m&m's: A Benchmark to Evaluate Tool-Use for *m*ulti-step *m*ulti-modal Tasks

Anonymous CVPR submission

Paper ID 16

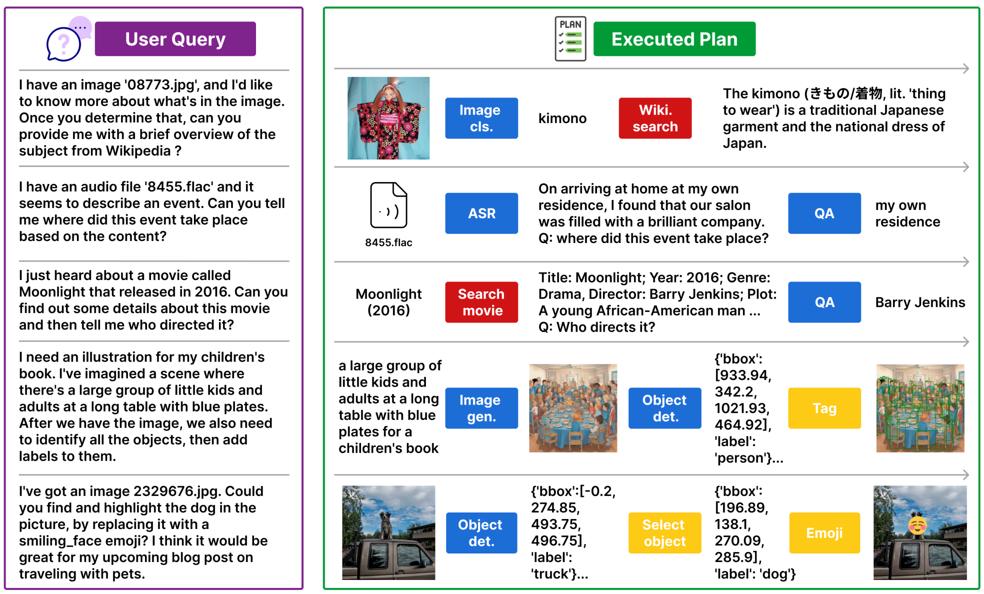

Figure 1. We present examples of query-plan pairs along with the execution results of the plans in *m&m*'s. Our benchmark contains a large quantity of diverse user queries involving three modalities (i.e. text, image, and audio) as well as human-verified plans that consist of 1 - 3 tools across three categories: multi-modal machine learning models (blue), public APIs (red) and image processing modules (yellow).

Abstract

Real-world multi-modal problems are rarely solved by a single machine learning model, and often require multi-step computational plans that involve stitching several models. Tool-augmented LLMs hold tremendous promise for automating the generation of such computational plans. However, the lack of standardized benchmarks for evaluating LLMs as planners for multi-step multi-modal tasks has prevented a systematic study of planner design decisions. Should LLMs generate a full plan in a single shot or step-by-step? Does feedback improve planning? To answer these questions and more, we introduce m&m's: a benchmark containing 4K+ multi-step multi-modal tasks involving 33 tools that include multi-modal models, (free) public APIs, and image processing modules. For each of these task queries, we provide automatically generated plans using this realistic toolset. We further provide a high-quality subset of 1,565 task plans that are human-verified and correctly executable. With m&m's, we evaluate 6 popular LLMs with 2 planning strategies (multi-step vs. step-by-step planning), 2 plan formats (JSON vs. code), and 3 types of feedback (parsing/verification/execution). Finally, we summarize takeaways from our extensive experiments.

1. Introduction

Planning agents—powered by large language models (LLMs)—are becoming increasingly proficient at decomposing user-specified tasks into a series of subtasks, where each subtask is executed by invoking tools (Figure 1). Given an LLM and toolset, the design space of planning agents is extremely rich, involving many decisions such as *planning strategy* (e.g. generation of the whole plan vs one step of the plan at a time), *forms of feedback* (e.g. parsing/verification/execution feedback), and *plan format* (e.g.

JSON strings that specify tools and arguments vs free-form Python code).

Unfortunately, there is no existing planning benchmark that supports evaluation along this combinatorially rich design space with a realistic set of multimodal tools. Recent concurrent benchmarks such as ToolEmu and TaskBench [20, 23] provide user queries and ground truth plans but lack support for realistic plan execution. For instance, TaskBench assumes that a list of tools is available for planning without providing actual implementation of the tools. ToolEmu likewise uses LLMs to emulate tool execution instead of providing tool implementations. The lack of actual implementations of tools and real execution feedback makes the study of the design space elucidated above unrealistic at best, if not impossible.

