`DocPilot`: Copilot for Automating PDF Edit Workflows in Documents

Puneet Mathur, Alexa Siu, Varun Manjunatha, Tong Sun
Adobe Research
{puneetm, asiu, manjunatha, tsun}@adobe.com
Demo Video: https://github.com/docpilot-ai/demo

Abstract

Digital documents, such as PDFs, are vital in business workflows, enabling communication, documentation, and collaboration. Handling PDFs can involve navigating complex workflows and numerous tools (e.g., comprehension, annotation, editing), which can be tedious and time-consuming for users. We introduce DocPilot, an AI-assisted document workflow Copilot system capable of understanding user intent and executing tasks accordingly to help users streamline their workflows. `DocPilot` undertakes intelligent orchestration of various tools through LLM prompting in four steps: (1) Task plan generation, (2) Task plan verification and self-correction, (3) Multi-turn User Feedback, and (4) Task Plan Execution via Code Generation and Error log-based Code Self-Revision. Our goal is to enhance user efficiency and productivity by simplifying and automating their document workflows with task delegation to `DocPilot`.

1 Introduction

Digital documents, particularly PDFs, play a crucial role in business workflows, facilitating communication, documentation, and collaboration. Handling PDF documents involves a wide array of functionalities. These include tasks such as understanding content, annotating, editing content (e.g., comments, redaction, highlights), organizing pages (e.g., crop, rotate, extract), adding signatures or watermarks, and form-filling.

Several document processing applications provide standalone tools and APIs to help users complete these tasks. However, accomplishing complex workflows involving numerous tools can be tedious and time-consuming. Additionally, unfamiliar users may face challenges in understanding and navigating the various tools available. Hence, there is a need for an AI-assisted copilot system that can comprehend the user's intent, clarify unspecified details to eliminate ambiguity in requirements, and incorporate user feedback by interacting with the user . Further, it is desired that such a system should be able to sample from a large diversity of tools and resolve interdependencies between selected sub-tasks to generate coherent task plans. The copilot must then produce executable programs consistent with the initial intent while being extensible to accommodate the addition of new tools in the future (Kudashkina et al., 2020).

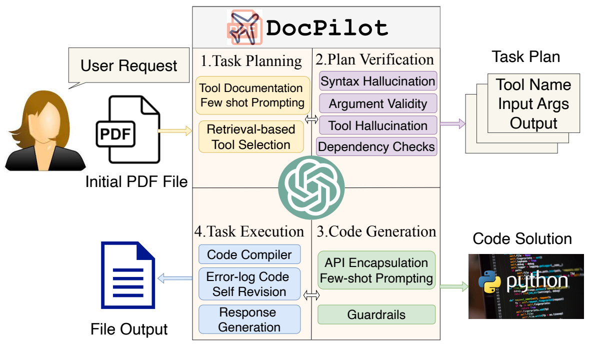
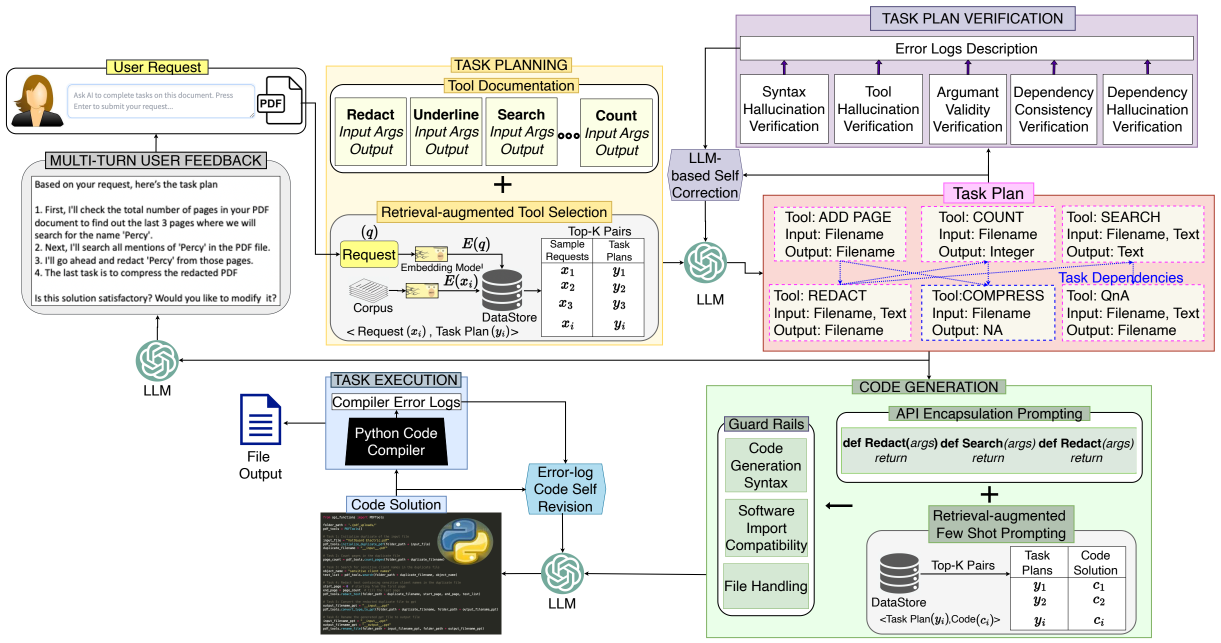


Figure 1: `DocPilot` is an LLM-assisted document workflow Copilot system capable of understanding user intent and executing PDF actions to help users achieve their editing needs.

To address these issues, we present `DocPilot` (Fig. 1), an LLM-based framework for automating editing workflows in PDF documents. Inspired by recent work like HuggingGPT, (Shen et al., 2024) and ControlLM (Liu et al., 2023), `DocPilot` takes the user's requests along with the PDF documents as inputs and leverages LLMs to infer the user's intent and transforms it into a task plan consisting of a sequence of PDF action tools. The task plan undergoes thorough verification checks to ensure accuracy and reliability. Any errors in the task plan are self-corrected by the LLM, and the final task plan is then presented to the user in easy-to-understand language, inviting feedback through conversation. Once the plan is approved by the user, `DocPilot` converts the task plan into a software program that can orchestrate external tool


User Request
Ask AI to complete tasks on this document. Press Enter to submit your request...
PDF
MULTI-TURN USER FEEDBACK
TASK PLANNING
Tool Documentation
Redact Input Args Output
Underline Input Args Output
Search Input Args Output
Count Input Args Output
Retrieval-augmented Tool Selection
Request
(q)
E(q)
Embedding Model
E(x_i)
Corpus
DataStore
Top-K Pairs
Sample Requests
Task Plans
< Request (x_i) , Task Plan (y_i)>
TASK PLAN VERIFICATION
Error Logs Description
Syntax Hallucination Verification
Tool Hallucination Verification
Argumant Validity Verification
Dependency Consistency Verification
Dependency Hallucination Verification
LLM-based Self Correction
LLM
Task Plan
Tool: ADD PAGE Input: Filename Output: Filename
Tool: COUNT Input: Filename Output: Integer
Tool: SEARCH Input: Filename, Text Output: Text
Task Dependencies
Tool: REDACT Input: Filename, Text Output: Filename
Tool:COMPRESS Input: Filename Output: NA
Tool: QnA Input: Filename, Text Output: Filename
LLM
TASK EXECUTION
Compiler Error Logs
Python Code Compiler
File Output
Error-log Code Self Revision
Code Solution
LLM
CODE GENERATION
Guard Rails
Code Generation Syntax
Software Import Compatibility
File Handling
API Encapsulation Prompting
def Redact(args) return
def Search(args) return
def Redact(args) return
Retrieval-augmented Few Shot Prompting
DataStore
Top-K Pairs
Task Plans
Code Solution
<Task Plan(y_i),Code(c_i)>

Based on your request, here's the task plan

1. First, I'll check the total number of pages in your PDF document to find out the last 3 pages where we will search for the name 'Percy'.
2. Next, I'll search all mentions of 'Percy' in the PDF file.
3. I'll go ahead and redact 'Percy' from those pages.
4. The last task is to compress the redacted PDF

Is this solution satisfactory? Would you like to modify it?

```
from api_functions import PDFTools

folder_path = './pdf_uploads/'
pdf_tools = PDFTools()

# Task 1: Initialize duplicate of the input file
input_file = "VoltGuard Electric.pdf"
pdf_tools.initialize_duplicate_pdf(folder_path + input_file)
duplicate_filename = "__input__.pdf"

# Task 2: Count pages in the duplicate file
page_count = pdf_tools.count_pages(folder_path + duplicate_filename)

# Task 3: Search for sensitive client names in the duplicate file
object_name = "sensitive client names"
text_list = pdf_tools.search(folder_path + duplicate_filename, object_name)

# Task 4: Redact text containing sensitive client names in the duplicate file
start_page = 0  # starting from the first page
end_page = page_count  # till the last page
pdf_tools.redact_text(folder_path + duplicate_filename, start_page, end_page, text_list)

# Task 5: Convert the redacted duplicate file to ppt
output_filename_ppt = "__input__.ppt"
pdf_tools.convert_type_to_ppt(folder_path + duplicate_filename, folder_path + output_filename_ppt)

# Task 6: Rename the generated ppt file to output file
input_filename_ppt = "__input__.ppt"
output_filename_ppt = "__output__.ppt"
pdf_tools.rename_file(folder_path + input_filename_ppt, folder_path + output_filename_ppt)
```

Figure 2: DocPilot: (1) Task Plan Generation decomposes user requests into a task plan using Tool Documentation prompting of Retrieval-augmented selection of PDF tools. (2) Task Plan Verification applies a series of syntax and dependency checks, and error descriptions are passed as feedback for LLM-based self-correction. (3) Multi-turn User Feedback allows users to critique the verbose task plan via the chat interface. (4) Task Plan Execution converts the approved task plan into Python code via API Encapsulation-based few-shot prompting with guardrails. Error log-based Code Self-Revision repairs code errors; the compiler executes code solution to generate output files.

