawing that is the same as the ground truth.

Existing benchmarks actively explore the intelligence limit of LLMs in various scenarios (Srivastava et al., 2022; Wu et al., 2023b; Zhuo et al., 2024; Xue et al., 2024), but they fall short of automating industrial tasks. They assess the capabilities of LLMs with the tasks of getting answers by following

instructions straightforwardly and paying less attention to implicit policies. Besides, the interactive style of multi-turn and multi-round is a trend to evaluate the serviceability of agent assistants in the real world (Ma et al., 2024; Wang et al., 2023). However, agents in these styles are sometimes found to be unstable and overly creative, which conflicts with the high-robustness requirement in industrial tasks (Kim et al., 2023; Xu et al., 2023; Mialon et al., 2023). Meanwhile, they usually require an observable environment or human feedback to promote the mandate process. However, for autonomous tasks in industry, building an observable environment may not be easy, and it is desirable to minimize human involvement to maximize their productivity. In addition, many benchmarks shed light on the completion of the final result, obscuring the comprehensive analysis of where a failure stems from (Yin et al., 2024a). The latter is hugely helpful for having a deeper and clearer insight into the capabilities of models. This is especially true given that in industry tasks, ground truth results can be obtained both through ground truth paths in various forms and contaminated paths in accidents.

In this work, we introduce DrafterBench, an open-source automatic toolkit, to evaluate AI agents in automating a representation task of civil engineering, drawing revisions. By analyzing real drawing revision documents, 12 types of tasks are summarized across four types of operations (adding, content modification, mapping, and format updating) and three objects (text, table, and vector). Forty-six tools for revising drawings in PDF files are customized and provided with the necessary prior knowledge and implicit policy in the system information. As a result, 1920 tasks in dynamic instruction quality were prepared and verified by humans to simulate the real scenario. DrafterBench assesses four essential capabilities of models: structured data understanding, function execution, instruction following, and critical reasoning. It offers a systematic analysis of task accuracy and error statistics on six subtasks. To accurately assess the models, dual functions are designed to record ground operation paths, which may be expressed in various coding styles by different models in different responses. The operation paths, instead of the output drawings, were compared with the ground truth paths to grade the performance of the models and analyze detailed errors. Our contributions are as follows:

- We introduce DrafterBench, which provides a comprehensive analysis of the strengths and limitations of LLM agents to automate monotonous and low-tech tasks for industry scenarios, especially civil engineering.
- A fully automated evaluation toolkit is released, providing a stable and accurate evaluation on models, resisting stochastic variation in models' response style and manner, avoiding the instability that may be encountered in interactive tasks.
- We conducted experiments for different mainstream LLMs. The results demonstrate that DrafterBench can give a clear view of their strengths and deficiencies. We hope that the proposed benchmark can provide insight for the future development of LLM agents, especially for integrating LLMs in engineering applications.

2 Related Works

LLM-based Agent Significant advances in LLM agents have been made with the development of the reasoning enhancement framework for LLM (Wei et al., 2022; Yao et al., 2024, 2022; Besta et al., 2023). By accomplishing relatively simple subtasks one by one, agents can solve complex tasks (Masterman et al., 2024; Srinivasan et al., 2023). ReAct is the most popular agent construction approach, which commands the agent to observe the environment to think deeply, complete a simple operation each turn, and gradually move toward the final answer (Yao et al., 2022). Based on the structure, agents can be categorized as single-agent or multi-agent. A single-agent accomplishes observing, planning, selecting, executing, and other actions by self-asking and answering (Paranjape et al., 2023), while a multi-agent system has independent modules responsible for each of the above four or more aspects and accomplishes the task through inter-module communication (Shi et al., 2024). Despite the ReAct-style agent's power, in industrial application scenarios, there is likely no environment for interaction, and agents that can accomplish tasks in a single turn are preferred.

Function-calling Benchmarks There are a number of benchmarks that have been built to evaluate the performance of agents, especially function-calling agents (Xu et al., 2023; Qin et al., 2023; Li et al., 2023), on basic capabilities such as understanding, reasoning, planning, problem-solving, and their behavior in real tasks (Song et al., 2023; Yin et al., 2024b). The benchmarks have included a

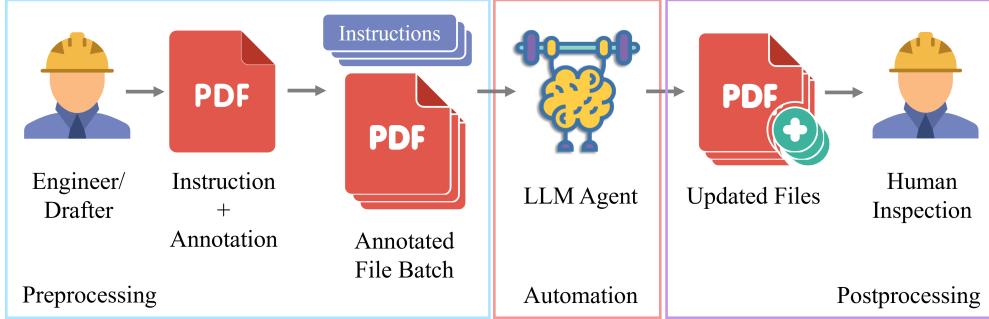


Figure 2: Expected workflow of an automation agent for drawing revision: The engineers or drafters leave their modification instructions and mark the related locations in the target files. Then, preprocessing will extract instructions and prepare files containing only the position marker for the agent. The LLM agent receives the instructions and files and automatically takes action to execute the modification. Finally, the workflow ends with a human inspection of the updated files.

variety of tasks such as math, games, science quizzes, code, etc (Ma et al., 2024; Zhuo et al., 2024). However, few benchmarks have been built on the real engineering tasks that need to be automated. Meanwhile, recent benchmarks have focused chiefly on multi-step, multi-turn problem solving (Yan et al., 2024; Ma et al., 2024) and not so much on one-turn, long-sequence high-level function callings, which are much more efficient in some tasks in the industry.

3 The DrafterBench

DrafterBench is designed to evaluate LLM models when serving as an automation agent to save engineers’ or drafters’ effort on low-tech monotonous work in industrial scenarios like drawing revision. Thus, tasks in DrafterBench are limited to the part of the work that can be completed by calling functions according to instructions and implicit policies. From a practical application perspective, this assumed automation agent has a workflow shown in Figure 2. This benchmark skips the preprocessing process and simulates the situation in which the agent receives the extracted instruction and prepared files and starts to take action. The action is calling the tools/functions provided by coding (which is more flexible and stable than other function calling methods (Wang et al., 2024)) to implement the revision instruction in one turn without human participation.

3.1 Task Collection

Over 100+ real-world drawing revision files (provided by design firms and construction companies) were collected and comprehensively analyzed. The target tasks were filtered out and categorized into three elements: text, table and vector entity, and four operations: adding, content modification, mapping, and format updating, in total 12 types. Detailed descriptions of them can be found in Appendix B. The difficulty of each task varies greatly and is generally controlled by six parameters in four dimensions:

Difficulty in understanding structured data Since these documents come from different companies and are handled by different people, the language style or expression of the instructions varies greatly, leading to a dynamic difficulty in understanding embedded data. They can be either extremely clear and concise sentences, which are named structured language in this work, or verbose paragraphs providing more and sometimes abundant information, namely unstructured language.

Difficulty in function execution There are different function calling pipelines for each type of task. But their complexity varies greatly. Figure 3ab illustrates the pipelines for adding text and adding vectors, respectively, as examples. It can contain only straightforward steps in relatively easier tasks, like adding text, or a graph structure with richer nodes and edges for tasks like adding vectors.

Difficulty of instruction following The number of objects and revisions involved makes the length and complexity of an instruction fluctuate greatly. The impact of multiple objects and multiple revision operations on the implementation of an instruction is shown in Figure 3c.

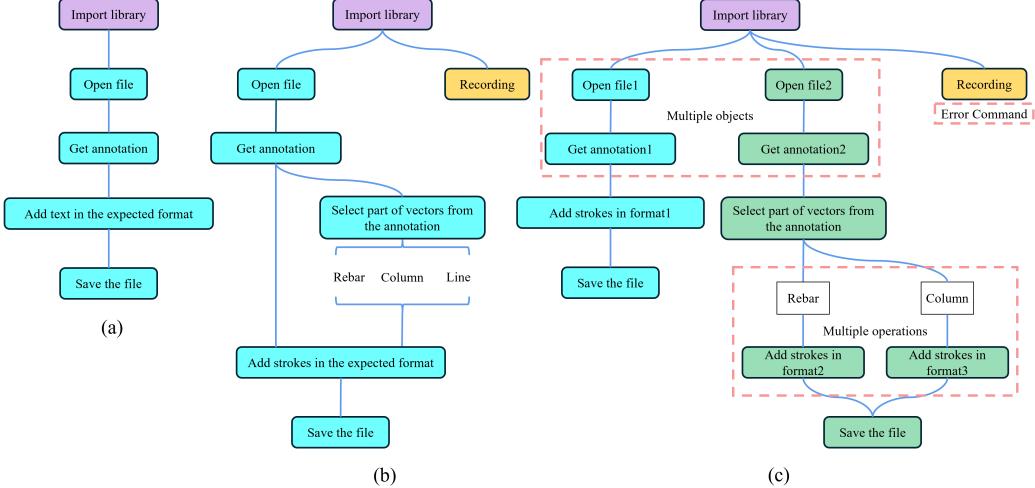


Figure 3: An illustration of function calling pipelines for different situations: (a) A standard pipeline for adding texts. (b) A standard pipeline for adding vectors. (c) Execution details for an instruction with multiple objects and long operation chains in one turn.

Difficulty of critical reasoning Due to the stochastic nature of human manual work, the quality of instructions is not always maintained at a constant standard. There are two types of errors that frequently appear. Sometimes, detailed values are not clearly specified, like “Move the table to the left a little”, that is, vaguely defining the value. It also happens that some necessary information is not specified, such as “add the missing dimension” without specifying the content or value to be added, which is an incomplete(error) instruction. The fully automated workflow expects the agent to respond to the above errors in two ways. The agent should think carefully and self-correct the ambiguous details with reasonable values. It is better to record incomplete instructions in the logging files using specified tools to alert the user rather than implement them anyway and make things worse.

The six parameters are summarized in Table 1. When constructing the benchmark, all filtered instructions were first grouped according to the six parameters. Tasks were sampled directly for groups with adequate tasks as candidate tasks or supplied with synthesized instructions for groups with fewer tasks. Then, the candidate tasks were verified by humans to ensure the solvability of the instructions and the uniformity of the value distribution. As a result, 1920 tasks were obtained with five tasks for each combination of difficulty parameters, as shown in Table 1.

Table 1: Task Specifications

Items	Parameter details	Number
Task category	12 types of tasks	12
Object per instruction	Single object/Multiple objects	2
Operation per instruction	Single operation/Multiple operations	2
Information completeness	Complete/Incomplete	2
Value specifying style	Precisely/Vaguely	2
Language style	Structured/Unstructured	2
Tasks per situation		5
Total		1920

3.2 Tool Preparation

Drawing revision tools/functions The tools/functions provided for the agent were specifically tailored for PDF drawing revision in this work, a total of 46 tools. The well-known PDF editing libraries, including PyMuPDF (Artifex, 2025) and Reportlab (ReportLab, 2025), and computer vision libraries, such as cv2 (OpenCV-Team, 2025) and pyresseract (Hoffstaetter, 2025), are introduced when designing them.

Dual tools/functions As it is difficult to conduct an accurate evaluation of the performance of models directly from the output drawings, dual tools/functions are introduced. They have function names, calling methods, arguments, and output types the same as the corresponding original functions. When the agent executes the generated code, the dual functions are executed instead of the original functions. They do not modify the drawing but record the argument details and the ground operations path, following the rules, such as repeated operations being recorded as one. They allow and record common coding errors that are not allowed in the original functions, such as parameter type errors. These records will be compared with the ground truth for analysis at the operational level. It can clearly estimate the operation quality, eliminate the noise introduced by coding styles, and easily distinguish unclean paths even though they have output drawings that are the same as the ground truth paths. In summary, the dual tools facilitate an accurate and comprehensive assessment of the mode while ensuring the uniqueness of the ground truth and considering the flexibility of the coding.

3.3 Default Prompt

After several experiments and adjustments, we have obtained a prompt framework that can effectively solve the tasks of drawing revision by coding with function calling, following the instructions and implicit policies in this benchmark. The default prompts for each task can be found in Appendix D. We also release the editing privileges of the prompts to encourage users to develop their own prompts to achieve higher performance.

4 Evaluation Metric

The evaluation metric was developed to evaluate both the task accuracy and the error statistics of the agent’s performance. The operation paths recorded by dual functions are compared with the ground truth paths. For each task, its score is graded by three parts: code executability, target completeness (completeness of the instruction).

The evaluation has two levels: *Level 1*, check whether a generated code can be run with dual functions. The dual functions are compatible with some common coding errors. Thus, there must be significant errors in an unrunnable code string and it will receive a score of 0. *Level 2*, when a code string can be run with dual functions. Each function execution is monitored for executability and will be recognized as executable if and only if (a) all required arguments are specified and (b) all specified arguments are in the correct data format. The executability score of the response is graded as 0 if any function is non-executable; otherwise, it is 30. The target completeness is assessed in six subtasks described below.

Argument defining: define arguments according to the details in the instruction.

Variables transferring: transfer intermediate variables between functions.

Function calling: call functions or tools.

(Single) Tool selection: select a suitable tool from the tool library.

Multi-tool selection: select a series of tools in sequence from the tool library.

Plan execution: The critical details of a sequence of functions corresponding to an object’s revision operations.

Figure 4 illustrates the six subtasks. The scores for the first five subtasks were calculated using Equation 1.

$$\frac{TP - FP}{TP + FN} \times \frac{70}{6} \quad (1)$$

$$\sum_{(p^p, p^g)} IoU(p^p, p^g) \times \frac{70}{6} \quad (2)$$

$$mean(s_i) - \frac{(100 - min(s_i))}{n} \quad (3)$$

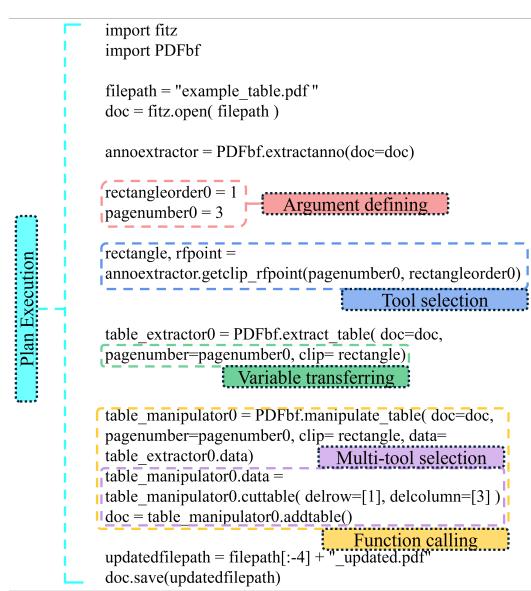


Figure 4: Subtasks when calling a function by coding to revise a drawing.