Motivated by this need for a standardized benchmark for studying the design space of multi-step multi-modal planning agents, we first propose the *m&m*'s benchmark, which contains 4K+ realistic user tasks and automatically generated task plans. Among these, 1565 are human-verified and executable with 33 curated tools consisting of multi-modal models, public APIs, and image processing modules. Next, we use *m&m*'s to systematically study the impact of 2 planning strategies (step-by-step and multi-step), 2 kinds of feedback (verification and execution), and 2 plan formats (JSON and code). Through extensive experimentation with 6 popular open-source and proprietary LLMs of varying sizes, we reveal three key findings: First, existing LLMs instructed to perform multi-step planning consistently outperform step-by-step planning, regardless of the model size. Second, feedback improves LLM's ability to predict the correct argument name for each tool and generate overall executable tool plans but doesn't necessarily improve the ability to choose the right tools. Third, most models perform comparably on tool prediction with JSON-format generation and Python code generation, but they all produce more executable plans with JSON-format generation than with code generation.

2. *m&m*'s: the benchmark

We curate the *m&m*'s benchmark to facilitate the study of LLM planners for *m*ulti-step *m*ulti-modal tasks.

2.1. Dataset generation

To create such a dataset, our data generation process consists of five major steps: tool graph sampling, input example sampling, query generation, plan generation, and finally human verification of generated query-plan pairs (Figure 2.1).

1 **Tool graph sampling.** We first create a directed graph with all 33 tools as the nodes and edges denoting valid connections between nodes. A connection is valid only if the output of the source tool matches the expected input type of the target tool. We then sample subgraphs from the full tool graph to obtain tool sequences with valid tool dependencies.

2 **Input example sampling.** Besides the tool sequences, we also need input examples to the tools to ground queries generation. To do so, we first collect real-world examples from 11 existing datasets, including ImageNet [4], SQUAD [18], Visual Genome [12], MagicBrush [41], librispeech [16]. Then, to pair a tool graph sampled in the previous step with an input, we randomly sample an input for the first tool in the graph. For example, if the first tool in a tool sequence is `image classification`, we randomly sample an image (e.g. "16611.jpg") from ImageNet as the input.

3 **Query generation.** With a set of tool sequences and input examples to the first tools, we prompt GPT-4 to generate realistic user queries. Concretely, we randomly sample 5 different input examples for each tool sequence and ask GPT-4 to generate 2 queries for each tool sequence with the same input (See Appendix for the full prompt).

4 **Plan generation.** For plan generation, we write a rule-based program to generate a plan for each query. Each step in the plan contains an id, tool name, and an argument dictionary with this tool's argument names as the keys and argument values as values. We populate each node's ID and name based on the sampled tool sequence and fill in the argument names for each tool using a pre-defined metadata document.

5 **Human verification** Finally, we perform extensive human verification on all 4k+ generated query-plan pairs. We ask three expert annotators (who are undergraduate and Ph.D. students in CS) to rate each query-plan pair with 0 or 1, where 1 indicates that the plan can resolve the query perfectly. We obtain a subset of 1.5k+ examples on which all three annotators rate 1 and perform further filtering to balance the overall distribution of tools (See the Appendix for more details).

2.2. Dataset quantity and quality

Overall, *m&m*'s contains a large quantity of **diverse ecologically-valid task** queries (see Figure 1). Each task is associated with **human-verified** and executable plans, where 1565 have been verified to be correct by three human annotators and 882 remain after additional filtering. Tasks are **granular in difficulty** with 70 queries that require a single tool, 159 need two tools, and 653 need three tools. See more examples and details in the Appendix.

3. Experiment

Using our benchmark with a flexible agent design, we experiment with 6 instruction-tuned large language models of varying sizes across different planning setups.