API calls using LLM's code generation capabilities. The generated program is simulated using a code interpreter and detected error logs are passed as feedback to the LLM for code revision. The resultant error-free code solution executes seamless cooperation between diverse tools and provides users with a modified document that meets their expectations. To assess DocPilot's performance in supporting users, we collected user feedback on diverse workflows completed with the help of DocPilot. We find that DocPilot is effective in improving user productivity by automating repetitive tasks and simplifying complex processes.

The main contributions of DocPilot are:

(1) Accessibility: By employing LLMs as task planners, DocPilot engages users in multi-turn interactions to disambiguate complex requests. This eliminates the need to master the skillful use of document processing software, making it accessible to a broader audience.

(2) Modularity: DocPilot is designed to be highly extensible, allowing users to expand its functionality by adding more PDF tools and APIs. To achieve this, we introduce *Tool Documentation-based prompting* for generating task plans grounded in real-world tool usage, *Retrieval-Augmented Tool Selection* to tailor few-shot tool usage examples suitable for input queries, and *API Encapsulation prompting* for generating modularized code.

(3) Reliability: DocPilot promotes reliable workflow automation by mitigating task hallucinations, handling complex interdependencies between subtasks via dependency verification, and iterative self-correction to generate an executable program.

2 Related Work

Recent research informs us how LLMs can act as autonomous agents for task automation in various application domains (Xi et al., 2023; Wang et al., 2023a). **AI-powered LLM Agents**: Frameworks like AgentGPT and HuggingGPT (Shen et al., 2024) leverage LLMs as a controller to analyze user requests and invoke relevant tools for solving the task. AudioGPT (Huang et al., 2023) solves numerous audio understanding and generation tasks by connecting LLMs with input/output interface (ASR, TTS) for speech conversations. TPU (Ruan et al., 2023) proposes a structured framework tailored for LLM-based AI Agents for task planning and execution. (Zhu et al., 2023) introduced the Ghost in Minecraft (GITM), a framework of Generally Capable Agents (GCAs) that can skillfully navigate complex, sparse-reward environments with text-based interactions and develop a set of structured actions executed via LLMs. AssistGPT (Gao et al., 2023) proposed an interleaved code and language reasoning approach called Plan, Execute, Inspect, and Learn (PEIL) for processing complex images and long-form videos. RecMind (Wang et al., 2023b) designed an LLM-powered autonomous recommender agent capable of leveraging external knowledge and utilizing tools with careful planning to provide zero-shot personalized recommendations. Frameworks like AutoDroid (Wen et al., 2023) and AppAgent (Zhang et al., 2023a) presented smartphone task automation systems that can automate arbitrary tasks on any mobile application by mimicking human-like interactions such as tapping and swiping leveraged through LLMs like GPT-3.5/GPT-4. AdaPlanner (Sun et al., 2024) allows LLM agents to refine their self-generated plan adaptively in response to environmental feedback using few-shot demonstrations. (Chen et al., 2023) proposed a tool-augmented chain-of-thought reasoning framework that allows chat-based LLMs (e.g., ChatGPT) to indulge in multi-turn conversations to utilize tools in a more natural conversational manner. CREATOR (Qian et al., 2023) built a novel framework that enables LLMs to create their own tools using documentation and code realization. ControlLLM (Liu et al., 2023) proposed a Thoughts-on-Graph (ToG) paradigm that searches the optimal solution path on a pre-built tool graph to resolve parameter and dependency relations among different tools for image, audio, and video processing. LUMOS (Yin et al., 2023) trained open-source LLMs with unified data to represent complex interactive tasks. DataCopilot (Zhang et al., 2023b) built an LLM-based system to autonomously transform raw data into visualization results that best match the user's intent by designing versatile interfaces for data management, processing, and visualization. (Song et al., 2023) connects LLMs with REST software architectural style (RESTful) APIs, conducts coarse-to-fine online planning, and executes the APIs by meticulously formulating API parameters and parsing responses. Gorilla (Patil et al., 2023) explores the use of self-instruct fine-tuning and retrieval to enable LLMs to accurately select from a large, overlapping, and changing set of APIs. LLM-Grounder (Yang et al., 2023) created a novel open-vocabulary LLM-based 3D visual grounding pipeline to decompose complex natural language queries into semantic constituents for spatial object identification in 3D scenes. (Qiao et al., 2024) put forth the AUTOACT framework that automatically synthesizes planning trajectories from experience to alleviate the reliance of copilot systems on large-scale annotated data. Toolken (Hao et al., 2024) addresses the inherent problems of context length constraints and adaptability to a new set of tools by proposing LLM tool embeddings. Recent work has shown that descriptive tool documentation can be more beneficial than simple few-shot demonstrations for tool-augmented LLM automation (Hsieh et al., 2023).

3 DocPilot

Fig. 2 shows `DocPilot`, a chat-based AI assistant framework that uses LLM as a controller to translate a user's PDF editing request into an actionable task plan and orchestrates numerous software tools to realize the document editing tasks into modified PDF outputs. `DocPilot` undertakes intelligent orchestration of various LLM capabilities into an executable workflow, which includes four steps: (1) Task plan generation, (2) Task plan verification and self-correction, (3) Multi-turn User Feedback, and (4) Task Plan Execution via Code Generation and Error log-based Code Self-Revision.

3.1 Task Plan Generation

User requests involve several intricate intentions that need to be decomposed into a sequence of sub-tasks to be solved to achieve the final output. The task planning stage utilizes an LLM to analyze the user request and determine the execution orders of the PDF Tool API calls based on their resource dependencies. We represent the LLM-generated task plan in the JSON format to parse the sub-tasks through slot filing. Each sub-task is composed of five slots - "task", "id", "dep", "args", and "return" to represent the PDF tool function name, unique identifier, dependencies, arguments, and returned values, respectively. To better understand the intention and criteria for task planning, we utilize *Tool Documentation-based prompting*. The task planning prompt contains documentation of the PDF tool APIs (see Table 1 for the API list), briefly mentioning each function's utilities, arguments, and return values. Without explicitly exposing the API implementation, this novel prompting technique ensures that our methodology embraces API-level abstraction and encapsulation by restricting access to proprietary data and internal functions for enhanced user privacy to black-box LLM models.

Retrieval-Augmented Tool Selection: The task planning stage may involve a large number of tools. Many of these tools might not be relevant to the user request, and including all in the LLM prompt may lead to reduced context length for subsequent chat prompting. Hence, based on the incoming user request, we utilized a retrieval-augmented selection approach to only include the most relevant few-shot examples in the task plan prompt.

Let q denote the user request and $Z = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ represents the set of few-shot examples curated for the task plan prompt. Each example consists of a sample request (x_i) paired with the corresponding ground truth task plan (y_i). We use a text embedding model E to encode the sample user requests from the few-shot examples into vector representations - $\{E(x_i), E(x_2), \cdots, E(x_n)\}$, respectively. We construct a datastore of few-shot examples with keys as vectorized sample requests and values as ground truth task plans. We encode the incoming user request via embedding model as $E(q_i)$ at inference. Next, we use the k-nearest neighbor technique with the Euclidean distance metric to query top-k sample requests from the datastore which are semantically most similar to the encoded user query. The selected pairs of user requests and their task plans, similar to the example shown in Fig. **??**, are utilized in prompting the LLM model to generate the task plan for the current user query.

3.2 Task Plan Verification and Self-Correction

LLM-generated task plans involve a risk of hallucinations when selecting unspecified functions, connecting dependency connections, or invalid argument parsing, which may lead to undesired outputs. We introduce two novel modules to ensure robustness in the generated task plans against logical inconsistencies: "Task Plan Verification" and "LLM-based Self-Correction".

First, the "Task Plan Verification" consists of three *static composition verification* and two *inter-task dependency verification* checks on the generated task plan JSON (Appendix Figure 6 shows an illustrative example). Static composition verification checks the individual constituents of the task plan for hallucinations on syntax, tool name and API calls, and function arguments (Appendix A.4). Second, the inter-task dependency verification checks the validity of dependency relations between various function calls in the task plan as:
(1) **Dependency hallucination verification** – Each function call depends on arguments provided by the user request or outputs of preceding functions in the task plan. We add checks to ensure the LLM does not hallucinate dependencies referencing non-existent or future function calls in the task plan.
(2) **Dependency consistency verification**: Each function call in the task plan sequence may depend on one or more prior function calls. These functional dependencies need not be linear and can be better represented as a graph of connected components (also known as a dependency graph). A function call may often try to access resources from another function call. However, in some cases, these interdependencies may be cyclic or unreachable. Hence, subsequent function calls can not proceed ahead without resolving the prior. This may give rise to deadlock conditions during the task execution. To avoid deadlocks and resource conflicts, it is important to ensure that there are no cyclic dependencies between the intermediate function calls. To solve this problem, we create a dependency graph G from the task plan T where all function calls denote the set of nodes V, and their interdependencies represent the set of edges E of the graph. To check for the presence of cyclic dependencies in a graph, it should be sufficient to check if the dependency graph is a directed acyclic graph (DAG). We utilize Kahn's algorithm (Kahn, 1962) to evaluate this condition, which involves performing a topological sort of the dependency graph followed by a depth-first traversal to evaluate if all nodes have been visited exactly once without repetition. Violation of this condition indicates a lack of DAG property. The dependency error is then attributed to the API function corresponding to the failure node in the graph.
LLM-based Self-Correction: The verification module generates error log descriptions based on the nature of the fault and the responsible API functions. The error logs and original task plan sequence are passed as feedback to the LLM model as a chat completion prompt to rework the solution. This process recursively improves the task plan solution until no further errors are encountered.