Table 2: Average task scores for different task sets and results of the comprehensive evaluations. Bold and underlined numbers are the first and second-highest performances in each category.

		OpenAI o1	ChatGPT-4o-2024-08-06	Claude3.5-sonnet	Deepseek-v3-685B	Qwen2.5-72B-Instruct	Llama3-70B-Instruct
Language style	Structured language	81.58	75.14	74.34	<u>76.26</u>	75.31	70.71
	Unstructured language	82.26	73.84	<u>78.20</u>	75.84	74.23	70.62
Details ambiguity	Precise details	89.82	79.46	<u>81.15</u>	79.25	76.18	73.50
	Vague details	74.02	69.52	71.39	<u>72.86</u>	73.36	67.84
Instruction completeness	Complete instruction	81.43	79.63	82.00	<u>82.74</u>	84.49	80.96
	Incomplete instruction	82.41	69.36	<u>70.54</u>	69.37	65.06	60.38
Objects number	Single object	83.23	73.87	<u>75.36</u>	75.28	75.07	69.57
	Multiple objects	80.60	75.11	<u>77.19</u>	76.83	74.47	71.77
Maximum operation chain length	Single operation	82.06	75.46	76.15	<u>77.21</u>	76.26	72.52
	Multiple operations	81.63	72.56	<u>76.52</u>	73.74	71.79	66.97
Average tasks score		81.92	74.49	<u>76.27</u>	76.05	74.77	70.67
Comprehensive score		79.90	71.76	<u>73.79</u>	73.09	72.05	67.55

TP refers to the number of subtasks in a response that are the same as the ground truth, and vice versa for *FN*. *FP* is the number of subtasks that are not in the ground truth, but are performed in the response. The drawing revision task is a result-sensitive task. If an agent performs unexpected operations, they are likely to introduce new errors and make things worse. Therefore, *FP* is introduced as a penalty when scoring the response.

The intersection over Union (IoU) was employed to score the plan execution. p is a sequence of revision operations performed on an object. p^g refers to the ground truth, and p^p refers to the response of the agent. Equation 2 was employed to score plan execution. The total score for all subtasks is 70. The sum of executability and target completeness score is the total score of the agent’s response to an instruction.

The comprehensive score of the agent is calculated by Equation 3, where s_i is the average score for the i -th type of task. The second part is introduced to better consider the weak points of the agent. If we simply take the average score of the 12 tasks as the comprehensive score, it is impossible to distinguish between the following two situations: 1) Agents scored high in some sets and low in others. 2) Agents scored evenly on each type of task. These two situations may have similar comprehensive scores, but the latter is preferred in the industry scenario.

5 Experimental Results

We test various state-of-the-art commercial and open-source language models for agents through their APIs: OpenAI GPT API (o1, gpt-4o-2024-08-06) (OpenAI, 2024), Anthropic Claude API (claude-3-5-sonnet-20241022) (Anthrop, 2024), Deepseek API (deepseek-chat) (Deepseek, 2024), Deepinfra API (Qwen/Qwen2.5-72B-Instruct, Meta-Llama-3-70B-Instruct) (Qwen-Team, 2024; Meta, 2024).

5.1 Task Accuracy and Robustness

Table 2 illustrates the main results. First, OpenAI o1 leads the performance in almost all task sets with obvious dominance compared to other models. There are two models in the second tier, Claude3.5-sonnet and Deepseek-v3-685B, which are neck-and-neck for the second-highest performance. Second, even the widely recognized powerful OpenAI o1 fails to earn around 20 points for simple, monotonous, and low-tech industrial tasks, highlighting the necessity of evaluation centered on industrial applications. Third, the performance of all models fluctuates in tasks with different difficulty, especially for tasks with some sensitive parameters.

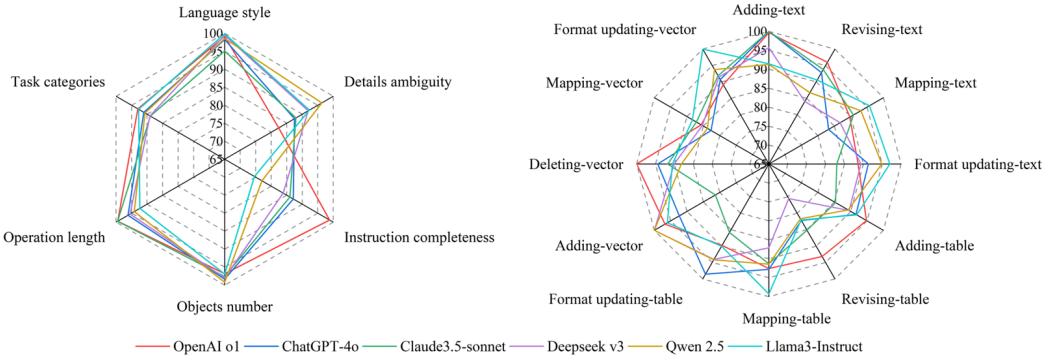


Figure 5: An illustration of performance degradation in terms of six parameters (left) and detailed performance of each task type (right). The values in the graph are the percentage of the model’s performance in the weaker task set versus its stronger task set.

To dissect the capabilities of LLMs to automate drawing revision tasks and to gain insight into their robustness, the agent’s performance in tasks of different complexity is shown in Figure 5. From the robustness of **structured data understanding** in different language styles, almost all models show great stability with an average degradation of 1%, except for the Claude3.5-sonnet (5%). For the robustness of **function execution**, the performance degradation of each type of task versus the best type of task for each model is around 9%. No significant gap was observed between the models. The relatively most stable model, OpenAI o1, decreases by 7%, and the least stable model, Claude3.5-sonnet, decreases by 11%. There is an interesting observation in Figure 5 that the best performance occurs in tasks having the least number of arguments (deleting vectors – o1, adding vectors – Deepseek-v3-685B, Qwen2.5-72B-Instruct) or having the most simple pipeline (adding text – ChatGPT-4o-2024-08-06, Claude3.5-sonnet, updating vector format - Llama-3-70B-Instruct). From the robustness of **instruction following**, the good news is that all models exhibit good stability with a performance degradation of less than 4% when dealing with tasks involving multiple objects compared to only involving a single object. Meanwhile, when dealing with tasks having multiple operations, OpenAI o1 and Claude3.5-sonnet exhibit excellent stability with a degradation of less than 1%, while that of others is 6% on average. From the robustness of **critical reasoning**, all models somehow struggle to complete the tasks. When details are vaguely specified, the degradation of all models is great, 5% for Qwen2.5-72B-Instruct and 12% for others. When there are errors in the instructions, performance degradation reaches 18% for all models except OpenAI o1, which shows great resilience, in contrast.

5.2 Error Analysis

To have a clear understanding of where the errors stem from and to have a deeper understanding of the limitations of the models, we calculated the accuracy of the six subtasks mentioned in Section 4, as shown in Figure 6. Similar to the comprehensive score, OpenAI o1 leads the way in accuracy in almost all subtasks, followed by Claude3.5-sonnet and Deepseek-v3-685B. To benefit the reader’s reading experience, only a comprehensive analysis is provided here; a more detailed discussion, such as how the performance of six subtasks varies at task sets in different difficulties, can be found in Appendix C.

In Figure 6, there is an obvious gap between the accuracy of plan execution and the other five subtasks. It is interesting that the difference is constantly around 20% for all models, though the value of their subtask accuracy varies greatly.

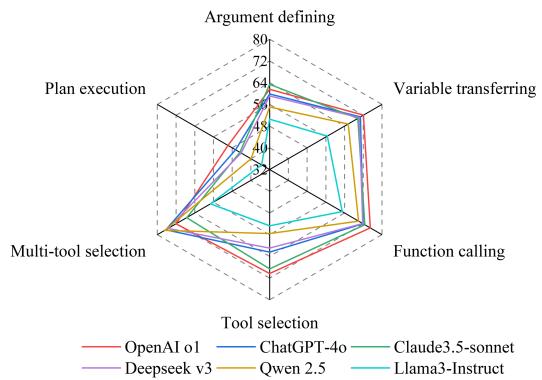


Figure 6: Subtasks accuracy(%) of tested models.

OpenAI o1, one of the most advanced models in the world, shows the same gap, even though it exceeds other models in almost all other subtasks. The plan execution was scored by checking whether all critical details of a sequence of tools to modify an object were consistent with the ground truth. Therefore, it can be deduced that the models are trying their best to complete each subtask with high accuracy, but it is difficult for them to pay enough attention to every detail. The models are able to understand a significant portion of the task objectives, but still lack the ability to avoid all errors throughout the modification process of an object.

5.3 Limitations and Future Directions

To fulfill the requirements of automating monotonous, low-tech, and high-labor-intensity tasks in the industry, like drawing revision tasks, there are still several challenges that need to be considered in the future development of LLMs.

Interactive style In our experiments, when LLMs encounter an incomplete instruction, they prefer to ask the user for more information or to fill the missing information with a placeholder to give the output rather than record it according to the polices in the system information. This contributes to the low accuracy in the incomplete instruction task set. It also corroborates the fact that current LLMs are paranoid about instantaneous interaction with humans. Instantaneous interaction is indeed an important scenario for the application of LLMs. But in real engineering applications, especially when automating industrial tasks, there are also scenarios that prefer no human involvement, such as the drawing revision task considered in this benchmark. Although LLMs are evolving toward super-intelligence and seamless interaction with humans, it is also important to maintain compatibility with other modes of interaction, such as delayed, indirect interaction through preprocessing and postprocessing files between LLMs and humans.

Understanding of details LLMs are in trouble when dealing with instructions that vaguely define key arguments. Common errors include filling in placeholders or directly filling in the description words in the instructions for vaguely defined arguments. Therefore, it is necessary to strengthen LLMs to speculate, reason, and understand the user’s exact intention behind the instruction details rather than literally executing them and giving a response for reference only.

Implementation of new policies LLMs perform quite stubbornly when they meet instructions that are in some kind of conflict with their intrinsic policy. For example, during the testing of this benchmark, the system information in the prompt repeatedly emphasizes the user’s expected response when encountering detail-ambiguous instructions and error instructions. They can obey the police in some tasks, but also ignore the new policies in a significant number of subtasks and give the unexpected responses mentioned above, in accordance with their habits or intrinsic policy. Having and remaining loyal to intrinsic policies is a necessary competency for LLMs, especially when it comes to cybersecurity or illegal risk. However, in some other less serious and stereotypical aspects, there is a need to improve LLMs’ tolerance and implementation of new policies, especially considering the diverse demands and polices of industrial application scenarios. More benchmarks are also needed to illustrate more requirements for integrating LLMs in industrial scenarios.

6 Conclusion

In this work, we introduce DrafterBench, an open-source toolkit designed to offer a systematic and comprehensive evaluation of the capabilities of LLMs to automate monotonous, low-tech, and high-labor-intensity tasks from industry. We emphasize the challenges and rigorous requirements for AI agents from an industrial perspective. DrafterBench is under the context of a representation civil engineering task, drawing revision, a total of 1920 drawing revision tasks are collected from real documents. The complexity of tasks is controlled by six parameters to investigate four essential capabilities of LLMs, namely structured data understanding, function execution, instruction following, and critical reasoning. The automatic evaluation toolkit assesses the performance of LLMs with task accuracy and error statistics. We conducted experiments on mainstream LLMs to reveal their strengths and deficiencies in automating industry tasks. From the result, we posit that DrafterBench is a useful metric that can provide valuable evaluation and useful insight for the future development of LLMs from the perspective of industry, especially Civil Engineering.

References

- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, and Karol Hausman. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.
- Anthrop. Claude3.5-sonnet, 2024. URL <https://www.llama.com/>.
- Artifex. Pymupdf, 2025. URL <https://pypi.org/project/PyMuPDF/>.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, and Piotr Nyczek. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2023. ISBN 2374-3468.
- Deepseek. Deepseek v3, 2024. URL <https://www.deepseek.com/>.
- EvolveLab. Glyph copilot, 2025. URL <https://www.evovelab.io/glyph>.
- S. Hoffstaetter. pytesseract, 2025. URL <https://pypi.org/project/pytesseract/>.
- Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 2024.
- Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, and Agustin Dal Lago. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. ISSN 0036-8075.
- Jiepeng Liu, P Liu, L Feng, W Wu, and H Lan. Automated clash resolution of rebar design in rc joints using multi-agent reinforcement learning and bim. In *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, volume 36, pages 921–928. IAARC Publications, 2019.
- 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. *ArXiv*, abs/2401.13178, 2024.
- Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. *ArXiv*, abs/2404.11584, 2024.
- Meta. Llama3.3, 2024. URL <https://www.llama.com/>.
- Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. *ArXiv*, abs/2311.12983, 2023.
- OpenAI. Openai 01, 2024. URL <https://openai.com/index/learning-to-reason-with-llms/>.
- OpenCV-Team. Opencv, 2025. URL <https://pypi.org/project/opencv-python/>.
- Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, and Bill Qian. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

- Qwen-Team. Qwen2.5, 2024. URL <https://qwen2.org/qwen2-5/>.
- ReportLab. Reportlab, 2025. URL <https://docs.reportlab.com/>.
- Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. Learning to use tools via cooperative and interactive agents. *arXiv preprint arXiv:2403.03031*, 2024.
- Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, and Rong Yao. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*, 2023.
- Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.
- Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, and Adrià Garriga-Alonso. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.
- Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *ArXiv*, abs/2309.10691, 2023.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- Yue Wu, So Yeon Min, Yonatan Bisk, Ruslan Salakhutdinov, Amos Azaria, Yuanzhi Li, Tom Mitchell, and Shrimai Prabhumoye. Plan, eliminate, and track–language models are good teachers for embodied agents. *arXiv preprint arXiv:2305.02412*, 2023a.
- Yue Wu, Xuan Tang, Tom M Mitchell, and Yuanzhi Li. Smartplay: A benchmark for llms as intelligent agents. *arXiv preprint arXiv:2310.01557*, 2023b.
- Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*, 2023.
- Siqiao Xue, Tingting Chen, Fan Zhou, Qingyang Dai, Zhixuan Chu, and Hongyuan Mei. Famma: A benchmark for financial domain multilingual multimodal question answering. *arXiv preprint arXiv:2410.04526*, 2024.
- Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard, 2024. URL https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
- Guoli Yin, Haoping Bai, Shuang Ma, Feng Nan, Yanchao Sun, Zhaoyang Xu, Shen Ma, Jiarui Lu, Xiang Kong, and Aonan Zhang. Mmau: A holistic benchmark of agent capabilities across diverse domains. *arXiv preprint arXiv:2407.18961*, 2024a.