3.3 Multi-turn User Feedback

User consent is a prerequisite for executing actions that could potentially alter a user's proprietary PDF files. Adhering to this principle, the meticulously verified error-free task plan is transformed into a clear and comprehensible layman explanation through LLM prompting. This elucidation is then presented to the user through the chat interface. Subsequently, the user can engage in a multi-turn chat conversation with the LLM to challenge the proposed task plan and provide additional feedback. The user's input is integrated to iteratively refine the task plan by recursively following the task planning and verification stages. This iterative process of modifying the task plan through multi-turn chat conversations continues until the user is content with the solution or decides to abort the request.

3.4 Task Plan Execution

The task plan obtained in the last step lists tool APIs with corresponding arguments and return values. However, the sequence of function calls that need to be executed is not linear due to interdependencies between the API calls. Hence, there is a need to convert the task plan into a software program with a logical flow of information. We introduce the Task Plan Program Execution step, where the LLM converts the task plan into a software program that can be executed to give the desired output PDF file to the user. This stage is divided into two modules - "Task Plan Code Generation" and "Error log-based Code Self-Revision".

Task Code Generation: We utilize the LLM code generation abilities to transform the task plan sequence into executable Python code. However, unrestricted LLM-generated code may hallucinate functions that do not exist, use incompatible libraries, be unable to navigate file handling at the user's end or perform flawed executions that may harm user data, leading to deteriorated user trust.

To safeguard against such detrimental cases, we incorporate a novel *API Encapsulation*-based few-shot prompting with strong guardrails. The prompt consists of the code documentation of *PDFTools()* class, which encapsulates publicly accessible tool API function methods and exposes limited information regarding the function name, input arguments, and returned values. The LLM can utilize this abstracted view of tool APIs for program synthesis without knowing or modifying their internal code implementation. In this manner, we alleviate the problem of function hallucinations while ensuring that only well-trusted and rigorously tested API functions are used for user data modifications. Additionally, we augment the prompt with a few shot examples of task plans (y_i) retrieved during the task plan generation step, paired with their corresponding ground truth Python code solutions (c_i) to guide the code generation process to remain faithful to task plan logic. Further, we designed stringent guard rails to safeguard program execution by ensuring consistency in code generation syntax, avoiding lazy code generation phenomenon of LLMs, machine compatibility of software imports, safe-listing of approved Python packages, secure access of file addresses, and cautious file handling. More details in Appendix Sec. A.5.

Error log-based Code Self-Revision: Despite carefully crafted prompts and strong guard rails, the generated program solution may give errors upon code execution. To screen for errors in advance and recover from a failed execution state, we propose Error log-based Self-Revision prompting. In particular, we build a Python code interpreter to simulate code execution in a sandboxed environment to mimic the actual PDF file editing. Compilation errors from the code interpreter are captured as error logs and combined with the original code solution to be passed as feedback to the LLM model to rework the code solution. The code interpreter again tests the reworked code solution to check for errors, and the process continues recursively until the code solution is improved and no further errors are encountered. Fig. 7 in the Appendix shows an example code solution. Finally, we execute the resultant error-free code solution to produce the PDF document modifications requested by the user.

4 Implementation Details

Backbone LLM: We use GPT-4 API through the Microsoft Azure platform for all our experiments. We also tried GPT-3.5 model but it performed consistently worse than GPT-4 owing to its limited context length and weak code generation abilities.

RAG architecture: We utilized FAISS to construct the data stores for the Retrieval-augmented tool selection module. We used SentenceBert (Reimers and Gurevych, 2019) as the embedding model. We used Scikit Learn's KNN library to get top-k request-task plan pairs. We used Gradio for the demo UI hosted on the AWS cloud platform.

5 User Evaluation

We conducted a user evaluation to assess the efficacy of `DocPilot` in supporting users' PDF workflows. The research goals were as follows:

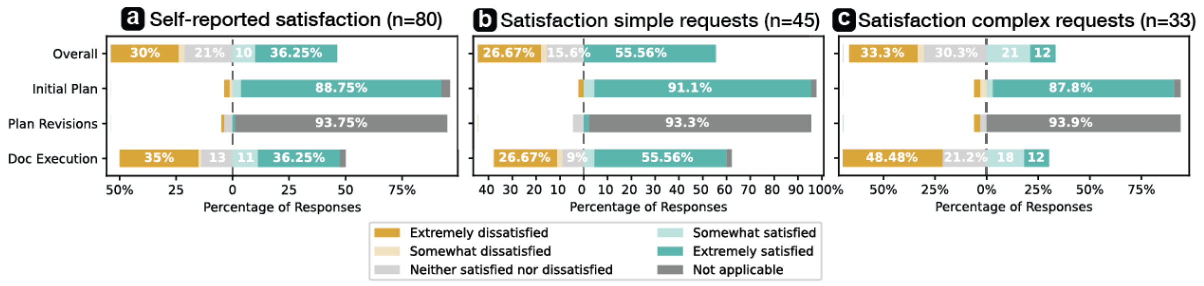


Figure 3: a) Self-reported user satisfaction scores from using `DocPilot` to complete 80 workflow requests. b) Simple requests (<=5 actions) had higher satisfaction scores compared to c) complex requests (>5 actions).

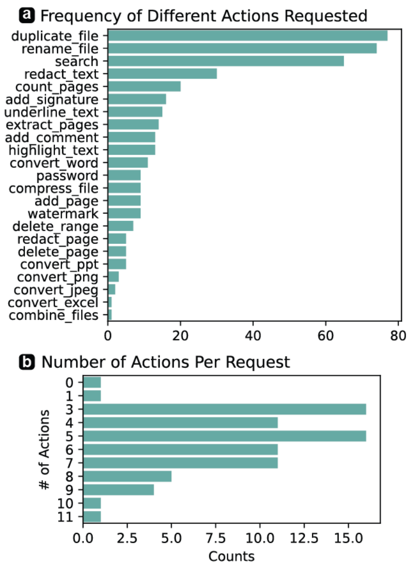


Figure 4: (a) Frequency of different actions referenced in task plans; (b) Distribution of actions executed per request during user evaluation.

R1 Measure `DocPilot`'s performance in suggesting a reasonable plan in response to a user-provided multi-step workflow. Relatedly, we wanted to understand how well our system handled ambiguity in user requests.

R2 Understand when/how breakdowns happen and whether users are able to refine their plan through conversation with `DocPilot`.

Our data collection focused on a case study with one expert PDF user who works on PDF processing tasks daily as part of his professional work. Our evaluator was hired through UpWork with expertise in PDF workflows. The evaluator interacted with the `DocPilot` app (Fig. 5) to complete several workflows and provided feedback through a survey form (Methodology details in Appendix A.7.1).

5.1 Results

5.1.1 User Workflow Requests

We collected data from 80 workflow requests. 16 workflows were user-provided based on the users' real-world PDF workflows, and 64 were workflows suggested to the user. We provided the suggested workflows to ensure that the workflows evaluated included a variety of the types of actions used and the number of actions requested. Appendix A.7.3 has examples of user-provided and suggested workflows. Fig. 4a shows the frequency of different actions referenced as part of the user's requests. The most common actions included duplicating a file (77), renaming a file (74), searching content (65), redacting content (30), and counting pages (20). Fig. 4b shows the distribution of actions executed per request with a median of 5 (IQR $4-7$).

5.1.2 Self-Reported Satisfaction Ratings

To understand `DocPilot`'s performance in suggesting a satisfactory plan in response to a user's request (R1), we collected self-reported measures of user satisfaction after each step of the `DocPilot` pipeline (Fig. 1). Fig. 3a shows user satisfaction aggregated over all 80 workflow requests. `DocPilot` performs extremely well in suggesting a reasonable initial plan, with 88.75% ($71/80$) receiving positive ratings of *Extremely satisfied* and the majority of requests not requiring plan revisions from the user. The main concern of dissatisfaction with `DocPilot` was related to the task execution step, which received the *Extremely satisfied* ratings only in 36.25% ($29/80$) of requests, similarly reflected in the Overall satisfaction ratings.

To understand whether workflow complexity impacts the system's efficacy in planning, we further analyze satisfaction ratings by complexity. Fig. 3(b-c) show satisfaction ratings for simple (n=45) and complex (n=33) requests. We consider simple requests as those requiring 5 actions or less to be executed to fulfill the users' request. We observe that the positive satisfaction ratings (*Extremely satisfied* + *Somewhat satisfied*) are higher for simple requests (25/45, or 55.55%) compared to those on complex requests (11/33, or 33.33%). Simple requests also resulted in much higher satisfaction with task execution (27/45, or 60%) compared to complex requests (10/33, or 30.3%).

5.1.3 Qualitative Feedback

To further understand breakdowns in DocPilot from the users' perspective (R2), we conducted a thematic analysis of users' requests that resulted in failure as well as open-ended user feedback. For the Task Planning step, the user majorly provided positive comments, *"The plan is concise, to the point, and explained well. I like that the assistant understands the request completely"*.