Guoli Yin, Haoping Bai, Shuang Ma, Feng Nan, Yanchao Sun, Zhaoyang Xu, Shen Ma, Jiarui Lu, Xiang Kong, and Aonan Zhang. Mmau: A holistic benchmark of agent capabilities across diverse domains. *arXiv preprint arXiv:2407.18961*, 2024b.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, and Indraneil Paul. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Limitation and Computing Resources

A.1 Limitation

The tasks involved in this work are all in English. Multilingual input will be included in future work to consider the ability of LLMs to automate industrial tasks in non-English environments. At the same time, this work covers only drawing revision tasks in the field of civil engineering, and more types of tasks are yet to be performed in future work.

A.2 Computing Resources

This benchmark calls LLMs via APIs and has no special requirements for hardware devices. In this experiment, the benchmark runs on an i9-13900F CPU for about 30 min to get all the results.

B Description of Drawing Revision Tasks

The detailed description of the 12 drawing revision tasks is shown in Table 3.

Table 3: 12 types of tasks for the drafter agent.

Object element	Operation	Description
Text	Adding	Add new texts to the target area.
	Content modification	Delete or replace words in the target text strings.
	Mapping	Move, rotate or scale the target texts.
	Format updating	Modify the format of target texts, including but not limited to text color, font type, font size, and alignment.
Table	Adding	Add new tables to the target area.
	Content modification	Delete, clear, or replace cells in the target tables.
	Mapping	Move, rotate or scale the target tables.
	Format updating	Modify the format of target tables, including but not limited to font type, font size, boundary width, and alignment.
Vector entity	Adding	Add standard strokes to the target position.
	Content modification	Delete target strokes, including rebars, columns, and lines.
	Mapping	Move, rotate or scale target strokes.
	Format updating	Modify the format of target strokes, including but not limited to line color, line thickness, and line type.

C Performance of Agents for Tasks in Different Difficulty

C.1 Structured Data Understanding

Table 4: Accuracy of subtasks in different language styles in percentage. STR means structured language, and U-STR means unstructured language.

	OpenAI o1		ChatGPT-4o-2024-08-06		Claude3.5-sonnet		Deepseek-v3-685B		Qwen2.5-72B-Instruct		Llama3-70B-Instruct	
	STR	U-STR	STR	U-STR	STR	U-STR	STR	U-STR	STR	U-STR	STR	U-STR
Argument defining	59.84	63.31	58.86	60.81	62.81	63.94	59.72	58.51	57.05	53.09	54.67	46.45
Variable transferring	71.03	73.13	70.95	70.62	72.67	66.70	72.92	67.64	68.44	62.68	63.36	49.97
Function calling	73.20	76.85	71.63	72.24	73.78	71.56	72.28	71.27	70.92	69.08	68.58	57.43
(Single) Tool selection	68.31	72.28	63.28	61.57	67.30	69.87	62.58	59.31	57.44	53.72	52.72	52.72
Multi-tool selection	70.15	73.86	75.38	77.40	76.22	58.68	75.89	74.20	77.91	75.72	77.07	37.10
Plan execution	49.94	49.59	48.09	45.09	47.11	42.45	45.29	43.07	41.64	38.27	39.57	31.75

Table 4 lists agents’ performance for instructions in structured and unstructured language. The stable scores of all LLMs show their excellent adaptability to different language styles. The difference in both comprehensive scores and subtasks between the two language styles is around 5%.

C.2 Instruction Following

Objects per instruction The number of objects per instruction determines whether the pipeline will be implemented in parallel. As shown in Table 5, the good news is that LLMs perform stably for both a single object per instruction and multiple objects per instruction. An interesting observation is that models except OpenAI o1 perform better for multi-object instruction, which is actually more difficult. However, the increase in accuracy in all subtasks is not due to better performance but to a lower requirement for “Recording” incomplete instructions. For an incomplete instruction with a single object, the only ground truth is to call the “Recording” function; it will be 100% false if it is not called. In contrast, for a multi-object instruction, “Recording” should only be called for objects that lack necessary information, and the standard pipeline should be executed normally for others. Therefore, LLMs can get higher scores even though none of the “Recording” is called in the multiple objects instructions.

Table 5: Accuracy of subtasks for instructions with a single object (SIN) or multiple objects (MULT).

	OpenAI o1		ChatGPT-4o-2024-08-06		Claude3.5-sonnet		Deepseek-v3-685B		Qwen2.5-72B-Instruct		Llama3-70B-Instruct	
	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT
Argument defining	62.65	60.93	53.77	63.40	60.72	64.93	50.70	64.05	46.25	60.24	34.89	59.76
Variable transferring	72.45	71.90	67.93	72.17	70.73	69.17	64.44	73.12	61.34	67.61	37.68	65.90
Function calling	76.64	74.13	68.22	73.98	73.95	71.97	66.04	74.94	65.13	72.68	47.08	71.78
(Single) Tool selection	74.73	67.00	62.10	62.66	70.48	67.18	58.62	62.66	54.84	56.13	49.94	54.78
Multi-tool selection	67.95	73.00	67.52	78.57	68.38	67.23	59.40	79.31	64.10	79.94	0.00	79.20
Plan execution	41.13	54.80	38.17	51.64	37.41	48.85	34.23	50.34	31.58	45.17	24.97	42.55

Operations per object The number of operations per object changes the execution loop of sub-steps in the standard pipeline. The LLMs demonstrate some adaptability to varying operations. The difference between instructions that perform only one operation on an object and those that perform multiple operations is about 5% for all subtasks. It is noted that some local drops also occur in multi-tool selection, as shown in Table 6.

Table 6: Accuracy of subtasks for instructions with a single operation per object(SIN) or multi-operations per object(MULT).

	OpenAI o1		ChatGPT-4o-2024-08-06		Claude3.5-sonnet		Deepseek-v3-685B		Qwen2.5-72B-Instruct		Llama3-70B-Instruct	
	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT	SIN	MULT
Argument defining	64.64	59.58	58.50	61.86	61.52	66.23	61.87	57.33	55.29	54.63	52.49	49.29
Variable transferring	71.20	72.74	72.51	68.47	67.96	72.00	67.77	72.15	68.76	61.28	57.02	56.40
Function calling	75.16	74.93	73.07	70.22	70.66	75.71	71.85	71.73	71.92	67.09	63.60	62.61
(Single) Tool selection	70.14	70.41	64.14	59.91	69.81	66.79	59.54	61.90	58.51	51.30	48.39	55.67
Multi-tool selection	72.71	71.57	76.79	75.76	59.62	73.36	77.73	73.90	77.88	75.11	68.34	36.54
Plan execution	53.45	47.26	44.51	49.42	41.93	48.82	46.37	42.55	38.56	41.92	39.63	32.75

C.3 Critical Reasoning

As mentioned in the main body, there are two situations that require LLMs critical reasoning to give a correct response, which are incomplete instructions and vague details. For most LLMs, switching from specifying values precisely to vaguely decreases the accuracy of all subtasks (Table 7). The most obvious difficulty introduced is the need to assume arguments in a reasonable way. The most common error is simply filling in the description text instead of assuming a reasonable value. For

Table 7: Performance of agents for instructions specifying values precisely (P)/vaguely (VA).

	OpenAI o1		ChatGPT-4o-2024-08-06		Claude3.5-sonnet		Deepseek-v3-685B		Qwen2.5-72B-Instruct		Llama3-70B-Instruct	
	P	VA	P	VA	P	VA	P	VA	P	VA	P	VA
Argument defining	84.53	38.87	69.88	49.91	72.12	54.72	65.94	52.38	57.43	52.72	51.16	49.96
Variable transferring	90.58	55.48	79.26	63.18	75.38	64.57	72.79	68.03	65.38	65.72	52.64	60.28
Function calling	93.15	58.17	79.46	64.94	76.72	68.91	75.46	68.35	69.99	70.01	58.88	66.84
(Single) Tool selection	91.04	50.73	68.91	56.30	73.47	63.98	63.26	58.75	54.66	56.45	53.11	52.35
Multi-tool selection	90.47	55.71	86.87	67.14	70.86	64.44	82.19	69.37	78.06	75.71	38.67	73.65
Plan execution	63.14	38.39	54.90	38.59	51.04	38.17	49.42	39.06	43.79	36.16	38.37	32.45

Command: In file H7L3M.pdf, on page 7, rotate the table in rectangle 4 by a noticeable angle counterclockwise, and scale the table down in rectangle 6 by a slight amount.

Response:

```
import fitz
import PDFbf

#Task 1: Rotate the table in rectangle 4 on page 7 by a "noticeable angle counterclockwise" (magnitude missing)
missing_info0 = "file-level:H7L3M.pdf, page-level:7, order-level:4, base-level: rotate the table in rectangle 4 by a noticeable angle counterclockwise (incomplete)"*
recorder0 = PDFbf.recorder(missing_information=missing_info0)
recorder0.recording()

#Task 2: Scale table in rectangle 6 on page 7 by a "slight amount" (magnitude missing)
missing_info1 = "file-level:H7L3M.pdf, page-level:7, order-level:6, base-level: scale the table down in rectangle 6 by a slight amount (incomplete)"*
recorder1 = PDFbf.recorder(missing_information=missing_info1)
recorder1.recording()
```

Figure 7: An example of incorrectly recoding vague instruction.

example, when the instruction asks for a “general font color”, the LLMs will define a variable as follows: “`fontcolor=“general color”`”. Besides, OpenAI o1 is the most sensitive model to vague instructions. Considerable decrease exists not only in argument defining, but also in function calling and tool selection. This is because some values vaguely described are wrongly recognized as missing necessary information. Some vague instructions are incorrectly treated as error instructions. Here is an example response generated by OpenAI o1 shown in Figure 7.

Consistent with the analysis in the main body, there is a very significant decrease in the performance of the models on the tasks set with error instructions in almost all subtasks, except OpenAI o1 (Table 8). However, the reason for OpenAI o1’s stable comprehensive score is not a stable accuracy for all subtasks, but there is a significant decrease in plan execution and a small increase in the other subtasks. This means that, although OpenAI o1 can apply the correct response pattern to erroneous instructions, it remains or even becomes more challenging for o1 to fully align with the instructions in detail. This results in more imperfect modifications.

Table 8: Performance of agents for complete instructions (Com) and incomplete instructions (ERR).

	OpenAI o1		ChatGPT-4o-2024-08-06		Claude3.5-sonnet		Deepseek-v3-685B		Qwen2.5-72B-Instruct		Llama3-70B-Instruct	
	Com	ERR	Com	ERR	Com	ERR	Com	ERR	Com	ERR	Com	ERR
Argument defining	60.04	63.86	68.66	46.64	71.43	51.33	72.78	38.68	75.72	24.19	74.09	15.37
Variable transferring	71.41	73.52	77.09	57.18	74.56	59.16	82.40	44.13	82.62	28.75	78.91	8.64
Function calling	72.98	78.54	78.41	60.85	76.74	65.69	82.95	52.64	85.94	42.68	84.24	26.63
(Single) Tool selection	66.69	75.59	72.27	47.95	78.23	54.41	78.61	34.97	80.52	18.94	78.78	14.41
Multi-tool selection	69.97	76.15	78.64	71.79	63.69	75.13	79.15	67.69	84.42	61.28	82.91	4.87
Plan execution	59.66	35.87	60.51	30.11	56.90	29.72	61.18	26.40	59.36	21.97	57.72	17.31

D Default Prompts

All default prompts have four components: task background, standard pipelines(task plan), available functions, and examples:

Task background: This section describes the role of LLM and provides prior knowledge and implicit policy. It includes definitions of terminology, terms defined by the tool designer or user, and other necessary information.

Standard pipeline: This section is important in improving the agent’s performance, especially in integrating and utilizing functions. It describes a graph containing multi-mode operations to implement the revision of an object, such as branch shift, step skipping, single tool selection, and multi-tool selection from a sub-library. Instead of working as a plan for a particular task, it works for all possible situations. It supports parallel implementation for instructions with multiple objects, or loop execution for an object with long operations.

Available functions: Detailed information about the available functions is provided in this section, such as function names, calling statements, application scenarios, and the function’s essential arguments, optional arguments, input data types, and output results. It is also possible to provide complementary implicit policies if necessary.

Examples: After experimentation, it was found that it is still a challenge for LLMs to follow the pipeline and call functions to complete the instruction in a zero-shot manner. Thus, one or several examples are recommended to help illustrate how pipelines are implemented.

Here are the default prompts:

D.1 Adding Text

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks. Now, the drafter wants to insert text on the target position boxed by rectangle markups. One string of text is seen as one task. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the text string) from high to low. A high-level information can correspond to multiple low-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't make assumptions if the previous four necessary information are missing, but you can assume a rational value according to your knowledge if other information is missing. Don't assume the value of necessary information, but identify tasks lacking necessary information, record the missing details, and execute the task only once all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Incomplete information recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the remaining operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information can be found.

1. Import libraries

2. Open the file.

3. Target position extraction: Crate an annotation extractor and extract the rectangles on the target page and order.

4. Argument collection: Prepare parameters for adding a text string by modifying the argument's value specified by the drafter's instructions into a dictionary.

5. Text adding: Create a text manipulator instance and add text.

6. Save the file you have opened as a new file with an updated name.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the text manipulator instance. Here is an example dictionary including all possible attributes:

argument = {"Clip": extracted target rectangle; "text": a string of new texts, "font": a string refers to the font of the text; "fontsize": an int refers to the font size; "textcolor": a string refers to the color of the text; "rotate": an int in 90, 180, 270, refers to the rotation of the text, "align": a fitz_constant selected from `fitz.TEXT_ALIGN_RIGHT` or `fitz.TEXT_ALIGN_LEFT` or `fitz.TEXT_ALIGN_CENTER` or
`fitz.TEXT_ALIGN_JUSTIFY` refers to the align of the text}

The drafter will selectively parameters such as font, font size, text color, rotation, alignment, etc. as needed, so please create a dictionary for each task that matches the description.

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz  
import PDFbf"
```

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```
"  
updated_file = filepath[:-4] + "_updated.pdf" # update file path  
doc.save(updated_file)  
"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

```
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
```

Get a rectangle and its reference points:

```
"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"
```

Text adding:

A text manipulator is instantiated by specifying the essential arguments (document(doc), pagenumber, clip, text) and mentioned optional arguments (font, fontsize, text color, rotate, align). Here is an example with all possible arguments mentioned:

```
"Text_manipulator=PDFbf.manipulate_text(doc=fitz.document, pagenumber=int, clip=rectangle, text=string, font=string, fontsize=int,  
textcolor=string, rotate=int, align=fitz_constant)"
```

You can get a doc with texts added by calling the following function:

```
"doc=Text_manipulator.addtext()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in the original instruction following with an "incomplete" in brackets (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording"
```

Here is an example and you should respond in a similar way.

User: The file name is "a123gfsdfd_text.pdf". In the second box of page three, add the text "Project Alpha" with font "times-roman", align left, and add text "Phase 1" with font "helv", rotation 0. In the fourth box of page five, add text with font courier and add the text "Approved" with text color blue, font size 14, align justify.

Response:

```
1 import fitz
2 import PDFbf
3
4 filepath = "a123gfsdfd_text.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8 pagenumber0=2
9 rectangleorder0=1
10 rectangle0, rfpoin0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
11
12 argument0 = dict(clip=rectangle0, text="Project Alpha", font="times-roman", align=fitz.TEXT_ALIGN_LEFT)
13 Text_manipulator0 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
14 text=argument0.get("text"), font=argument0.get("font"), align=argument0.get("align"))
15 doc = Text_manipulator0.addtext()
16
17 argument1 = dict(clip=rectangle0, text="Phase 1", font="helv", rotate=0)
18 Text_manipulator1 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber0, clip=argument1.get("clip"),
19 text=argument1.get("text"), font=argument1.get("font"), rotate=argument1.get("rotate"),
20 align=argument1.get("align"))
21 doc = Text_manipulator1.addtext()
22
23 missing_inf2 = '''file-level: a123gfsdfd_text.pdf, page-level: 5, order-level: 4, base-level: add text with font
24 courier(incomplete)'''
25 recorder2 = PDFbf.recorder(missing_information=missing_inf2)
26 recorder2.recording()
27
28 pagenumber1=4
29 rectangleorder1=3
30 rectangle1, rfpoin1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1, rectangleorder=rectangleorder1)
31 argument3 = dict(clip=rectangle1, text="Approved", textcolor="blue", fontsize=14, align=fitz.TEXT_ALIGN_JUSTIFY)
32 Text_manipulator3 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber1, clip=argument3.get("clip"),
33 text=argument3.get("text"), textcolor=argument3.get("textcolor"), fontsize=argument3.get("fontsize"),
34 align=argument3.get("align"))
35 doc = Text_manipulator3.addtext()
36
37 updatedfilepath = filepath[:-4] + "_updated.pdf"
38 doc.save(updatedfilepath)
```

D.2 Revising Text

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks.

Now, the drafter wants to edit the text content on the target area boxed by rectangle markups. Editing text on a target position is seen as a task. There are two possible operations and at least one of them must be selected: deleting and replacing. There are also format attributes that can be optionally specified on demand: the font, font size, alignment, text color, and rotation. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the necessary information of the operation) from high to low. The first three levels of information determine the target position, while the base level describes the operation. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the four level information corresponding to each task from an instruction that may contain multiple tasks. Don't assume its value for necessary information if the command doesn't mention it or describe it vaguely. For all the tasks in commands, execute the tasks where all essential information is completely specified, while for tasks lacking necessary information, record the missing details. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.

1. Import libraries
2. Open the file.

3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for updating text content by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Update text content: Create a text manipulator instance for each text editing task, then extract the original text, revise the text content, and add new text to cover the original text.
6. Save the file.

Available tools:

Two constants defined by the user:

```
"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;
```

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the text editing task. Here is an example dictionary including all possible attributes:

```
{"clip": a rectangle boxes the text that needed to be revised; "font": a string refers to the font of the text; "fontsize": an int refers to the font size; "align": a constant get by ```fitz.TEXT_ALIGN_RIGHT``` or ```fitz.TEXT_ALIGN_LEFT``` or ```fitz.TEXT_ALIGN_CENTER``` or ```fitz.TEXT_ALIGN_JUSTIFY``` refers to the align of the text in box; "rotate": a int in 0, or 90 refers to the direction of the original text to be edited, 0 means horizontal text, 90 means vertical text, if the direction of the text is not mentioned, the text is horizontal;}
```

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:
 "import fitz
 import PDFbf"

File open:

You can open the file by calling the following function:
 "doc = fitz.open("filepath")" Open the file with the file path.

Save the updated file:

You can save the file with an updated name by calling the following function:

```
"  

updated_file = filepath[:-4] + "_updated.pdf" # update file path  

doc.save(updated_file)  

"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

```
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
```

Get a rectangle and its reference points:

```
"rectangle, rpoint = annoextractor.getclip_rpoint(pagenumber=int,rectangleorder=int)"
```

Update text content:

A text manipulator is instantiated by specifying arguments (document(doc), page number(pagenumber), clip) and optional arguments (font, fontsize, text color, rotate, align) that are mentioned like this:

```
"Text_manipulator=PDFbf.manipulate_text(doc=fitz.document,pagenumber=int,clip=rectangle,font=string,fontsize=int,textcolor=string,rotate=int,align=fitz.TEXT_ALIGN_LEFT)"
```

You can extract the original horizontal text by calling the following function:

```
"Text_manipulator.text=Text_manipulator.gethorizontext()
```

You can get a string of original vertical text by calling the following function:

```
"Text_manipulator.text=Text_manipulator.getverttext()
```

Here are some arguments that should be specified when you want to delete or replace specified texts from the original texts:

"deltex": a string refers to the text the user wants to delete from the original text;

"retext": a string refers to the text the user wants to replace from the original text;

"totext": a string refers to the text the user wants to use to replace the original text;

You can delete specified text from the string of original text by defining the text to delete(deltex), and calling the following function:

```
"Text_manipulator.text = Text_manipulator.deletetext(deltex=string)"
```

```
"Text_manipulator.text = Text_manipulator.deletetext(deltex="target text")" delete the text "target text".
```

```
"Text_manipulator.text = Text_manipulator.deletetext()" delete all the text if a delete action is required with no delttext specified.
```

You can replace specified text with certain text for the string of original text by defining the text to replace(retext), text used to replace(totext) and calling the following function:

```
"Text_manipulator.text = Text_manipulator.replace(text=retext,string,totext=string)"
```

```
"Text_manipulator.text = Text_manipulator.replace(text="old",totext="new")" replace the text "old" using text "new".
```

```
"Text_manipulator.text = Text_manipulator.replace(text="1236")" replace all the text using text "1236" if a replace action is required with no retext specified.
```

You can get a doc with original text covered by manipulated new text added by calling the following functions:

```
"doc = Text_manipulator.addtext()
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in the original instruction following with an "incomplete" in a brace (if there is some related description but some necessary information is missing)".

The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording"
```

Here is an example of executing the user's instructions. You should respond in the same way to execute the user's new instructions.

User: The file name is "42Del9iutete_text.pdf". For the text in the second box of page 5, delete "temporary and text". Replace the text with "assignment". Use left align. Font size 8.
For the text in the fourth box of page 12, replace "manager" with "supervisor". Delete "completed" and "new".
For the text in page 18, replace "due" with "scheduled". Delete "soon". Font size 12.

Response:

```
1 import fitz
2 import PDFbf
3
4 filepath = "42Del9iutete_text.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8pagenumber0 = 4
9 rectangleorder0 = 1
10 rectangle0, rfpoint0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
11
12 argument0 = dict(clip=rectangle0, align=fitz.TEXT_ALIGN_LEFT, fontsize=8)
13 Text_manipulator0 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
14 align=argument0.get("align"), fontsize=argument0.get("fontsize"))
15 Text_manipulator0.text = Text_manipulator0.gethortex()
16 Text_manipulator0.text = Text_manipulator0.deletetext(deltex="temporary and text")
17 Text_manipulator0.text = Text_manipulator0.replacetext(totext="assignment")
18 doc = Text_manipulator0.addtext()
19
20 pagenumber1 = 11
21 rectangleorder1 = 3
22 rectangle1, rfpoin1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1, rectangleorder=rectangleorder1)
23
24 argument1 = dict(clip=rectangle1)
25 Text_manipulator1 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"))
26 Text_manipulator1.text = Text_manipulator1.gethortex()
27 Text_manipulator1.text = Text_manipulator1.replacetext(retext="manager", totext="supervisor")
28 Text_manipulator1.text = Text_manipulator1.deletetext(deltex="completed")
29 Text_manipulator1.text = Text_manipulator1.deletetext(deltex="new")
30 doc = Text_manipulator1.addtext()
31
32 missing_information2 = '''file-level:42Del9iutete_text.pdf,page-level:18,order-level:missing,base-level:replace
33 "due" with "scheduled". Delete "soon". Font size 12.'''
34 recorder2 = PDFbf.recorder(missing_information=missing_information2)
35 recorder2.recording()
36 updatedfilepath = filepath[:-4] + "_updated.pdf"
37 doc.save(updatedfilepath)
```

D.3 Mapping Text

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks. Now, the drafter wants to change the position of text on the target area boxed by rectangle markups. There are three operations that the drafter may want: translation, rotation, and scaling. The details defining the operation include operation type, direction, and magnitude. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the details for at least one of the three operations) from high to low. What needs to be made more clear is that the parts of the details defining each type of operation that can be considered necessary information are pairs of information consisting of the type of operation and the direction, translation-direction, rotation-direction, and scale-factor or at least bigger or smaller than the original text, respectively. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks. Don't assume the value of missing necessary information (only the magnitude of operation can be assumed), but identify tasks lacking necessary information, record the missing details, and execute the task with all essential information gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for changing a text position by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Changing position: Create a text projector instance and change position.
6. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;

"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the text position-changing task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle boxes the text that needs to be dealt with; "move": a list refers to translation details, general form: [move in its right direction (positive value if move right, negative value if move left), move in its upper direction (positive value if move up, negative value if move down), reference_point], "rotation": a list refers to the rotation details, general form: ['r', rotation degrees in the clockwise direction (positive value in clockwise direction, negative value in counterclockwise direction), reference_point], if the direction of rotation is not specified, clockwise is considered; "scal": a list refers to the scaling, general form: ['sc', [zoom factor X direction, zoom factor Y direction], reference_point]}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz  
import PDFbf"
```

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```
"  
updated_file = filepath[:-4] + "_updated.pdf" # update file path  
doc.save(updated_file)  
"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

```
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
```

Get a rectangle and its reference points:

```
"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"
```

Changing position:

A projector is instantiated by specifying the arguments (document(doc), page number(pagenumber), clip) and 1-3(at least one) optional arguments (translation (move), rotation(rotation) and scaling(scal)) on demand like this:

```
"text_projector=PDFbf.Projector(doc=fitz.document,pagenumber=int,clip=rectangle,move=[move_in_right_direction, move_in_up_direction, reference_point],rotation=['r', rotation degrees in clockwise direction,reference_point],scal=['sc',[zoom factor in X direction, zoom factor in Y direction], reference_point])"
```

You can get a doc with the text mapped by calling the following function:

```
"doc = text_projector.project()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in the original instruction following with an "incomplete" in brackets (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording()"
```

Here is an example. You should respond in the same way.

User: Move the text in the second box on the tenth page. For the file "47_mapqweqping_text.pdf", move the text in the first box on page 5 20 units left,30 units up. rotate the text 45 degrees clockwise.

Response:

```
1 import fitz  
2 import PDFbf  
3  
4 missing_information0 = '''file-level:incomplete, page-level: 10, order-level: 2, base-level: Move the text in  
the second box on the tenth page.(incomplete)'''  
5 recorder0 = PDFbf.recorder(missing_information=missing_information0)  
6 recorder0.recording()  
7  
8 filepath1 = "47_mapqweqping_text.pdf"  
9 doc = fitz.open(filepath1)  
10  
11 annoextractor1 = PDFbf.extractanno(doc=doc)  
12 pagenumber1 = 4  
13 rectangleorder1 = 0  
14 rectangle1, rfpoin1 = annoextractor1.getclip_rfpoin(pagenumber=pagenumber1,rectangleorder=rectangleorder1)  
15  
16 argument1 = dict(clip=rectangle1, move=[-20, 30, rfpoin1], rotation=['r', 45, rfpoin1])  
17 text_project1 = PDFbf.Projector(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"),  
move=argument1.get("move"), rotation=argument1.get("rotation"))  
18 doc = text_project1.project()  
19  
20 updatedfilepath = filepath1[:-4] + "_updated.pdf"  
21 doc.save(updatedfilepath)
```

D.4 Updating Text Format

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks. Now, the drafter wants to update the text format on the target area boxed by rectangle markups. The possible format attributes are: font, font size, alignment, text color. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing at least one of the format attributes) from high to low. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information (if a format attribute is mentioned without specifying the value clearly, you can only assume that value), but identify tasks lacking necessary information, record the missing details, and execute the task only once all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Crate an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for updating text format by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Update format: create a text manipulator instance, extract the original text, and refresh the text format.
6. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the text position-changing task. Here is an example dictionary including all possible attributes:

```
{"clip": a rectangle boxes the text that needed to be updated; "font": a string refers to the font of the text; "fontsize":an int refers to the font size; "align": a constant get by ```fitz.TEXT_ALIGN_RIGHT``` or ```fitz.TEXT_ALIGN_LEFT``` or ```fitz.TEXT_ALIGN_CENTER``` or ```fitz.TEXT_ALIGN_JUSTIFY``` refers to the align of the text in box;"rotate": a int in 0, or 90 refers to the direction of the text, 0 means horizontal text, 90 means vertical text, if the direction of the text is not mentioned, the text is horizontal text; "textcolor": a string refers to the color of the text}
```

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz  
import PDFbf"
```

File open:

You can open the file by calling the following function:
"doc = fitz.open("filepath")" Open the file with the file path.