However, we also report a small number of negative comments that were primarily centered on the Task Execution step, where DocPilot either missed a step or detail in the resulting files. The user had certain expectations of the results based on the plan suggested by DocPilot, which were unmet. We observed instances where DocPilot executed the action incorrectly, *"Instead of deleting pages 1, 2, and 5, the assistant deleted pages 1 to 4"*. Users also reported a few cases where DocPilot simply missed an action, *"...the assistant successfully converted pages but was unable to add digital signatures."*. We also noticed some cases where DocPilot did not understand the multimodal content in the document properly, which in turn affected performance for actions that required searching for content in the document. For example, *"My request is to redact the numerical values in the 'Annual Energy Use' and 'Water' columns of the table. However, the assistant does not understand and redacts incorrect words."* Lastly, we also recorded a handful of cases where the user had high expectations that were beyond the DocPilot's current tooling capabilities (e.g., replacing text and images). In the future, we aim to handle such discrepancies by improving prompt engineering, extending the PDF tool APIs available to DocPilot, and integrating Large Multimodal Models such as GPT-4V for multimodal document search/QA tasks.

5.1.4 LLM Iterations & Self-Correction

To quantify breakdowns due to program execution (R2), we analyzed the code interpreter error logs for output code. A small number of workflow requests (8/80) required more than one LLM self-correction step (Sec. 3.2) to reach a desirable action plan. In contrast, the majority of requests (69/80) required at least one LLM self-correction (Sec. 3.4) to produce an executable program that passed all checks. More details in Appendix Table 2.

6 Discussion, Limitations & Future Work

Our evaluation results show that DocPilot's Task Planning step is effective for most workflows as the proposed task plan captures the user's intent well and requires few clarifications by the user. Only 10% of the evaluated workflows required more than one LLM iteration to self-correct the generated task plans. The majority of breakdowns we observed occurred due to a mismatch in the user's expectations between the plan suggested by DocPilot and how it was executed. Our current interface with DocPilot primarily uses a conversational UI. Leveraging interactions from graphical UIs can help lessen this gap by providing the user affordances for direct manipulation in content selection when executing a workflow (Ma et al., 2023). Additionally, DocPilot may allow users to edit action parameters (e.g., page number, password) directly rather than requiring the user to type a new request. Both of these future works could increase user control and understanding of the system plan (Amershi et al., 2019). Quantitatively, we observe that most workflows (69/80) required at least one LLM-based code revision to produce an error-free program, thus introducing latency and impacting the utility of the tool. Self-reported ratings indicate more failures in the task execution step for complex workflows. Hence, our future work will focus on instruction-tuning LLMs on pairs of ground truth task plans and Python code.

7 Conclusion

We present Docpilot, an LLM-powered copilot for automating document workflows. Our copilot helps novices plan document workflows by selecting the appropriate tools and executing the task plan autonomously. DocPilot benefits the users by enhancing their accessibility, being extensible to include more tools, and being consistently reliable.

8 Ethics Statement

Our experiments used publicly available API-accessible LLM - GPT-3.5 and GPT-4 (March 2024 version). For our user evaluation, participants' personal information is maintained confidential and private. Participants were trained and informed about the task before participating. Participants were also compensated fairly, with each annotator paid equal to or more than 15 USD/hr.

References

Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for human-ai interaction. In *CHI 2019*. ACM. CHI 2019 Honorable Mention Award.

Kranti Chalamalasetti, Jana Götze, Sherzod Hakimov, Brielen Madureira, Philipp Sadler, and David Schlangen. 2023. clembench: Using game play to evaluate chat-optimized language models as conversational agents. *arXiv preprint arXiv:2305.13455*.

Zhipeng Chen, Kun Zhou, Beichen Zhang, Zheng Gong, Xin Zhao, and Ji-Rong Wen. 2023. ChatCoT: Tool-augmented chain-of-thought reasoning on chat-based large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14777–14790, Singapore. Association for Computational Linguistics.

Difei Gao, Lei Ji, Luowei Zhou, Kevin Qinghong Lin, Joya Chen, Zihan Fan, and Mike Zheng Shou. 2023. Assistgpt: A general multi-modal assistant that can plan, execute, inspect, and learn. *arXiv preprint arXiv:2306.08640*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2024. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*.

Rongjie Huang, Mingze Li, Dongchao Yang, Jiatong Shi, Xuankai Chang, Zhenhui Ye, Yuning Wu, Zhiqing Hong, Jiawei Huang, Jinglin Liu, et al. 2023. Audiogpt: Understanding and generating speech, music, sound, and talking head. *arXiv preprint arXiv:2304.12995*.

Arthur B Kahn. 1962. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Katya Kudashkina, Patrick M. Pilarski, and Richard S. Sutton. 2020. Document-editing assistants and model-based reinforcement learning as a path to conversational ai. *ArXiv*, abs/2008.12095.

Jiaju Lin, Haoran Zhao, Aochi Zhang, Yiting Wu, Huqiuyue Ping, and Qin Chen. 2023. Agentsims: An open-source sandbox for large language model evaluation.

Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, and Wenhai Wang. 2023. Controlllm: Augment language models with tools by searching on graphs. *ArXiv*, abs/2310.17796.

Xiao Ma, Swaroop Mishra, Ariel Liu, Sophie Su, Jilin Chen, Chinmay Kulkarni, Heng-Tze Cheng, Quoc Le, and Ed Chi. 2023. Beyond chatbots: Explorellm for structured thoughts and personalized model responses. *arXiv preprint arXiv:2312.00763*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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

Cheng Qian, Chi Han, Yi Ren Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Conference on Empirical Methods in Natural Language Processing*.

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Eleanor Jiang, Chengfei Lv, and Huajun Chen. 2024. Autoact: Automatic agent learning from scratch via self-planning. *ArXiv*, abs/2401.05268.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Conference on Empirical Methods in Natural Language Processing*.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Xingyu Zeng, and Rui Zhao. 2023. Tptu: Task planning and tool usage of large language model-based ai agents. *arXiv preprint arXiv:2308.03427*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world restful apis.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2024. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems*, 36.

Lei Wang, Chengbang Ma, Xueyang Feng, Zeyu Zhang, Hao ran Yang, Jingsen Zhang, Zhi-Yang Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji rong Wen. 2023a. A survey on large language model based autonomous agents. *ArXiv*, abs/2308.11432.

Yancheng Wang, Ziyan Jiang, Zheng Chen, Fan Yang, Yingxue Zhou, Eunah Cho, Xing Fan, Xiaojiang Huang, Yanbin Lu, and Yingzhen Yang. 2023b. Recmind: Large language model powered agent for recommendation. *arXiv preprint arXiv:2308.14296*.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering llm to use smartphone for intelligent task automation. *arXiv preprint arXiv:2308.15272*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Qin Liu, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huan, and Tao Gui. 2023. The rise and potential of large language model based agents: A survey. *ArXiv*, abs/2309.07864.

Binfeng Xu, Xukun Liu, Hua Shen, Zeyu Han, Yuhan Li, Murong Yue, Zhiyuan Peng, Yuchen Liu, Ziyu Yao, and Dongkuan Xu. 2023a. Gentopia.AI: A collaborative platform for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 237–245, Singapore. Association for Computational Linguistics.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023b. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*.

Jianing Yang, Xuweiyi Chen, Shengyi Qian, Nikhil Madaan, Madhavan Iyengar, David F. Fouhey, and Joyce Chai. 2023. Llm-grounder: Open-vocabulary 3d visual grounding with large language model as an agent. *ArXiv*, abs/2309.12311.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Raghavi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. 2023. Lumos: Learning agents with unified data, modular design, and open-source llms. *ArXiv*, abs/2311.05657.

China. Xiaoyan Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023a. Appagent: Multimodal agents as smartphone users. *ArXiv*, abs/2312.13771.

Wenqi Zhang, Yongliang Shen, Weiming Lu, and Yue Ting Zhuang. 2023b. Data-copilot: Bridging billions of data and humans with autonomous workflow. *ArXiv*, abs/2306.07209.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 2023. Ghost in the minecraft: Generally capable agents for open-world enviroments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*.

A Appendix

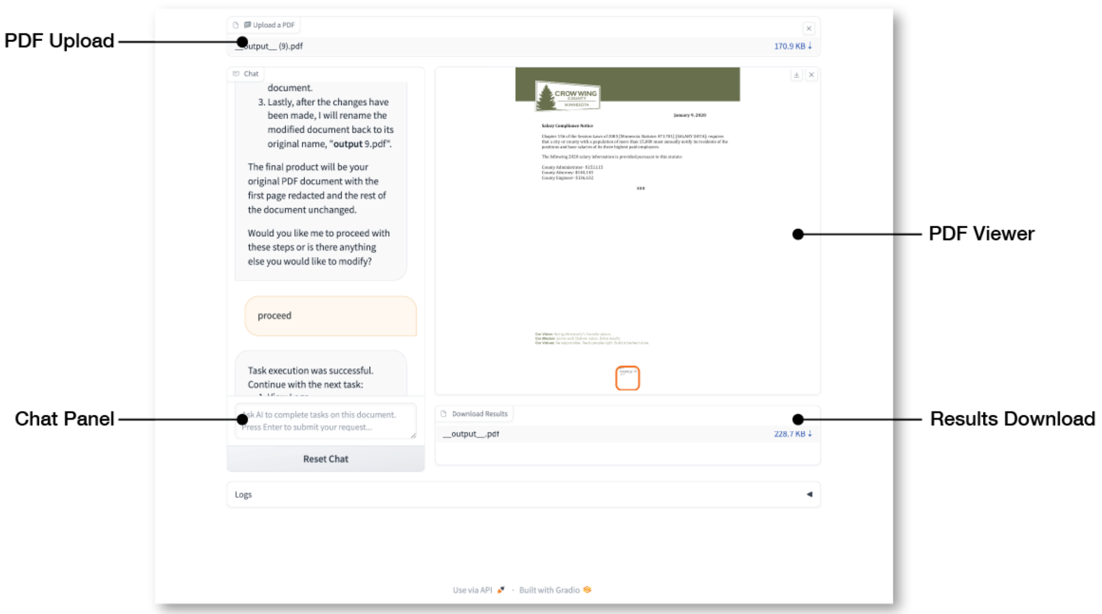
A.1 `DocPilot` Demo App

Figure 5 shows the `DocPilot` demo app. This app was also used by our evaluator to complete all workflow requests. The app was built using Gradio[1]. The app requires an OpenAI token to access the GPT-4 model. The interface includes a PDF upload panel, a PDF viewer, and a chat panel. Users can directly upload their input PDF file and type in their request in the chat panel. The chat panel facilitates multi-turn chat and shows all the intermediate interactions and results generated by the system. Once a workflow is executed, the user can download the resulting files for inspection. The users also has the ability to reset their chat history to start a new workflow conversation.

A.2 Implementation Details

Backbone LLM: We use GPT-4 API through the Microsoft Azure platform for all our experiments. We also tried GPT-3.5 model but it performed consistently worse that GPT-4 owing to its limited context length and weak code generation abilities.
RAG architecture: We utilized FAISS to construct the data stores for the Retrieval-augmented tool selection module. We used SentenceBert (Reimers and Gurevych, 2019) as the embedding model. We used Scikit Learn's KNN library to get top-k request-task plan pairs. We used Gradio for the demo UI hosted on the AWS cloud platform.
LLM Agent Evaluations: ToolBench (Xu et al., 2023b) released a tool manipulation benchmark

[1]https://www.gradio.app



document.
3. Lastly, after the changes have been made, I will rename the modified document back to its original name, "**output** 9.pdf".

The final product will be your original PDF document with the first page redacted and the rest of the document unchanged.

Would you like me to proceed with these steps or is there anything else you would like to modify?

proceed

Task execution was successful. Continue with the next task:

Ask AI to complete tasks on this document. Press Enter to submit your request...

Figure 5: UI for `DocPilot`

consisting of diverse software tools for real-world tasks to evaluate LLM capabilities for tool manipulation. AgentSim (Lin et al., 2023) created an interactive infrastructure for researchers to evaluate the task completion abilities of LLM agents in a simulated environment. WebArena (Zhou et al., 2023) introduces a benchmark on interpreting high-level realistic natural language commands to concrete web-based interactions. ClemBench (Chalamalasetti et al., 2023) provides a systematic evaluation of LLM's capability to follow game-play instructions. (Xu et al., 2023a) created the GentPool platform that registers and shares user-customized, composable, and collaborative agents. WebShop (Yao et al., 2022) is another challenging benchmark that tests LLM agent's capabilities to navigate multiple types of webpages, find, customize, and purchase a product given text instruction in an e-commerce website simulation with 1.18 million real-world products. This is the first work to provide a novel benchmark for evaluating LLM agent workflows in a document editing software environment.

A.3 DocPilot PDF Tool APIs

Table 1 shows the set of PDF tool APIs and their descriptions available during the task plan generation in DocPilot.

A.4 Task Plan Verification Module

Figure **??** shows a qualitative example of task verification checks - Syntax Hallucination, Tool Hallucination, Argument Validity, Dependency Hallucination, and Dependency Consistency.

Static composition verification checks the individual constituents of the task plan for hallucinations on syntax, tool name and API calls, and function arguments:

1. **Syntax hallucination verification** – Incorrect JSON formatting of the task plan may cause downstream JSON parsing errors. This verification step ensures the task plan returned is a list of Python maps with key-value pairs denoting function names, dependencies, input arguments, and returned values.

2. **Tool hallucination verification** – Despite prompting the syntactically correct task plan, LLMs may hallucinate invalid tool names and API calls. This step ensures that all PDF tool APIs are valid and present in the documentation.

3. **Argument validity verification** - Each function in the task plan has a pre-defined number and type of arguments and return values. Any hallucinations in this regard may cause errors during program execution. Hence, we check for any extra, missing, or incorrect arguments in each task plan sequence function call.

Tool	Description
Duplicate	Initializes a duplicate of the input file and saves it as "input.pdf"
Rename	Renames the input file to the output file name with a default value "output.pdf"
Search	Returns a list of text strings matching the matching query found in the input document denoted as filename; Otherwise, returns an empty list.
QnA	Answers a question in the form of a text string from the LLM query result.
Count Pages	Counts the number of pages in the PDF file and returns it as an integer
Compress	Reduce the PDF file size given as the input filename and save the new file as the output filename.
Convert to PPT	Convert the input PDF file into a PowerPoint presentation (ppt) file and save the converted file as output filename
Convert to Word	Convert the input PDF file into a Word (docx) file and save the converted file as output filename
Convert to PNG	Convert the input PDF file into a PNG image file and save the converted file as output filename
Convert to JPEG	Convert the input PDF file into a JPEG image file and save the converted file as output filename
Convert to TIFF	Convert the input PDF file into a TIFF image file and save the converted file as output filename
Convert to Excel	Convert the input PDF file into an Excel (.xlsx) file and save the converted file as output filename
Add Password	Add the input passcode text string as password protection to the input PDF file.
Check Password	Check if the input PDF file has password protection
Combine Files	Combine all files given in the list of input files into a single file and save the output file as output_filename
Redact Pages	Redacts all pages of the input PDF file in the range starting from start_page till end_page. Start and end pages are 1-indexed
Redact Text	Redacts all mentions of strings in the list denoted by "object_name" from the input PDF file within the range starting from the start page to the end page. Start and end pages are 1-indexed
Highlight Text	Highlight all instances in the input PDF file matched by input string
Underline Text	Underline all instances in the input PDF file matched by input string
Extract Pages	Extracts pages from the input PDF in the range from the start page to the end page. Start and end pages are 1-indexed
Delete Page	Deletes page denoted by integer "page_number_to_delete" from the input PDF file; the page number to be deleted is 1-indexed
Delete Page Range	Deletes pages from the input PDF in the range from the start page to the end page. Start and end pages are 1-indexed
Add Signature	Add an image of the signature on the page denoted by "page_number" in the input PDF file; input page number is 1-indexed
Add Watermark	Fix the watermark image on the input PDF file pages in the range from the start page to the end page. Start and end pages are 1-indexed
Add Comment	Add input text comment in the input PDF file at the input page number or by default at the last page
Add Page Text	Add a new page to the input PDF file at the page number specific by "page number". The new page has the text string "content" added to it. Page numbers are 1-indexed

Table 1: Overview of tasks and associated tools in DocPilot

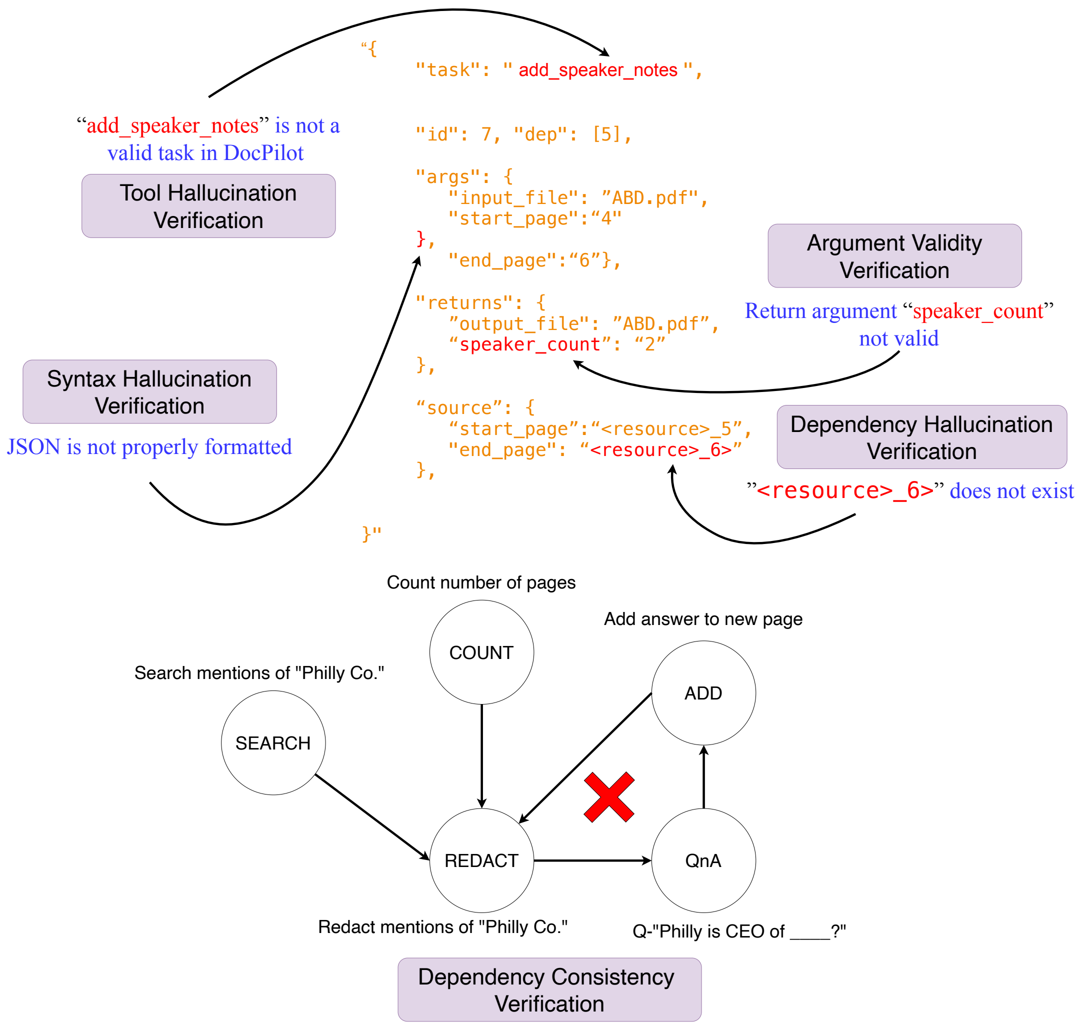

```
"{
    "task": " add_speaker_notes ",

    "id": 7, "dep": [5],

    "args": {
       "input_file": "ABD.pdf",
       "start_page":"4"
    },
       "end_page":"6"},

    "returns": {
       "output_file": "ABD.pdf",
       "speaker_count": "2"
    },

    "source": {
       "start_page":"<resource>_5",
       "end_page": "<resource>_6>"
    },

"}"
```
"add_speaker_notes" is not a valid task in DocPilot
Tool Hallucination Verification
Argument Validity Verification
Return argument "speaker_count" not valid
Syntax Hallucination Verification
JSON is not properly formatted
Dependency Hallucination Verification
"<resource>_6>" does not exist
Count number of pages
COUNT
Add answer to new page
ADD
Search mentions of "Philly Co."
SEARCH
REDACT
QnA
Redact mentions of "Philly Co."
Q-"Philly is CEO of ____?"
Dependency Consistency Verification