Save the updated file:

You can save the file with an updated name by calling the following function:

```
"  
updated_file = filepath[:-4] + "_updated.pdf" # update file path  
doc.save(updated_file)  
"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
Get a rectangle and its reference points:
"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"

Update format:

A text manipulator is instantiated by specifying arguments (document(doc), page number(pagenumber), clip) and optional arguments (font, fontsize, textcolor, rotate, align) on demand like this:

```
"Text_manipulator=PDFbf.manipulate_text(doc=fitz.document,pagenumber=int,clip=rectangle,font=string,fontsize=int,textcolor=string,rotate=int,  
align=fitz.TEXT_ALIGN_LEFT)"
```

You can extract the original horizontal text by calling the following function:

```
"Text_manipulator.text = Text_manipulator.gethorizontext()"
```

You can get a string of original vertical text by calling the following function:

```
"Text_manipulator.text = Text_manipulator.getvertext()"
```

After extracting the original text, you can get a doc with the original text covered by refreshed new text added by calling the following functions:

```
"doc = Text_manipulator.addtext()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is ""file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in the original instruction following with an "incomplete" in a brace (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording()"
```

Here is an example. You should respond in the same way.

User: The file name is "234_Manipdfgulate_text.pdf". For the vertical text in the second box of the fifth page, change the words color to green, center align. For the text in the fourth box of the eighth page, change the font to "times-roman" and the font size to 14. Update the text in the sixth box of the twelfth page.

Response:

```
1 import fitz
2 import PDFbf
3
4 filepath = "234_Manipdfgulate_text.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8 pagenumber0 = 4
9 rectangleorder0 = 1
10
11 rectangle0, rfpoin0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
12 argument0 = dict(clip=rectangle0, rotate=90, textcolor="green", align=fitz.TEXT_ALIGN_CENTER)
13 Text_manipulator0 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
    rotate=argument0.get("rotate"), textcolor=argument0.get("textcolor"), align=argument0.get("align"))
14 Text_manipulator0.text = Text_manipulator0.getvertext()
15 doc = Text_manipulator0.addtext()
16
17 pagenumber1 = 7
18 rectangleorder1 = 3
19 rectangle1, rfpoin1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1, rectangleorder=rectangleorder1)
20
21 argument1 = dict(clip=rectangle1, font="times-roman", fontsize=14)
22 Text_manipulator1 = PDFbf.manipulate_text(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"),
    font=argument1.get("font"), fontsize=argument1.get("fontsize"))
23 Text_manipulator1.text = Text_manipulator1.getvertext()
24 doc = Text_manipulator1.addtext()
25
26 missing_information1 = '''file-level:234_Manipdfgulate_text.pdf, page-level: 12, order-level: 6, base-level:
    missing'''
27 recorder2 = PDFbf.recorder(missing_information=missing_information1)
28 recorder2.recording()
29
30 updatedfilepath = filepath[:-4] + "_updated.pdf"
31 doc.save(updatedfilepath)
```

D.5 Adding Table

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks. Now, the drafter is trying to insert a table on the target area boxed by rectangle markups. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the content of a table) from high to low. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information, but identify tasks lacking necessary information, record the missing details, and execute the task that all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Crate an annotation extractor and extract the rectangles on the target page and order.
4. Modify the task instructions into dictionaries. Each dictionary contains the arguments for a table-adding mission.
5. Argument collection: Prepare parameters for adding a table by modifying the argument's value specified by the drafter's instructions into a dictionary.
6. Table adding: Create a table manipulator instance and add the table.
7. Save the file you have opened as a new file with an updated name.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
 "rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the text manipulator instance. Here is an example dictionary including all possible attributes:

argument = {"clip": a rectangle refers to the target position; "data": a list contains the content of each row, general form: [[row1 content],[row2 content], ...,[ith row content]]; "arrange": a list of row height and column width, general form:[[row height],[column width]], if only row height is defined, it is noted as [[row height], None], and vice versa as [None,[column width]]; "font": a string refers to the font of the text; "fontsize": an int refers to the size of text }

The drafter will selectively specify parameters such as font, font size etc. as needed, so please create a dictionary for each task that matches the description.

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:
 "import Fitz
 import PDFbf"

File open:

You can open the file by calling the following function:
 "doc = fitz.open("filepath")" Open the file with the file path.

Save the updated file:

You can save the file with an updated name by calling the following function:
 "updated_file = filepath[:-4] + "_updated.pdf" # update file path
 doc.save(updated_file)
 "

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):
 "annoextractor = PDFbf.extractanno(doc=fitz.document)"
 Get a rectangle and its reference points:
 "rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"

Table adding:

A table adder is instantiated by specifying the essential arguments (document(doc), page number(pagenumber), clip, data) and mentioned optional arguments (arrange, font, fontsize) on demand like this:
 "Table_adder = PDFbf.manipulate_table(doc=fitz.document, pagenumber=int, clip=rectangle, data=list, arrange=list, font="string, fontsize=int)"
 You can get a doc with a table added by calling the following function:
 "doc = Table_adder.addtable()"

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
 "recorder = PDFbf.recorder(missing_information=str)"
 Where "missing_information" is a string indicating the missing information, the standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: the relevant part of the original instruction, missing if there is no such part, fill in the relevant part of the original instruction and mark it as 'incomplete' in brackets if there is a relevant part but not enough necessary information".
 The incomplete instruction can be recorded by calling the following function:
 "recorder.recording()"

Here is an example:

User: The file name is "af35dsdd_table.pdf". Add tables in the first box on the second page. The first table: table content is: first row: "A, B, C", second row: "1, 2, 3", third row: "X, Y, Z". Font size 12. Row height: [30, 18, 18].The second table: table content is: first row: "Name, Age, Gender", second row: "Alice, 30, F", third row: "Bob, 25, M". Row height: [20, 15, 15]. Column width 50.
 Add tables in the second box on page 4. The first table content is: first row: "Item, Price", second row: "Apple, 1.0", third row: "Banana, 0.5". Font size 14. Column width: [20, 20]. The second table content is: first row: "Date, Time", second row: "2023-01-01, 12:00", third row: "2023-01-02, 13:00". Row height 20, column width: 60. The third table content is:

Response:

```

1 import fitz
2 import PDFbf
3
4 filepath = "af35dsdd_table.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8
9 pagenumber0 = 1
10
11 rectangleorder0 = 0
12 rectangle0, rfpoin0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
13
14 argument0 = dict(clip=rectangle0, data=[["A", "B", "C"], ["1", "2", "3"], ["X", "Y", "Z"]], arrange=[[30, 18, 18], None], fontsize=12)
15 table_adder0 = PDFbf.manipulate_table(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
16 data=argument0.get("data"), arrange=argument0.get("arrange"), fontsize=argument0.get("fontsize"))
16 doc = table_adder0.addtable()

```

```

18 argument1 = dict(clip=rectangle0, data=[["Name", "Age", "Gender"], ["Alice", "30", "F"], ["Bob", "25", "M"]],
19     arrange=[[20, 15, 15], [50, 50, 50]])
20 table_adder1 = PDFbf.manipulate_table(doc=doc,pagenumber=pagenumber0, clip=argument1.get("clip"),
21     data=argument1.get("data"),arrange=argument1.get("arrange"))
20 doc = table_adder1.addtable()
21
22 pagenumber1 = 3
23 rectangleorder1 = 1
24 rectangle1, rfpoint1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1,rectangleorder=rectangleorder1)
25
26 argument2 = dict(clip=rectangle1, data=[[{"Item", "Price"}, {"Apple", "1.0"}, {"Banana", "0.5"}], arrange=[None,
27     [20, 20]], fontsize=14)
27 table_adder2 = PDFbf.manipulate_table(doc=doc,pagenumber=pagenumber1, clip=argument2.get("clip"),
28     data=argument2.get("data"),arrange=argument2.get("arrange"), fontsize=argument2.get("fontsize"))
28 doc = table_adder2.addtable()
29
30 argument3 = dict(clip=rectangle1, data=[[{"Date", "Time"}, ["2023-01-01", "12:00"], ["2023-01-02", "13:00"]], arrange=[[20, 20, 20], [60, 60]])
31 table_adder3 = PDFbf.manipulate_table(doc=doc,pagenumber=pagenumber1, clip=argument3.get("clip"),
32     data=argument3.get("data"),arrange=argument3.get("arrange"))
32 doc = table_adder3.addtable()
33
34 missing_information1 = '''file-level:af35dsdd_table.pdf, page-level: 4, order-level: 2, base-level: missing'''
35
36 recorder1 = PDFbf.recorder(missing_information=missing_information1)
37 recorder1.recording()
38
39 updatedfilepath = filepath[:-4] + "_updated.pdf"
40 doc.save(updatedfilepath)

```

D.6 Revising Table

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks.

Now, the drafter is trying to edit the table content on the target area boxed by rectangle markups. Editing a table on a target position is seen as a task. There are three possible operations that must be selected from: deleting, clearing, and replacing. There are also format attributes that can be optionally specified on demand: the font, font size, alignment, and border width. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level (providing the necessary information of the operation) from high to low. The first three levels of information determine the target position, while the base level describes the operation. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the four-level information corresponding to each task from an instruction that may contain multiple tasks. Don't assume its value for necessary information if the command doesn't mention it or describe it vaguely. For all the tasks in commands, execute the tasks where all essential information is completely specified, while for tasks lacking necessary information, record the missing details. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information (file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for updating text content by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Extract original table: Create a table extractor instance and extract the original table.
6. Update table: Create a table manipulator instance, revise the table content, and add the new table.
7. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;

"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the table editing task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle boxes the table that needed to be revised; "font": a string refers to the font of the text in table; "fontsize": an int refers to the size of text in table; "borderwidth": an int refers to the width of the table border; "align": a string refers to the align of the text in the table, which can be one of "center", "left", "right"}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```

"import fitz
import PDFbf"

```

File open:

You can open the file by calling the following function:
"doc = fitz.open("filepath")" Open the file with the file path.

Save the updated file:

You can save the file with an updated name by calling the following function:
"

```
updated_file = filepath[:-4] + "_updated.pdf" # update file path  
doc.save(updated_file)  
"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
Get a rectangle and its reference points:
"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"

Extract the original table:

A table extractor can be instantiated by specifying the document(doc), page number(pagenumber), clip like this:
"table_extractor = PDFbf.extract_table(doc=fitz.document, pagenumber=int, clip=rectangle)"
You can get the date of the table by calling the following function:
"table_data = table_extractor.data"

Update table:

A table manipulator can be instantiated by specifying the essential arguments (document(doc), page number(pagenumber), clip, data) and optional arguments (font, fontsize, borderwidth, align) on demand like this:
"table_manipulator=PDFbf.manipulate_table(doc=fitz.document, pagenumber=int, clip=rectangle, data=list, font='string', fontsize=int, borderwidth=int, align='string')"

Here are some arguments that should be specified when you want to delete specified contents from the original tables:
"delcolumn": a list of columns that need to be deleted, [1] for the first column;
"delrow": similar to "delcolumn", a list of rows that need to be deleted, [5] for the fifth row;

You can get the revised table data by deleting the whole table or specified row or column from the original table data by specifying rows to delete(delrow), columns to delete(delcolumn) and calling the following function:
"table_manipulator.data = table_manipulator.cuttable(delrow=list,delcolumn=list)"
"table_manipulator.data = table_manipulator.cuttable()" delete the whole table if the delete table action is required with no target rows and columns specified.
"table_manipulator.data = table_manipulator.cuttable(delrow=[3,4,5,6])" delete the 3rd to 6th rows from the original table.
"table_manipulator.data = table_manipulator.cuttable(delrow=[3],delcolumn=[2,4])" Delete the 3rd row and delete the 2nd and 4th columns from the original table.

Here are some arguments that should be specified when you want to clear or replace specified contents from the original tables:

```
"startcell": a list refers to the position of the start cell to be cleared or replaced in the data list, general form: [row number, column number];  
"endcell": a list refers to the position of the end cell to be cleared or replaced in the data list, general form: [row number, column number];  
"repdata": a list of data to replace the specified content from the original table, general form: [[row1 content],[row2 content],...,[ith row content]];
```

You can get the revised table data by clearing the whole table or the data of specified cells of the original table by defining the start cell(startcell), end cell(endcell) and calling the following function:
"table_manipulator.data = table_manipulator.emptytable(startcell=list,endcell=list)"
"table_manipulator.data = table_manipulator.emptytable()" clear the whole table if the clear table action is required with no start cell(startcell) and end cell(endcell) specified.
"table_manipulator.data = table_manipulator.emptytable(startcell=[2,3],endcell=[5,7])" clear the content of cells starting from the 2nd row and 3rd column, end at the 5th row and 7th column from the original table.
"table_manipulator.data = table_manipulator.emptytable(startcell=[“all”,3])" clear the column 3.
"table_manipulator.data = table_manipulator.emptytable(startcell=[7,”all”])" clear the row 7.
"table_manipulator.data = table_manipulator.emptytable(startcell=[8,9])" clear the data of one cell at the 8th row and 9th column from the original table if clear table action is required with only one cell specified.

You can get the revised table data by replacing the whole table or the data of specified cells from the original table by defining the data used to replace(repdata), start cell(startcell), end cell(endcell) and calling the following function:
"table_manipulator.data = table_manipulator.modifytable(repdata=replace_data, startcell=list, endcell=list)"
"table_manipulator.data = table_manipulator.modifytable(repdata=replace_data)" Replace the whole table with the replace data if no start cell(startcell) and end cell(endcell) are specified.
"table_manipulator.data = table_manipulator.modifytable(repdata=replace_data, startcell=[7,9], endcell=[12,13])" Replace the data of cells starting from 7th row and 9th column, end at the 12th row and 13th column from the original table with the replace data.
"table_manipulator.data = table_manipulator.emptytable(repdata=replace_data, startcell=[“all”,3])" replace the column 3.
"table_manipulator.data = table_manipulator.emptytable(repdata=replace_data, startcell=[7,”all”])" replace the row 7.
"table_manipulator.data = table_manipulator.modifytable(repdata=replace_data, startcell=[2,3])" Replace the data of the cell at the 2nd row and 3rd column from the original table with the replace data if replace table action is required with only one cell specified.

You can get a doc with a revised table added by calling the following function:

```
"doc = table_manipulator.addtable()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
"recorder = PDFbf.recorder(missing_information=str)"
Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in

the original instruction following with an "incomplete" in brackets (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:
 "recorder.recording()"

Here is an example. You should respond in the same way.

User: The file name is "De222lsfqweete_table.pdf".

For the table in the first box of page 4, clear the cells from the second row and third column to the fourth row and fifth column. Replace the cells from the third row and first column to the fourth row and second column with the data "first row: A, B; second row: C, D". Then delete the first row and the second to fifth columns.

For the table on page 7, delete the second column. Clear the table.

For the seventh box on page eight, replace the table the data "first row: 10,11, second row: 25,26, third row: 31,32", clear the cell at 3rd row and 4th column, delete the 1-7th row and 10th columns.

Response:

```

1 import fitz
2 import PDFbf
3
4 filepath = "De222lsfqweete_table.pdf"
5 doc = fitz.open(filepath)
6 annoextractor = PDFbf.extractanno(doc=doc)
7
8pagenumber0 = 3
9rectangleorder0 = 0
10rectangle0, rfpoin0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
11
12argument0 = dict(clip=rectangle0)
13table_extractor0 = PDFbf.extract_table(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"))
14table_data0 = table_extractor0.data
15table_manipulator0 = PDFbf.manipulate_table(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
    data=table_data0)
16table_manipulator0.data = table_manipulator0.emptytable(startcell=[2, 3], endcell=[4, 5])
17repdata0 = [["A", "B"], ["C", "D"]]
18table_manipulator0.data = table_manipulator0.modifytable(repdata=repdata0, startcell=[3, 1], endcell=[4, 2])
19table_manipulator0.data = table_manipulator0.cuttable(delrow=[1], delcolumn=[2,3,4,5])
20doc = table_manipulator0.addtable()
21
22missing_information1='''file-level: De222lsfqweete_table.pdf, page-level:7, order-level: missing, base-level:
  delete the second column. Clear the table.'''
23recorder1 = PDFbf.recorder(missing_information=missing_information1)
24recorder1.recording()
25
26pagenumber2=7
27
28rectangleorder2=6
29rectangle2, rfpoin2 = annoextractor.getclip_rfpoin(pagenumber=pagenumber2, rectangleorder=rectangleorder2)
30argument2 = dict(clip=rectangle2)
31table_extractor2 = PDFbf.extract_table(doc=doc, pagenumber=pagenumber2, clip=argument2.get("clip"))
32table_data2 = table_extractor2.data
33table_manipulator2 = PDFbf.manipulate_table(doc=doc, pagenumber=pagenumber2, clip=argument2.get("clip"),
    data=table_data2)
34repdata2 = [[10,11],[25,26],[31,32]]
35table_manipulator2.data=table_manipulator2.modifytable(repdata=repdata2)
36table_manipulator2.data = table_manipulator2.emptytable(startcell=[3,4])
37table_manipulator2.data = table_manipulator2.cuttable(delrow=[1,2,3,4,5,6,7], delcolumn=[10])
38doc = table_manipulator2.addtable()
39
40updatedfilepath = filepath[:-4] + "_updated.pdf"
41doc.save(updatedfilepath)
```

D.7 Mapping Table

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks.