Figure 6: Task Verification example for syntax hallucinations, tool hallucinations, argument validity, dependency hallucinations, and dependency consistency.

A.5 Guard Rails for Task Plan Code Generation

1. Code Generation Syntax: Most state-of-the-art LLM architectures are geared towards a conversational chat interface trained via human chat feedback (Ouyang et al., 2022). Consequently, LLMs may occasionally interleave conversational text with code syntax during generation. Moreover, some LLMs may even provide pseudo-code instead of independently executable Python code. In order to avoid such pitfalls, we add explicit instructions in the prompt to force the LLM to follow a pre-defined Python syntax with all other extraneous text formatted as comments in the code block. Moreover, it has been recently reported that SOTA LLMs like ChatGPT-3.5 and GPT-4 tend to show signs of "lazy assistance" wherein they refuse to generate fully executable code, instead explaining how the user could answer the question. We carefully designed the LLM prompt with explicit instructions to satisfy our need for independently executable Python code as the output of the step.

2. Software Import Compatibility: Allowing unrestricted permission to import any software library or package specified in the LLM-generated code may potentially harm user privacy and security. Some of these may not be compatible with the user hardware, conflict with existing software versions, or be no longer supported by programming languages. Hence, appropriate guard rails are needed to regulate what software libraries can be imported during task plan execution. Towards this, we maintain a software safe-list of approved Python packages, libraries, and executable files in the tool API documentation that are permitted to be invoked by LLM-generated code. We add explicit instructions to the prompts to forbid the LLMs from generating any overhead software libraries and packages for code execution. Instead, we pre-append the safe-listed software imports to the generated code.

3. File Handling: An essential aspect of copilot-driven external file modifications is safeguarding data privacy by not exposing the input file names and types that need to be modified and the resultant output files generated by the copilot to the LLM. We achieve this by strongly type-casting all references to input and output file names and addresses in the generated code to their actual values at the code execution step. Further, we impose strict directory access restrictions on the copilot system, preventing accessing, reading, or saving files without explicit user permissions. The code execution step involves creating a copy of all files required as inputs to a temporary directory and saving all intermediate files and the final output PDF to avoid overwriting or modifying non-permitted files.

A.6 Task Plan Code Generation Examples

Figure 7 shows a qualitative example of a code solution generated by DocPilot corresponding to the task plan response to the user request - "Hey, can you please blacken out any sensitive client names from my 'VoltGaurd Electric.pdf' file and convert it into a PowerPoint presentation".

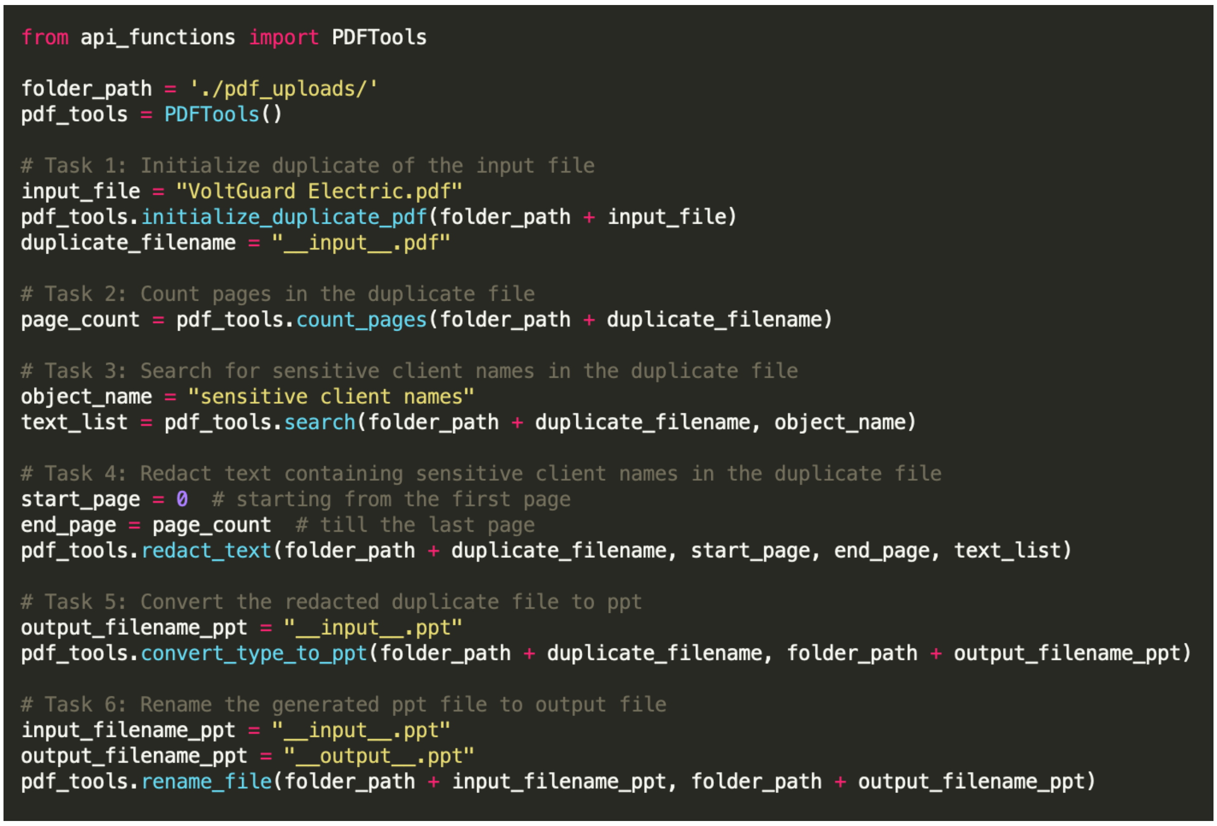
A.7 Qualitative Examples

Figures 8 and 9 show qualitative examples of PDf files edited by a user through `DocPilot`.

A.7.1 Evaluation Procedure

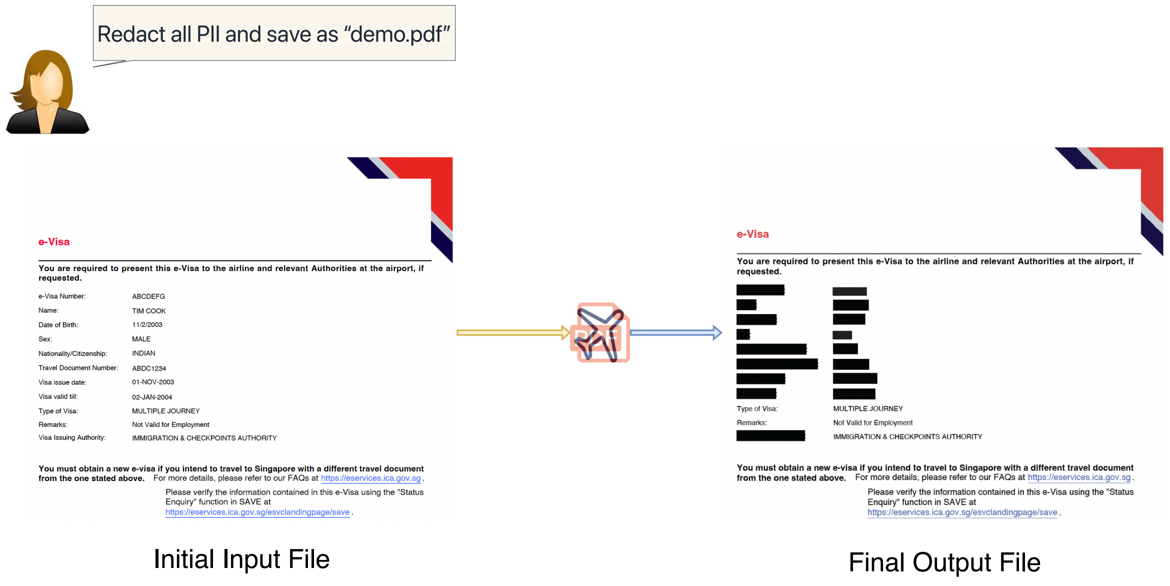
As an introduction, the user was provided with a guidelines document that detailed the PDF capabilities of `DocPilot`. The user was also provided access to the `DocPilot` app (Figure 5) and given a short tutorial on its usage. For data collection, the user was provided with a repository of PDF documents (n=61), a suggested prompt library (n=151), and a link to a survey form for data collection. The user was instructed their overall evaluation goal was to complete several PDF workflows as best as possible with the help of `DocPilot`.

For the first task, the user was asked to select a PDF document and either craft a prompt based on their own usage or select one from the suggested list. For the second task, the user was asked to prompt `DocPilot` with their workflow request and to carefully review `DocPilot`'s responses. The user was encouraged to request changes as needed to the suggested plan until satisfied that it met their workflow goals. For the third task, the user was asked to review the actions as executed by `DocPilot` in the resulting files. Last, after completing all interactions with `DocPilot` for one workflow, regardless of success or failure, the user was instructed to



```python
from api_functions import PDFTools

folder_path = './pdf_uploads/'
pdf_tools = PDFTools()

# Task 1: Initialize duplicate of the input file
input_file = "VoltGuard Electric.pdf"
pdf_tools.initialize_duplicate_pdf(folder_path + input_file)
duplicate_filename = "__input__.pdf"

# Task 2: Count pages in the duplicate file
page_count = pdf_tools.count_pages(folder_path + duplicate_filename)

# Task 3: Search for sensitive client names in the duplicate file
object_name = "sensitive client names"
text_list = pdf_tools.search(folder_path + duplicate_filename, object_name)

# Task 4: Redact text containing sensitive client names in the duplicate file
start_page = 0  # starting from the first page
end_page = page_count  # till the last page
pdf_tools.redact_text(folder_path + duplicate_filename, start_page, end_page, text_list)

# Task 5: Convert the redacted duplicate file to ppt
output_filename_ppt = "__input__.ppt"
pdf_tools.convert_type_to_ppt(folder_path + duplicate_filename, folder_path + output_filename_ppt)

# Task 6: Rename the generated ppt file to output file
input_filename_ppt = "__input__.ppt"
output_filename_ppt = "__output__.ppt"
pdf_tools.rename_file(folder_path + input_filename_ppt, folder_path + output_filename_ppt)
```

Figure 7: An example of task plan code solution generated for the query - "Hey, can you please blacken out any sensitive client names from my 'VoltGaurd Electric.pdf' file and convert it into a PowerPoint presentation"



e-Visa

You are required to present this e-Visa to the airline and relevant Authorities at the airport, if requested.

e-Visa Number:	ABCDEFG
Name:	TIM COOK
Date of Birth:	11/2/2003
Sex:	MALE
Nationality/Citizenship:	INDIAN
Travel Document Number:	ABDC1234
Visa issue date:	01-NOV-2003
Visa valid till:	02-JAN-2004
Type of Visa:	MULTIPLE JOURNEY
Remarks:	Not Valid for Employment
Visa Issuing Authority:	IMMIGRATION & CHECKPOINTS AUTHORITY

You must obtain a new e-visa if you intend to travel to Singapore with a different travel document from the one stated above. For more details, please refer to our FAQs at https://eservices.ica.gov.sg .

Please verify the information contained in this e-Visa using the "Status Enquiry" function in SAVE at https://eservices.ica.gov.sg/esvclandingpage/save .

e-Visa

You are required to present this e-Visa to the airline and relevant Authorities at the airport, if requested.

[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
Type of Visa:	MULTIPLE JOURNEY
Remarks:	Not Valid for Employment
[redacted]	IMMIGRATION & CHECKPOINTS AUTHORITY

You must obtain a new e-visa if you intend to travel to Singapore with a different travel document from the one stated above. For more details, please refer to our FAQs at https://eservices.ica.gov.sg .

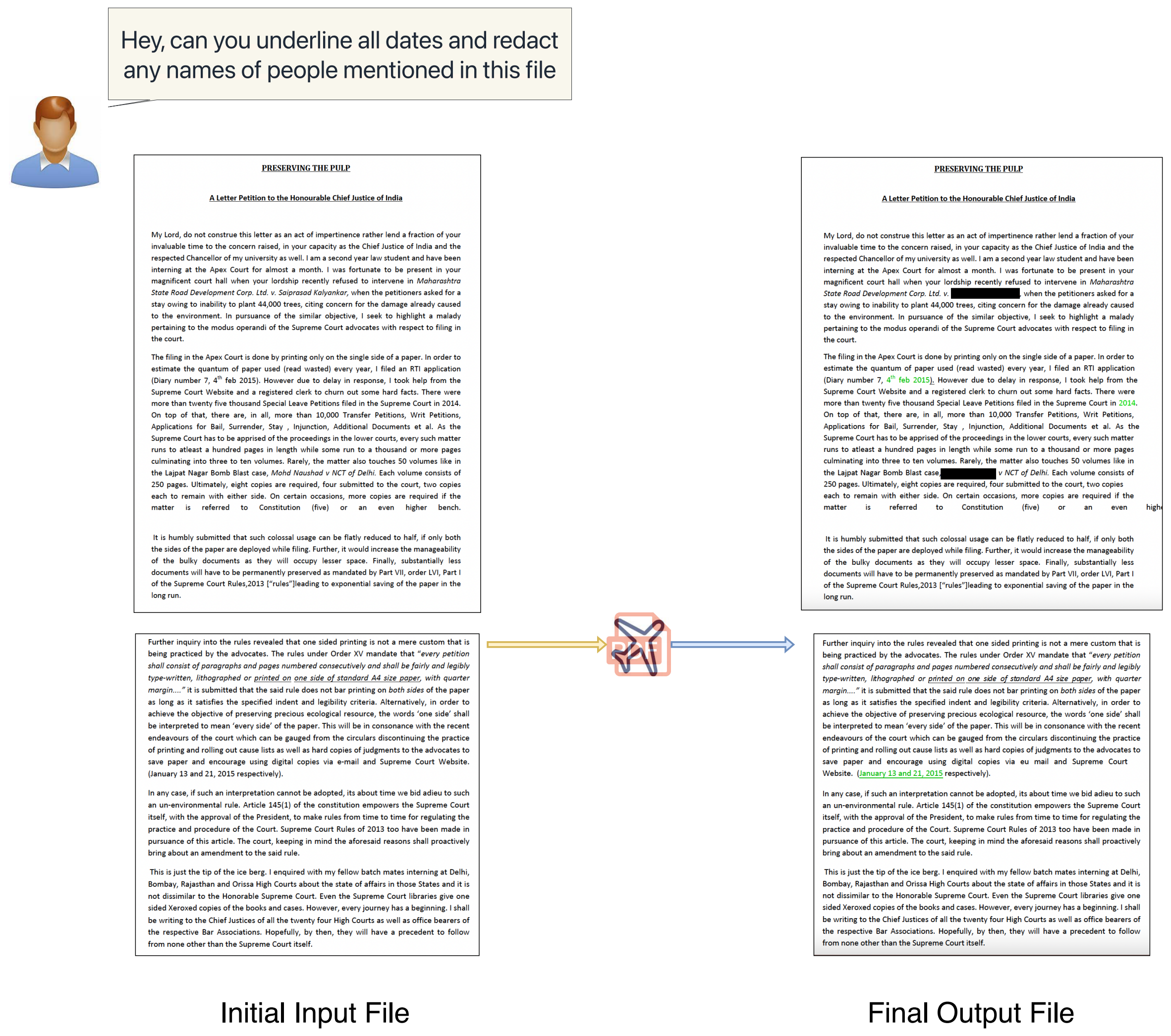
Please verify the information contained in this e-Visa using the "Status Enquiry" function in SAVE at https://eservices.ica.gov.sg/esvclandingpage/save .

Figure 8: Example of a visa document being edited using DocPilot. The user asks to "redact all mentions of Personally Identifiable Information in the document". DocPilot removes names, passport numbers, date of birth, sex, nationality, and dates in the input document.

complete a short survey form reflecting on their experience. Each of the tasks were repeated for every new workflow evaluated.

A.7.2 Measures

Our data collection included: 1) interaction logs during DocPilot app usage, 2) self-reported feedback after each workflow request, and 3) open-ended user feedback. The interaction logs included


Hey, can you underline all dates and redact any names of people mentioned in this file
Initial Input File
Final Output File

PRESERVING THE PULP

A Letter Petition to the Honourable Chief Justice of India

My Lord, do not construe this letter as an act of impertinence rather lend a fraction of your invaluable time to the concern raised, in your capacity as the Chief Justice of India and the respected Chancellor of my university as well. I am a second year law student and have been interning at the Apex Court for almost a month. I was fortunate to be present in your magnificent court hall when your lordship recently refused to intervene in *Maharashtra State Road Development Corp. Ltd. v. Saiprasad Kalyankar,* when the petitioners asked for a stay owing to inability to plant 44,000 trees, citing concern for the damage already caused to the environment. In pursuance of the similar objective, I seek to highlight a malady pertaining to the modus operandi of the Supreme Court advocates with respect to filing in the court.