Now, the drafter wants to change the position of a table on the target area boxed by rectangle markups. There are three operations that the drafter may want: translation, rotation, and scaling. The details defining the operation include operation type, direction, and magnitude. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the details for at least one of the three operations) from high to low. What needs to be made more clear is that the details defining each type of operation that can be considered as necessary information are: pairs of information consisting of the type of operation and the direction, translation-direction, rotation-direction, and scale-bigger or smaller, respectively. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information (only the magnitude of operation can be assumed), but identify tasks lacking necessary information, record the missing details, and execute the task that all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for changing a table position by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Changing position: Create a table projector instance and change position.
6. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the table position-changing task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle boxes the text that need to be dealt with; "move": a list refers to translation details, general form: [move in its right direction (positive value if move right, negative value if move left), move in its upper direction (positive value if move up, negative value if move down), reference_point], "rotation": a list refers to the rotation details, general form: ['r', rotation degrees in the clockwise direction (positive value in clockwise direction, negative value in counterclockwise direction),reference_point], if the direction of rotation is not specified, clockwise is considered; "scal": a list refers to the scaling, general form: ['sc', [zoom factor X direction, zoom factor Y direction], reference_point]}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz  
import PDFbf"
```

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```
"  
updated_file = filepath[:-4] + "_updated.pdf" # update file path  
doc.save(updated_file)  
"
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

```
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
```

Get a rectangle and its reference points:

```
"rectangle, rpoint = annoextractor.getclip_rpoint(pagenumber=int,rectangleorder=int)"
```

Changing position:

A projector is instantiated by specifying the essential arguments (document(doc), page number(pagenumber), clip) and optional arguments (movement of translation (move), rotation(rotation) and scaling(scal)) like this:

```
"table_projector=PDFbf.Projector(doc=fitz.document,pagenumber=int,clip=rectangle,move=[move_in_right_direction, move_in_up_direction,  
reference_point],rotation=['r', rotation degrees in clockwise direction,reference_point],scal=['sc',[zoom factor in X direction, zoom factor in Y  
direction], reference_point])"
```

You can get a doc with a table mapped by calling the following functions:

```
"doc = table_projector.project()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the missing information, the standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: the relevant part of the original instruction, missing if there is no such part, fill in the relevant part of the original instruction following an 'incomplete' in brackets if there is a relevant part but not enough necessary information".

The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording()"
```

Here is an example. You should respond in the same way.

User: The file name is "47_mapqweqping_table.pdf". Move the table in the first box on page 5 20 units left,30 units up. Rotate another table 45 degrees clockwise. Move the table in the second box of the tenth page 10 units left, 20 units down.

Response:

```
1 import fitz  
2 import PDFbf  
3  
4 filepath = "47_mapqweqping_table.pdf"  
5 doc = fitz.open(filepath)  
6  
7 annoextractor = PDFbf.extractanno(doc=doc)
```

```

9 pagenumber0 = 4
10 rectangleorder0 = 0
11 rectangle0, rfpoint0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
12
13 argument0 = dict(clip=rectangle0, move=[-20, 30, rfpoint0])
14 table_project0 = PDFbf.Projector(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
15 move=argument0.get("move"))
15 doc = table_project0.project()
16
17 missing_information1 = '''file-level:47_mapqweqping_table.pdf, page-level: missing, order-level: missing, base-
18 level: rotate 45 degrees clockwise'''
19 recorder1 = PDFbf.recorder(missing_information=missing_information1)
20 recorder1.recording()
21
22 pagenumber1 = 9
23 rectangleorder1 = 1
24 rectangle1, rfpoint1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1, rectangleorder=rectangleorder1)
25
26 argument2 = dict(clip=rectangle1, move=[-10, -20, rfpoint1])
27 table_project2 = PDFbf.Projector(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"),
28 move=argument1.get("move"))
28 doc = table_project2.project()
29
30 updatedfilepath = filepath[:-4] + "_updated.pdf"
31 doc.save(updatedfilepath)

```

D.8 Updating Table Format

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks.

Now, the drafter wants to update the table format on the target area boxed by rectangle markups. The possible format attributes are: arrange, font, font size, alignment, border width. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing at least one of the format attributes) from high to low. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks. Don't assume the value of missing necessary information (only the value of mentioned format attributes can be assumed), but identify tasks lacking necessary information, record the missing details, and execute the task that all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0.Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.

1.Import libraries

2.Open the file.

3.Target position extraction: Crate an annotation extractor and extract the rectangles on the target page and order.

4.Argument collection: Prepare parameters for updating text format by modifying the argument's value specified by the drafter's instructions into a dictionary.

5.Table extraction: Create a table extractor instance and extract the original table..

6.Table update: Create a table manipulator and update the original table format.

7.Save the file.

Available tools:

Two constant defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;

"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the table format update task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle refers to the region of text should be dealing with. "arrange": a list of row height and column width, general form:[[row height], [column width]], if only row height is defined, it is noted as [[row height], None], and vice versa as [None,[column width]]; "font": a string refers to the font of the text; "fontsize": an int refers to the size of text, "borderwidth": an int refers to the width of the table border, "align": a string refers to the align of the data in the table, which can be one of "center", "left", "right", "justify";}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

"import fitz

import PDFbf"

File open:

You can open the file by calling the following function:

"doc = fitz.open("filepath")" Open the file with the file path.

Save the updated file:
 You can save the file with an updated name by calling the following function:

```

"updated_file = filepath[:-4] + "_updated.pdf" # update file path
doc.save(updated_file)
"

```

Target position extraction:
 An annotation extractor is instantiated by providing the essential arguments document(doc):
`"annoextractor = PDFbf.extractanno(doc=fitz.document)"`
 Get a rectangle and its reference points:
`"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"`

Table extraction:
 A table extractor can be instantiated by defining the document(doc), page number(pagenumber), clip like this:
`"table_extractor = PDFbf.extract_table(doc=fitz.document, pagenumber=int, clip=rectangle)"`
 You can get a list of each row date of the table by calling the following function:
`"table_data = table_extractor.data"`

Table update:
 A table manipulator can be instantiated by defining the necessary arguments (document(doc), page number(pagenumber), clip, data) and optionally mentioned arguments (arrange, font, fontsize, borderwidth, align) like this:
`"Table_manipulator = PDFbf.manipulate_table(doc=fitz.document, pagenumber=int, clip=rectangle, data=list, arrange=list, font=string, fontsize=int, borderwidth=int, align=string)"`
 You can get a doc with the table updated by calling the following functions:
`"doc = Table_manipulator.addtable()"`

Incomplete instruction recording:
 A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
`"recorder = PDFbf.recorder(missing_information=str)"`
 Where "missing_information" is a string indicating the highest-level of missing information, a standard format is ""file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no realted part) or related part in the original instruction following with an "incomplete" in brackets (if there is some related description but some necessary information is missing) "".
 The incomplete instruction can be recorded by calling the following function:
`"recorder.recording()"`

Here is an example. You should respond in the same way to execute the user's instructions.
 User: The file name is "2anipusdflate_table.pdf". For the table in the third box on page two, change the align to the right. For the table on page one, change the font to "Times-Roman".

Response:

```

1 import fitz
2 import PDFbf
3
4 filepath = "2anipusdflate_table.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8 pagenumber0=1
9 rectangleorder0=2
10 rectangle0, rfpoin0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0,rectangleorder=rectangleorder0)
11
12 argument0 = dict(clip=rectangle0,align="right")
13 table_extractor0 = PDFbf.extract_table(doc=doc,pagenumber=pagenumber0,clip=argument0.get("clip"))
14 table_data0 = table_extractor0.data
15 Table_manipulator0 = PDFbf.manipulate_table(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
16 data=table_data0, align=argument0.get("align"))
16 doc = Table_manipulator0.addtable()
17
18 missing_information1 = '''file-level:2anipusdflate_table.pdf, page-level: 1, order-level: missing, base-level:
19 change the font to "Times-Roman"'''
20 recorder1 = PDFbf.recorder(missing_information=missing_information1)
20 recorder1.recording()
21
22 updatedfilepath = filepath[:-4] + "_updated.pdf"
23 doc.save(updatedfilepath)

```

D.9 Adding Vectors

Task background:
 You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks.
 Now, the drafter is trying to add the vector graphics drawn manually as standard drawings. The vector graphics have a parent class called "drawings" and three son classes, which are "rebar," "pillar/column," "line". Adding a class of vector graphics is seen as a task. The user will specify the target class he wants to deal with. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level(providing the annotation color) and base level(providing the vector class and format of new vectors) from high

to low. A lower level information can correspond to only one higher level information, while higher level information can correspond to multiple lower-level information. What needs to be made clearer is that the details defining a parent class that can be considered necessary information are: the color of annotation it is. You need to find the four level information corresponding to each task from an instruction that may contain multiple tasks. Don't assume its value for necessary information if the command doesn't mention it or describe it vaguely. For all the tasks in commands, execute the tasks where all essential information is completely specified, while for tasks lacking necessary information, record the missing details. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.

1. Import libraries
2. Open the file.

3. Argument collection: Modify the value of the argument specified by user instructions into dictionaries. Each dictionary contains the arguments for each stroke-adding action.

4. Get the target class of vector graphics

4.1Parent Class Extraction: If a parent class("drawings") is asked to be added, you need to extract the parent class ("drawings") by creating a parent class extractor and then extract the parent class ("drawings").

4.2Select Son Class: If a son class ("rebar", "pillar/column", "line", "rebars and columns") is asked to be added, you need to extract the parent class ("drawings") by creating a parent class extractor and then extract the parent class ("drawings") with the same "pagenumber" and "annocolor" as a prerequisite class. And then, create a son class selector, and select the target son classes ("rebar", "pillar/column", "line", "rebars and columns") from the prerequisite class.

5. Strokes adding: Create a stroke manipulator with the target class of vector graphics and add them as standard strokes with the stroke manipulator for each action.

6. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the vector graphics adding a task. Here is an example dictionary including all possible attributes:

{"annocolor": a string indicating the color of the annotation needed to be extracted; "drwcolor": a string indicating the color of the strokes when redrawing them as standard strokes; "width": an int indicating the line width of strokes when redrawing them as standard strokes}

The target class of vector graphics should named in this format: specified color + page number + class name. For example, "C_4_rebars" means the rebar in color C on page 5. "A_3_columns" means the columns in color A on page 4. "T_0_lines" means the lines in color T on page 0.

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz
import PDFbf"
```

You have to import the library before you use the following tools provided.

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```
""
updated_file = filepath[:-4] + "_updated.pdf" # update file path
doc.save(updated_file)
""
```

Parent Class Extraction:

A parent class extractor can be instantiated by specifying the document(doc), page number(pagenumber), and annotation color(annocolor) like this:

```
"anno_extractor = PDFbf.extractanno (doc=fitz.document, pagenumber=int, annocolor=string)"
```

You can extract a list of the parent class ("drawings") vector graphics in the annotation by calling the following function:

```
"color_pagenumber_drawings = anno_extractor.anno()";
```

Select Son Class:

A son classes selector can be instantiated by specifying the document(doc), page number(pagenumber), and the parent class with the same specified color and pagename of target son class(cdrawing) to extract a son classes ("rebars", "columns", "lines") with specified color and pagename, like this:

```
"color_pagenumber_son_classes_selector=PDFbf.select_from_drawings(doc=fitz.document, pagenumber=int,
cdrawing=color_pagenumber_drawing)"
```

You can get a son class ("rebars", "columns", "lines") with a specified color on a specified page from the parent class ("drawings") with the same specified color and page by calling the following functions:

```
"color_pagenumber_rebars=color_pagenumber_son_classes_selector.mode2_rebars_Cross_Touch_Intersect()", get a "rebars" son class with specified color on a specified page;
```

```
"color_pagenumber_columns=color_pagenumber_son_classes_selector.mode2_columns_Cross_Touch_Intersect()", get a "columns" son class with specified color on a specified page;
```

```
"color_pagenumber_lines=color_pagenumber_son_classes_selector.mode2_lines_Cross_Touch_Intersect()", get a "lines" son class with specified color on the specified page.
```

"color_pagenumber_rebar_and_column=color_pagenumber_son_classes_selector.get_rebar_column()", get a "rebars and columns" son class with specified color on the specified page.

Strokes adding:

A stroke manipulator can be instantiated by specifying the document(doc), page number(pagenumber), a list of the target class of vector graphics(sel_drawings) and optional arguments(stroke colors(drwcolor), line width of stroke(width)) like this:

"drawings_manipulator=PDFbf.manipulate_draw(doc=fitz.document, pagenumber=int, sel_drawings=list, drwcolor=string, width=int)"
You can get a doc with target vector graphics added by calling the following function:
"doc=drawings_manipulator.add_standrawing()"

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
"recorder = PDFbf.recorder(missing_information=str)"

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no related part) or related part in the original instruction following with an "incompleted" in a brace (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:
"recorder.recording()"

Here is an example of executing the user's instructions. You should respond in the same way.

User: The file name is "sdh.pdf". On the first page, add rebars marked by the red annotation, line width 3; Add drawings marked by blue annotation. The line color is green, line width is 2. On the third page, add columns. The line width is 4.

Response:

```

1 import fitz
2 import PDFbf
3
4 filepath = "sdh.pdf"
5 doc = fitz.open(filepath)
6
7 pagenumber0 = 0
8 argument0 = dict(annocolor="red", width=3)
9 annoextractor0 = PDFbf.extractanno(doc=doc, pagenumber=pagenumber0, annocolor=argument0.get("annocolor"))
10 red_0_drawings = annoextractor0.anno()
11 red_0_son_classes_selector=PDFbf.select_from_drawings(doc=doc, pagenumber=pagenumber0, cdrawings=red_0_drawings)
12 red_0_rebars=red_0_son_classes_selector.mode2_rebars_Cross_Touch_Intersect()
13 drawings_manipulator0 = PDFbf.manipulate_draw(doc=doc, pagenumber=pagenumber0, sel_drawings=red_0_rebars,
width=argument0.get("width"))
14 doc = drawings_manipulator0.add_standrawing()
15
16 argument1 = dict(annocolor="blue", drwcolor="green", width=2)
17 annoextractor1 = PDFbf.extractanno(doc=doc, pagenumber=pagenumber0, annocolor=argument1.get("annocolor"))
18 blue_0_drawings = annoextractor1.anno()
19 drawings_manipulator1 = PDFbf.manipulate_draw(doc=doc, pagenumber=pagenumber0, sel_drawings=blue_0_drawings,
drwcolor=argument1.get("drwcolor"), width=argument1.get("width"))
20 doc = drawings_manipulator1.add_standrawing()
21
22 missing_information1='''file-level: sdh.pdf, page-level:3, order-level: missing, base-level: add columns. Line
width is 4'''
23 recorder1 = PDFbf.recorder(missing_information=missing_information1)
24 recorder1.recording()
25
26 updatedfilepath = filepath[:-4] + "_updated.pdf"
27 doc.save(updatedfilepath)
```

D.10 Deleting Vectors

Task background:

You're a worker using tools provided by the drafter to implement his instruction on technical drawing editing tasks. Now, the drafter is trying to delete the vector graphics on the target area boxed by rectangle markups. The vector graphics have a parent class called "drawings" and three son classes, which are "rebar," "column," "line". There are two selection modes to select target class vectors, which are: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are covered within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox. Deleting vectors on a target position is seen as a task. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(the class(es) to be deleted) from high to low. A lower-level information can correspond to only one higher-level information, while higher-level information can correspond to multiple lower-level information. What needs to be made more clear is that the details defining a target class that can be considered necessary information are the class and the selection mode. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information, but identify tasks lacking necessary information, record the missing details, and execute the task only once all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for deleting vectors on a target position by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Target class(es) selection: Create a drawing selector and select the specified vector graphics(drawings, lines, rebars, columns) with the specified selection mode by the selector.
6. Delete vectors: Create a drawing manipulator or cleaner instance and delete the target vectors.
7. Repair: Create a repairer instance if the repair action is asked for a target position, and repair vectors that should not be affected by the cleaner or manipulator; otherwise, skip this step.
8. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the vector deleting task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle boxes the drawings that needed to be dealt with}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz  
import PDFbf"
```

You have to import the library before using the following tools provided.