The filing in the Apex Court is done by printing only on the single side of a paper. In order to estimate the quantum of paper used (read wasted) every year, I filed an RTI application (Diary number 7, 4th feb 2015). However due to delay in response, I took help from the Supreme Court Website and a registered clerk to churn out some hard facts. There were more than twenty five thousand Special Leave Petitions filed in the Supreme Court in 2014. On top of that, there are, in all, more than 10,000 Transfer Petitions, Writ Petitions, Applications for Bail, Surrender, Stay , Injunction, Additional Documents et al. As the Supreme Court has to be apprised of the proceedings in the lower courts, every such matter runs to atleast a hundred pages in length while some run to a thousand or more pages culminating into three to ten volumes. Rarely, the matter also touches 50 volumes like in the Lajpat Nagar Bomb Blast case, *Mohd Naushad v NCT of Delhi.* Each volume consists of 250 pages. Ultimately, eight copies are required, four submitted to the court, two copies each to remain with either side. On certain occasions, more copies are required if the matter is referred to Constitution (five) or an even higher bench.

It is humbly submitted that such colossal usage can be flatly reduced to half, if only both the sides of the paper are deployed while filing. Further, it would increase the manageability of the bulky documents as they will occupy lesser space. Finally, substantially less documents will have to be permanently preserved as mandated by Part VII, order LVI, Part I of the Supreme Court Rules,2013 [“rules”]leading to exponential saving of the paper in the long run.

Further inquiry into the rules revealed that one sided printing is not a mere custom that is being practiced by the advocates. The rules under Order XV mandate that *“every petition shall consist of paragraphs and pages numbered consecutively and shall be fairly and legibly type-written, lithographed or printed on one side of standard A4 size paper, with quarter margin….”* it is submitted that the said rule does not bar printing on *both sides* of the paper as long as it satisfies the specified indent and legibility criteria. Alternatively, in order to achieve the objective of preserving precious ecological resource, the words ‘one side’ shall be interpreted to mean ‘every side’ of the paper. This will be in consonance with the recent endeavours of the court which can be gauged from the circulars discontinuing the practice of printing and rolling out cause lists as well as hard copies of judgments to the advocates to save paper and encourage using digital copies via e-mail and Supreme Court Website. (January 13 and 21, 2015 respectively).

In any case, if such an interpretation cannot be adopted, its about time we bid adieu to such an un-environmental rule. Article 145(1) of the constitution empowers the Supreme Court itself, with the approval of the President, to make rules from time to time for regulating the practice and procedure of the Court. Supreme Court Rules of 2013 too have been made in pursuance of this article. The court, keeping in mind the aforesaid reasons shall proactively bring about an amendment to the said rule.

This is just the tip of the ice berg. I enquired with my fellow batch mates interning at Delhi, Bombay, Rajasthan and Orissa High Courts about the state of affairs in those States and it is not dissimilar to the Honorable Supreme Court. Even the Supreme Court libraries give one sided Xeroxed copies of the books and cases. However, every journey has a beginning. I shall be writing to the Chief Justices of all the twenty four High Courts as well as office bearers of the respective Bar Associations. Hopefully, by then, they will have a precedent to follow from none other than the Supreme Court itself.

PRESERVING THE PULP

A Letter Petition to the Honourable Chief Justice of India

My Lord, do not construe this letter as an act of impertinence rather lend a fraction of your invaluable time to the concern raised, in your capacity as the Chief Justice of India and the respected Chancellor of my university as well. I am a second year law student and have been interning at the Apex Court for almost a month. I was fortunate to be present in your magnificent court hall when your lordship recently refused to intervene in *Maharashtra State Road Development Corp. Ltd. v.* [redacted], when the petitioners asked for a stay owing to inability to plant 44,000 trees, citing concern for the damage already caused to the environment. In pursuance of the similar objective, I seek to highlight a malady pertaining to the modus operandi of the Supreme Court advocates with respect to filing in the court.

The filing in the Apex Court is done by printing only on the single side of a paper. In order to estimate the quantum of paper used (read wasted) every year, I filed an RTI application (Diary number 7, 4th feb 2015). However due to delay in response, I took help from the Supreme Court Website and a registered clerk to churn out some hard facts. There were more than twenty five thousand Special Leave Petitions filed in the Supreme Court in 2014. On top of that, there are, in all, more than 10,000 Transfer Petitions, Writ Petitions, Applications for Bail, Surrender, Stay , Injunction, Additional Documents et al. As the Supreme Court has to be apprised of the proceedings in the lower courts, every such matter runs to atleast a hundred pages in length while some run to a thousand or more pages culminating into three to ten volumes. Rarely, the matter also touches 50 volumes like in the Lajpat Nagar Bomb Blast case,[redacted] *v NCT of Delhi.* Each volume consists of 250 pages. Ultimately, eight copies are required, four submitted to the court, two copies each to remain with either side. On certain occasions, more copies are required if the matter is referred to Constitution (five) or an even high

It is humbly submitted that such colossal usage can be flatly reduced to half, if only both the sides of the paper are deployed while filing. Further, it would increase the manageability of the bulky documents as they will occupy lesser space. Finally, substantially less documents will have to be permanently preserved as mandated by Part VII, order LVI, Part I of the Supreme Court Rules,2013 [“rules”]leading to exponential saving of the paper in the long run.

Further inquiry into the rules revealed that one sided printing is not a mere custom that is being practiced by the advocates. The rules under Order XV mandate that *“every petition shall consist of paragraphs and pages numbered consecutively and shall be fairly and legibly type-written, lithographed or printed on one side of standard A4 size paper, with quarter margin….”* it is submitted that the said rule does not bar printing on *both sides* of the paper as long as it satisfies the specified indent and legibility criteria. Alternatively, in order to achieve the objective of preserving precious ecological resource, the words ‘one side’ shall be interpreted to mean ‘every side’ of the paper. This will be in consonance with the recent endeavours of the court which can be gauged from the circulars discontinuing the practice of printing and rolling out cause lists as well as hard copies of judgments to the advocates to save paper and encourage using digital copies via eu mail and Supreme Court Website. (January 13 and 21, 2015 respectively).

In any case, if such an interpretation cannot be adopted, its about time we bid adieu to such an un-environmental rule. Article 145(1) of the constitution empowers the Supreme Court itself, with the approval of the President, to make rules from time to time for regulating the practice and procedure of the Court. Supreme Court Rules of 2013 too have been made in pursuance of this article. The court, keeping in mind the aforesaid reasons shall proactively bring about an amendment to the said rule.

This is just the tip of the ice berg. I enquired with my fellow batch mates interning at Delhi, Bombay, Rajasthan and Orissa High Courts about the state of affairs in those States and it is not dissimilar to the Honorable Supreme Court. Even the Supreme Court libraries give one sided Xeroxed copies of the books and cases. However, every journey has a beginning. I shall be writing to the Chief Justices of all the twenty four High Courts as well as office bearers of the respective Bar Associations. Hopefully, by then, they will have a precedent to follow from none other than the Supreme Court itself.

Figure 9: Example of a legal court document being edited using DocPilot. The user asks, " Hey, can you underline all dates and redact any names of people mentioned in this file?". DocPilot covers names ("Saiprasad Kalyankar", "Mohd Naushad") and underlines dates ("4th Feb 2015", "2014", "January 13 and 21, 2015") in the input document.

the chat history, program execution actions, and resulting files after execution. The self-reported measures included overall workflow, satisfaction with the initial DocPilot suggested plan, satisfaction with DocPilot incorporating user feedback to the plan, and satisfaction with how well actions were executed in the resulting files.

A.7.3 Example Workflows

In total we collected data from 80 workflow requests. 16 workflows were user-provided based on the users' real world PDF workflows and 64 were workflows suggested to the user. We provided the suggested workflows to ensure workflows evaluated included variety in the types of actions used and in the number of actions requested. The user was encouraged to make small adjustments to the suggested workflows (as needed). For example if the workflow requested to delete page 5 of a document but the document only had 3 pages, then the user modified the workflow prompt accordingly.

Examples of **user-provided workflows**:

1. Add a watermark with the text "DRAFT" on every page, underline the test cycle types in the table, and extract the cleaning index values into a separate list
2. Highlight the text "ENERGY STAR Test Method for Determining Residential Dishwasher Cleaning Performance" in the document, convert page into image file, and add a header with the text "Energy Star Most Efficient 2016
3. Underline all section headings, redact the company's physical address, and add a watermark with text "Evaluation Copy" on each page

4. Extract pages 1-2 as a separate file with password "BUDGET2013", summarize the key issues discussed and action points, then add this summary to a new first page.

5. Extract all key terms and concepts, and create a glossary or index at the end of the document

Examples of **suggested workflows**:

1. Summarize all mentions of product launch dates and marketing strategies from the document, add a new page in front add this summary. Finally convert it to a Word file for later reference.

2. Redact all salary figures from the document, then add a line at the end stating the average salary of the listed positions. Underline the final mean salary figure for emphasis

3. Search for any mentions of project deadlines and add them as a new page at the end, then compress the file size to optimize storage space.

4. Search for any occurrences of the term 'Confidential' and redact them, after deleting pages 1-2. And add a watermark "Top Secret" to each remaining page.

5. Identify and highlight any technical or specialized terminology used within the document and add a signature to page 1 and protect the document with encryption.

A.7.4 Results: Code Iterations

Table 2 illustrates the number of code iterations for each workflow (n=80). The majority of workflows (48/80) required one code iteration, and most workflows were successful in a maximum of two LLM-based code revision cycles.

# of iterations	Count
0	11
1	48
2	13
3	1
5	2
6	3

Table 2: The number of code iterations for each workflow (n=80). The majority of workflows (48/80) required one code iteration.