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

"annoextractor = PDFbf.extractanno(doc=fitz.document)"

Get a rectangle and its reference points:

"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"

Target class(es) selection:

A drawing selector can be instantiated by specifying the document(doc), page number(pagenumber), and clip like this:

"selector = PDFbf.selector(doc=fitz.document, pagenumber=int, clip=rectangle)"

The selector is to select a class with two optional selection modes. The class can be described by the type of vector graphics ("drawings", "lines", "rebars", or "columns"). There are two selection modes: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are covered within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox. If the selection mode is not mentioned, take selection mode 2 as the default.

Choose the selection specified or closest to the description of the subgroup.

You can get a list of a subgroup with a specified selection mode by calling these functions:

```
"covered_drawings = selector.mode1_drawings_Window_Cover_Enclosure()" select drawings with selection mode 1;  
"intersected_drawings = selector.mode2_drawings_Cross_Touch_Intersect()" select drawings with selection mode 2;  
"covered_lines = selector.mode1_lines_Window_Cover_Enclosure()" select lines with selection mode 1;  
"intersected_lines = selector.mode2_lines_Cross_Touch_Intersect()" select lines with selection mode 2;  
"covered_rebars = selector.mode1_rebars_Window_Cover_Enclosure()" select rebars with selection mode 1;  
"intersected_rebars = selector.mode2_rebars_Cross_Touch_Intersect()" select rebars with selection mode 2;  
"covered_columns = selector.mode1_columns_Window_Cover_Enclosure()" select columns with selection mode 1;  
"intersected_columns = selector.mode2_columns_Cross_Touch_Intersect()" select columns with selection mode 2;
```

Delete vectors:

1. Delete selected class:

A drawing manipulator is instantiated by specifying the document(doc), page number(pagenumber), and list of drawings(listofcdraw) like this:

"drawing_manipulator = PDFbf.draw_drawer(doc= fitz.document, pagenumber=int, listofcdraw=list)"

You can get a doc with the list of drawings deleted by calling the following functions:

"doc = drawing_manipulator.delete_draw()"

2. Delete indiscriminately:

An indiscriminate cleaner is instantiated by specifying the document(doc), page number(pagenumber), and clip like this:

"cleaner=PDFbf.delete(doc=fitz.document, pagenumber=int, clip=rectangle)"

You can get a doc with everything in the clip covered by calling the following function:

"doc = cleaner.applydelete()"

Repair:

A repairer is instantiated by specifying the document(doc), page number(pagenumber), clip, cdrawings(cdrawings) and optional specify selected drawing(drawings) like this:

"repairer=PDFbf.repairer(doc= fitz.document,pagenumber=int,clip=rectangle, cdrawings=list, sel_drawings=list)"

"cdrawings" can be obtained by calling the selector like this: "selector.selected_lines", "sel_drawings" can be specified as None to repair the effect by an indiscriminate cleaner.

You can get a lines repaired doc by calling this function:

"doc = repairer.del_repair()"

Selected drawings should be the total selected drawings.

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
"recorder = PDFbf.recorder(missing_information=str)"
Where "missing_information" is a string indicating the highest-level of missing information, a standard format is "file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no realted part) or related part in the original instruction following with an "incompleted" in a brace (if there is some related description but some necessary information is missing)".
The incomplete instruction can be recorded by calling the following function:
"recorder.recording()"

Here is an example. You should respond in the same way.

User: The file name is "sdh.pdf". For the first box on page three, delete the drawings covered by the box, delete the rebars intersected with the box, and delete the columns covered by the box. For the fourth box on page eleven, delete the lines intersecting with the box. For the eighth box on page one, delete the lines covered by the box and delete the rebars covered by the box. For the second box on page 18, delete. Repair all the drawings.

Response:

```
1 import fitz
2 import PDFbf
3
4 filepath = "sdh.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8
9 pagenumber0 = 2
10 rectangleorder0 = 0
11 rectangle0, rfpoint0 = annoextractor.getclip_rfpoin
12 argument0 = dict(clip=rectangle0)
13 selector0 = PDFbf.selector(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"))
14 covered_drawing
15 intersected_rebars0 = selector0.mode2_rebars_Cross_Touch_Intersect()
16 covered_columns0 = selector0.mode1_columns_Window_Cover_Enclosure()
17 sel_drawing
18 drawing_manipulator0 = PDFbf.draw_drawer(doc=doc, pagenumber=pagenumber0, listofcdraw=sel_drawing
19 doc = drawing_manipulator0.delete_drawing()
20 repairer0 = PDFbf.repairer(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
21     cdrawing
22 doc = repairer0.del_repair()
23
24 pagenumber1 = 10
25 rectangleorder1 = 3
26 rectangle1, rfpoint1 = annoextractor.getclip_rfpoin
27 argument1 = dict(clip=rectangle1)
28 selector1 = PDFbf.selector(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"))
29 intersected_lines1 = selector1.mode2_lines_Cross_Touch_Intersect()
30 drawing_manipulator1 = PDFbf.draw_drawer(doc=doc, pagenumber=pagenumber1, listofcdraw=intersected_lines1)
31 doc = drawing_manipulator1.delete_drawing()
32 repairer1 = PDFbf.repairer(doc=doc, pagenumber=pagenumber1, clip=argument1.get("clip"),
33     cdrawing
34 doc = repairer1.del_repair()
35
36 pagenumber2=0
37 rectangleorder2=7
38 rectangle2, rfpoint2 = annoextractor.getclip_rfpoin
39 argument2 = dict(clip=rectangle2)
40 selector2=PDFbf.selector(doc=doc,pagenumber=pagenumber2,clip=argument2.get("clip"))
41 covered_lines2=selector2.mode1_lines_Window_Cover_Enclosure()
42 covered_rebars2 = selector2.mode1_rebars_Window_Cover_Enclosure()
43 sel_drawing2 = covered_lines2 + covered_rebars2
44 drawing_manipulator2 = PDFbf.draw_drawer(doc=doc,pagenumber=pagenumber2,listofcdraw=sel_drawing2)
45 doc = drawing_manipulator2.delete_drawing()
46 repairer2 = PDFbf.repairer(doc=doc, pagenumber=pagenumber2, clip=argument2.get("clip"),
47     cdrawing
48 doc = repairer2.del_repair()
49
50 missing_information3='''file-level: sdh.pdf, page-level:18, order-level:2, base-level: delete(incomplete)'''
51 recorder3 = PDFbf.recorder(missing_information=missing_information3)
52 recorder3.recording()
53
54 updatedfilepath = filepath[:-4] + "_updated.pdf"
55 doc.save(updatedfilepath)
```

D.11 Mapping Vectors

Task background:

You're a high-level programmer. You are writing code to help a drafter implement construction drawings editing tasks.

Now, the drafter is trying to map the vector graphics on the target area boxed by rectangle markups. The vector graphics have a parent class called

"drawings" and three son classes, which are "rebar," "column," "line". There are two selection mode to select target class vectors, which are: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are covered within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox. There are three operations that the drafter may want: translation, rotation, and scaling. The details defining the operation include operation type, direction, and magnitude. Deleting a group of vectors on a target position with a corresponding group of operations is seen as a task. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the classes of vectors to be mapped and providing the details for at least one of the three operations correspondingly) from high to low. A lower-level information can correspond to only one higher-level information, while higher-level information can correspond to multiple lower-level information. What needs to be made more clear is thatthe details defining a target class that can be considered necessary information are the class and the selection mode. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information (only the magnitude of operation can be assumed), but identify tasks lacking necessary information, record the missing details, and execute the task only once all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.

1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for mapping vectors on a target position by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Target class(es) selection: Create a drawing selector and select the specified vector graphics(drawings, lines, rebars, columns) with the specified selection mode by the selector.
6. Mapping vectors: Create a vectors projector instance and map vectors.
7. Save the file.

Available tools:

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;

"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the table position-changing task. Here is an example dictionary including all possible attributes:

```
{"clip": a rectangle boxes the text that needs to be dealt with; "move": a list refers to translation details, general form: [move in its right direction (positive value if move right, negative value if move left), move in its upper direction (positive value if move up, negative value if move down), reference_point], "rotation": a list refers to the rotation details, general form: [r', rotation degrees in the clockwise direction (positive value in clockwise direction, negative value in counterclockwise direction),reference_point], if the direction of rotation is not specified, clockwise is considered; "scal": a list refers to the scaling, general form: [sc', [zoom factor X direction, zoom factor Y direction], reference_point]}
```

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```
"import fitz
import PDFbf"
```

You have to import the library before you use the following tools provided.

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```
""
updated_file = filepath[:-4] + "_updated.pdf" # update file path
doc.save(updated_file)
""
```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):

```
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
```

Get a rectangle and its reference points:

```
"rectangle, rfpoin = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"
```

Target class(es) selection:

A vector selector can be instantiated by specifying the document(doc), page number(pagenumber), and clip like this:

```
"selector = PDFbf.selector(doc=fitz.document, pagenumber=int, clip=rectangle)"
```

The selector is to select a subgroup from drawings with two optional selection modes. The subgroup can be described by the type of vector graphics ("drawings", "lines", "rebars", or "columns"). There are two selection modes: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are completely within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox.

Choose the selection specified or closest to the description of the subgroup.

You can get a list of a subgroup with a specified selection mode by calling these functions:

```
"covered_drawings = selector.mode1_drawings_Window_Cover_Enclosure()" select drawings with selection mode 1;
"intersected_drawings = selector.mode2_drawings_Cross_Touch_Intersect()" select drawings with selection mode 2;
"covered_lines = selector.mode1_lines_Window_Cover_Enclosure()" select lines with selection mode 1;
"intersected_lines = selector.mode2_lines_Cross_Touch_Intersect()" select lines with selection mode 2;
"covered_rebars= selector.mode1_rebars_Window_Cover_Enclosure()" select rebars with selection mode 1;
```

```

"intersected_rebars = selector.mode2_rebars_Cross_Touch_Intersect()" select rebars with selection mode 2;
"covered_columns = selector.mode1_columns_Window_Cover_Enclosure()" select columns with selection mode 1;
"intersected_columns = selector.mode2_columns_Cross_Touch_Intersect()" select columns with selection mode 2;

Mapping vectors:
A vector projector is instantiated by specifying the essential arguments (document(doc), page number(pagenumber), clip, selected drawings(sel_drawings), cdrawings(cdrawings)) and optional arguments (movement of translation (move), rotation(rotation) and scaling(scal)) on demand like this:
"projector=PDFbf.project_draw(doc=fitz.document,pagenumber=int,clip=rectangle,sel_drawings=list, cdrawings=list, move=[move_in_right_direction, move_in_up_direction, reference_point], rotation=[r, rotation degrees in clockwise direction, reference_point], scal=['sc', [zoom factor X direction, zoom factor Y direction], reference_point])"
You can get a doc with the drawings mapped by calling the following function:
"doc = projector.project()"
"cdrawings" can be obtained by calling the selector like this "selector.selected_lines",

```

Incomplete instruction recording:
A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(missing_information):
"recorder = PDFbf.recorder(missing_information=str)"
Where "missing_information" is a string indicating the highest-level of missing information, a standard format is ""file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no realted part) or related part in the original instruction following with an "incompleted" in a brace (if there is some related description but some necessary information is missing)".
The incomplete instruction can be recorded by calling the following function:
"recorder.recording()"

Here is an example. You should respond in the same way.

User: The file name is "fjhgd.pdf". For the eighth box on page one, rotate the drawing covered by the box 45 degrees clockwise; move the rebars intersected with the box 50 units left, 20 units down. For the second box on page five, scale the columns intersected with the box 0.8 times in both x and y directions; rotate the rebars covered by the box 45 degrees counterclockwise and move it 30 units left, 20 units up and scale it 0.8 times in x direction, 1.2 times in y direction. For the first box on page seven, scale the rebars covered by the box 1.5 times in both x and y directions; move the lines covered by the box 20 units right, 10 units up, and rotate it.

Response:

```

1 import fitz
2 import PDFbf
3
4 filepath = "fjhgd.pdf"
5 doc = fitz.open(filepath)
6
7 annoextractor = PDFbf.extractanno(doc=doc)
8
9 pagenumber0=0
10 rectangleorder0=7
11 rectangle0, rfpoint0 = annoextractor.getclip_rfpoinpt(pagenumber=pagenumber0, rectangleorder=rectangleorder0)
12 argument0=dict(clip=rectangle0, rotation=[r, 45, rfpoint0])
13 selector0 = PDFbf.selector(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"))
14 covered_drawings0 = selector0.mode1_drawing_Window_Cover_Enclosure()
15 projector0 = PDFbf.project_draw(doc=doc, pagenumber=pagenumber0, clip=argument0.get("clip"),
16 sel_drawings=covered_drawings0, cdrawings=selector0.selected_lines, move=argument0.get("rotation"))
17 doc = projector0.project()
18
19 argument1=dict(clip=rectangle0, move=[-50, -20, rfpoint0])
20 intersected_rebars1 = selector0.mode2_rebars_Cross_Touch_Intersect()
21 projector1 = PDFbf.project_draw(doc=doc, pagenumber=pagenumber0, clip=argument1.get("clip"),
22 sel_drawings=intersected_rebars1, cdrawings=selector0.selected_lines, rotation=argument1.get("move"))
23 doc = projector1.project()
24
25 pagenumber1 = 4
26 rectangleorder1 = 1
27 rectangle1, rfpoint1 = annoextractor.getclip_rfpoinpt(pagenumber=pagenumber1, rectangleorder=rectangleorder1)
28 argument2 = dict(clip=rectangle1, scal=[sc, [0.8, 0.8], rfpoint1])
29 selector1 = PDFbf.selector(doc=doc, pagenumber=pagenumber1, clip=argument2.get("clip"))
30 intersected_columns2 = selector1.mode2_columns_Cross_Touch_Intersect()
31 projector2 = PDFbf.project_draw(doc=doc, pagenumber=pagenumber1, clip=argument2.get("clip"),
32 sel_drawings=intersected_columns2, cdrawings=selector1.selected_lines, scal=argument2.get("scal"))
33 doc = projector2.project()
34
35 argument3 = dict(clip=rectangle1, move=[-30, 20, rfpoint1], rotation=[r, -45, rfpoint1], scal=[sc, [0.8,
36 1.2], rfpoint1])
37 covered_rebars3 = selector1.mode1_rebars_Window_Cover_Enclosure()
38 projector3 = PDFbf.project_draw(doc=doc, pagenumber=pagenumber1, clip=argument3.get("clip"),
39 sel_drawings=covered_rebars3, cdrawings=selector1.selected_lines, rotation=argument3.get("rotation"),
40 move=argument3.get("move"), scal=argument3.get("scal"))
41 doc = projector3.project()
42
43 pagenumber2 = 6
44 rectangleorder2 = 0
45 rectangle2, rfpoint2 = annoextractor.getclip_rfpoinpt(pagenumber=pagenumber2, rectangleorder=rectangleorder2)
46 argument4 = dict(clip=rectangle2, scal=[sc, [1.5, 1.5], rfpoint2])
47 selector2 = PDFbf.selector(doc=doc, pagenumber=pagenumber2, clip=argument4.get("clip"))
48 covered_rebars4 = selector2.mode1_rebars_Window_Cover_Enclosure()
49 projector4 = PDFbf.project_draw(doc=doc, pagenumber=pagenumber2, clip=argument4.get("clip"),
50 sel_drawings=covered_rebars4, cdrawings=selector2.selected_lines, scal=argument4.get("scal"))
51 doc = projector4.project()

```

```

46 missing_information5='''file-level: fjhgdf.pdf, page-level:7, order-level:1, base-level: move the lines covered
    by the box 20 units right, 10 units up, and rotate it (incomplete)'''
47 recorder5 = PDFbf.recorder(missing_information=missing_information5)
48 recorder5.recording()
49
50 updatedfilepath = filepath[:-4] + "_updated.pdf"
51 doc.save(updatedfilepath)

```

D.12 Updating Vector Format

Task background:

You're a high-level programmer. You are writing code to help a drafter implement construction drawings editing tasks.

Now, the drafter is trying to update the vector graphics format on the target area boxed by rectangle markups. The vector graphics have a parent class called "drawings" and three son classes, which are "rebar," "column," "line". There are two selection modes to select target class vectors, which are: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are covered within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox. There are all possible format attributes that the drafter may want to change: stroke color, line type (dash or straight), close path, fill color, joint type, cap type, and stroke width. Updating a group of vectors on a target position with a corresponding group of format attributes is seen as a task. The necessary information for such a task has four levels, which are file-level (providing the file path), page-level (providing page number), order-level (providing rectangle order on the page), and base-level(providing the classes of vectors to be updated and providing the details for at least one of the format attributes correspondingly) from high to low. A lower-level information can correspond to only one higher-level information, while higher-level information can correspond to multiple lower-level information. What needs to be clarified is that the details defining a target class that can be considered necessary information are the class and the selection mode. You need to find the necessary information corresponding to each task from an instruction that may contain multiple tasks to complete the execution of the instruction. Don't assume the value of missing necessary information (only the value of the format attribute mentioned but not specified value explicitly can be assumed), but identify tasks lacking necessary information, record the missing details, and execute the task only once all essential information is gathered. Don't repeat the operations related to the same higher-level information when dealing with lower-level operations. Avoid using loops.

Task plan:

The following is a standard process for completing a task:

0. Inexecutable task recording: If you think a level of necessary information(file-level, page-level, order-level, base-level) for a task is missing in the instruction, don't assume the value but skip all the operations of that task and create a recorder to record the incomplete instruction, then continue on next task. Otherwise, ignore this step if all four levels of necessary information of a task can be found.
1. Import libraries
2. Open the file.
3. Target position extraction: Create an annotation extractor and extract the rectangles on the target page and order.
4. Argument collection: Prepare parameters for updating vectors on a target position by modifying the argument's value specified by the drafter's instructions into a dictionary.
5. Target class(es) selection: Create a drawing selector and select the specified vector graphics(drawings, lines, rebars, columns) with the specified selection mode by the selector.
6. Updating vectors: Create a vector manipulating instance and update vectors.
7. Save the file.

Available tools:

Two constants defined by the user:

"pagenumber": an int refers to the page number, pagenumber=0 indicates page one;
"rectangleorder" an int refers to the order of rectangles within the page, rectangleorder=0 indicates the first rectangle;

Argument collection:

Extract the argument values from the instruction and store them as a dictionary for the vector updating task. Here is an example dictionary including all possible attributes:

{"clip": a rectangle boxes the drawings that needed to be dealt with; "fillcolor": a string refers to the fill color; "drwcolor": a string refers to the color of the drawing; "dashes": a sting refers to the dashed line, the general format is "[n m] 0", n is an int indicating the dash length, m (optional) is an int indicating the subsequent gap length, and 0 is an int specifying no pixels should be skipped before the dashing starts, if a dashed line is required, but no parameters are specified, take "[3 3]0"; if a straight line is required, the parameters should be "[0 0]0"; "closePath": a bool indicating whether the path is a close path; "lineJoin": an int controlling the way how line connections look like. This may be either as a sharp join (0), a rounded join (1), or a cut-off join(2); "lineCap": an int controlling the look of line ends. the options are sharp ends(0), semi-circle ends(1), and semi-square ends(2); "width": an int refers to the line width of drawings}

Popular PDF editing library "PyMuPDF" and a user-tailored library "PDFbf" are provided. You can import the two libraries by:

```

"import fitz
import PDFbf"
```

You have to import the library before you use the following tools provided.

File open:

You can open the file by calling the following function:

```
"doc = fitz.open("filepath")" Open the file with the file path.
```

Save the updated file:

You can save the file with an updated name by calling the following function:

```

"
updated_file = filepath[:-4] + "_updated.pdf" # update file path
doc.save(updated_file)
"

```

Target position extraction:

An annotation extractor is instantiated by providing the essential arguments document(doc):
"annoextractor = PDFbf.extractanno(doc=fitz.document)"
Get a rectangle and its reference points:
"rectangle, rfpoint = annoextractor.getclip_rfpoin(pagenumber=int,rectangleorder=int)"

Target class(es) selection:

A vector selector can be instantiated by specifying the document(doc), page number(pagenumber), and clip like this:
"selector = PDFbf.select_from_drawings(doc=fitz.document, pagenumber=int, clip=rectangle)"

The selector is to select a subgroup from drawings with two optional selection modes. The subgroup can be described by the type of vector graphics ("drawings", "lines", "rebars", or "columns"). There are two selection modes: 1. Window/Cover/Enclosure mode: the vector will be selected only if all vertices and boundaries of the graphic are completely within the rectangular checkbox, 2 Cross/Touch/Intersect mode: the vector will be selected if any of the vertices and a boundary of the vector are within the rectangular checkbox.

Choose the selection specified or closest to the description of the subgroup.

You can get a list of a subgroup with a specified selection mode by calling these functions:

```
"covered_drawings = selector.mode1_drawings_Window_Cover_Enclosure()" select drawings with selection mode 1;  
"intersected_drawings = selector.mode2_drawings_Cross_Touch_Intersect()" select drawings with selection mode 2;  
"covered_lines = selector.mode1_lines_Window_Cover_Enclosure()" select lines with selection mode 1;  
"intersected_lines = selector.mode2_lines_Cross_Touch_Intersect()" select lines with selection mode 2;  
"covered_rebars= selector.mode1_rebars_Window_Cover_Enclosure()" select rebars with selection mode 1;  
"intersected_rebars = selector.mode2_rebars_Cross_Touch_Intersect()" select rebars with selection mode 2;  
"covered_columns = selector.mode1_columns_Window_Cover_Enclosure()" select columns with selection mode 1;  
"intersected_columns = selector.mode2_columns_Cross_Touch_Intersect()" select columns with selection mode 2;
```

Updating vectors:

A vector manipulator is instantiated by specifying the essential arguments (document(doc), page number(pagenumber), selected drawings(sel_drawings)) and optional arguments (fill color (fillcolor), drawing color(drwcolor), dashes line (dashes), close path or not (closePath), line joint shape (lineJoin), line end shape (lineCap), line width (width)) on demand like this:
"manipulator=PDFbf.manipulate_draw(doc=doc,pagenumber=int,sel_drawings=list,fillcolor=string,drwcolor=string,dashes=string,closePath=bool,lineJoin=int,lineCap=int,width=int)"

You can get a doc with the list of drawings refreshed by calling the following function:

```
"doc = manipulator.update_draw()"
```

Incomplete instruction recording:

A missing key attribute recorder can be instantiated to record the incomplete instruction by providing the essential arguments(incomplete_instruction, missing_information):

```
"recorder = PDFbf.recorder(missing_information=str)"
```

Where "missing_information" is a string indicating the highest-level of missing information, a standard format is ""file-level: value or missing, page-level: value or missing, order-level: value or missing, base-level: related part in the original instruction or missing(if no realted part) or related part in the original instruction following with an "incompletd" in a brace (if there is some related description but some necessary information is missing)". The incomplete instruction can be recorded by calling the following function:

```
"recorder.recording()"
```

Here is an example. You should respond in the same way.

User: The file name is "fjhgd.pdf". For the third box on page two, change the columns intersecting with the box to red, dash lines with rounded line joints; change the line covered by the box to a dashed line and line width 3. For the eighth box on page one, change the lines intersecting with the box to yellow filling, close path, semi-square line end; change the rebars covered by the box to blue with a dashed line and line width 3. For the fifth box on page ten, change the columns covered by the box to green with a semi-square line end and line width 4; change the rebars intersecting with the box.

Response:

```
1 import fitz  
2 import PDFbf  
3 filepath = "fjhgd.pdf"  
4 doc = fitz.open(filepath)  
5  
6 annoextractor = PDFbf.extractanno(doc)  
7  
8 pagenumber0=1  
9 rectangleorder0=2  
10 rectangle0, rfpoint0 = annoextractor.getclip_rfpoin(pagenumber=pagenumber0, rectangleorder=rectangleorder0)  
11 argument0=dict(clip=rectangle0,drwcolor="red",dashes=[3,3]0,lineJoin=1)  
12 selector0 = PDFbf.selector(doc=doc,pagenumber=pagenumber0,clip=argument0.get("clip"))  
13 intersected_columns0 = selector0.mode2_columns_Cross_Touch_Intersect()  
14 manipulator0=PDFbf.manipulate_draw(doc=doc,pagenumber=pagenumber0,sel_drawings=intersected_columns0,drwcolor=argument0.get("drwcolor"),dashes=argument0.get("dashes"),lineJoin=argument0.get("lineJoin"))  
15 doc = manipulator0.update_draw()  
16  
17 argument1=dict(clip=rectangles0,dashes=[3,3]0,width=3)  
18 covered_lines1 = selector0.mode1_lines_Window_Cover_Enclosure()  
19 manipulator1=PDFbf.manipulate_draw(doc=doc,pagenumber=pagenumber0,sel_drawings=covered_lines1,dashes=argument1.get("dashes"),width=argument1.get("width"))  
20 doc = manipulator1.update_draw()  
21  
22 pagenumber1=0  
23 rectangleorder1=7  
24 rectangle1, rfpoint1 = annoextractor.getclip_rfpoin(pagenumber=pagenumber1, rectangleorder=rectangleorder1)  
25 argument2=dict(clip=rectangle1,fillcolor="yellow",closePath=True,lineCap=2)  
26 selector1 = PDFbf.selector(doc=doc,pagenumber=pagenumber1,clip=argument2.get("clip"))
```

```

27 intersected_lines2 = selector1.mode2_lines_Cross_Touch_Intersect()
28 manipulator2=PDFbf.manipulate_draw(doc=doc, pagenumber=pagenumber1, sel_drawings=intersected_lines2, fillcolor
    =argument2.get("fillcolor"), closePath=argument2.get("closePath"), lineCap=argument2.get("lineCap"))
29 doc = manipulator2.update_draw()
30
31 argument3 = dict(clip=rectangles1, drwcolor="blue", dashes="[3 3]0", width=3)
32 covered_rebars3 = selector1.mode1_rebars_Window_Cover_Enclosure()
33 manipulator3 = PDFbf.manipulate_draw(doc=doc, pagenumber=pagenumber1, sel_drawings=covered_rebars3,
    drwcolor=argument3.get("drwcolor"), dashes=argument3.get("dashes"), width=argument3.get("width"))
34 doc = manipulator3.update_draw()
35
36 pagenumber2 = 9
37 rectangleorder2 = 4
38 rectangle2, rfpoint2 = annoextractor.getclip_rfpoint(pagenumber=pagenumber2, rectangleorder=rectangleorder2)
39 argument4 = dict(clip=rectangle2, drwcolor="green", lineCap=2, width=4)
40 selector2 = PDFbf.selector(doc=doc, pagenumber=pagenumber2, clip=argument4.get("clip"))
41 covered_columns4 = selector2.mode1_columns_Window_Cover_Enclosure()
42 manipulator4 = PDFbf.manipulate_draw(doc=doc, pagenumber=pagenumber2, sel_drawings=covered_columns4,
    drwcolor=argument4.get("drwcolor"), lineCap=argument4.get("lineCap"), width=argument4.get("width"))
43 doc = manipulator4.update_draw()
44
45 missing_information5='''
file-level: fjhgdf.pdf, page-level:10, order-level:5, base-level: change the rebars
intersecting with the box (incomplete)'''
46 recorder5 = PDFbf.recorder(missing_information=missing_information5)
47 recorder5.recording()
48
49 updatedfilepath = filepath[:-4] + "_updated.pdf"
50 doc.save(updatedfilepath)

```

Here are all the default prompts. We encourage users to develop their own prompts to achieve higher scores.