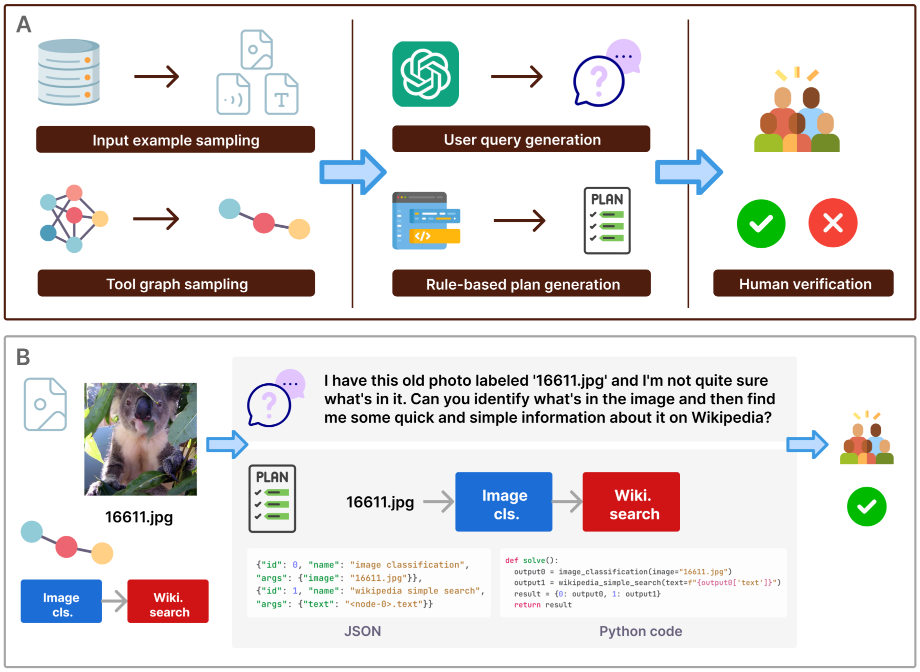

Figure 2. **Data generation pipeline.** As shown in panel A, our generation process consists of five main steps: input example sampling, tool graph sampling, user query generation with GPT-4, rule-based plan generation, and human verification. Panel B showcases an instantiation of this process with a real example.

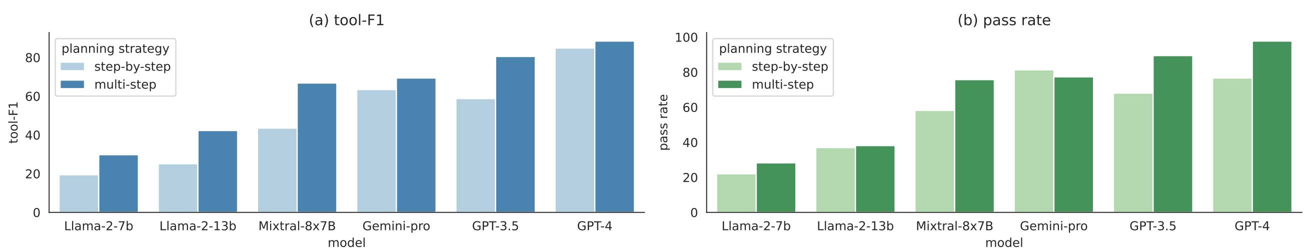

Figure 3. **Comparing planning strategies.** We find that models consistently perform better on tool-F1 under multi-step prediction compared to under step-by-step prediction regardless of their sizes. Similarly, all models except for Gemini-pro achieve a higher pass rate with multi-step prediction.

3.1. Setup

We establish a unified framework to categorize LLMs' task planning setups along the three axes below. **Planning strategy:** Prior works formulate task planning as either step-by-step or multi-step planning[17, 22, 39]. Step-by-step planning refers to the setup where a language model is instructed to predict only one action at a time (Figure 9 (1b)). On the other hand, in the setting of multi-step planning, a model can predict multiple actions at once (Figure 9 (1a)). **Plan format:** Additionally, existing works have also adopted different plan formats for tool use: often as code, pseudo-code, or predefined structured representations such as JSON [7, 22, 28]. In this work, we primarily focus on two of the common plan formats: JSON and code. **Feedback:** We experiment with three kinds of feedback - feedback from parsers, rule-based verifiers, and execution modules (Figure 13).

3.2. Evaluation metrics

To holistically evaluate planning agents' performance on our benchmark, we adopt three main metrics: **tool-F1**, **argname-F1**, and **pass rate**. **Tool-F1** is defined as the F1 score of tool name prediction, where we treat each predicted tool name as one example and compare the set of predicted tool names to the groundtruth set of tools in each plan. Similarly, **argname-F1** is defined as the F1 score of argument name prediction for each tool. **Pass rate** is the percentage

Table 1. We present the tool-F1 and argname-F1 of models with various feedback, where P, V, and E represent parsing, verification, and execution feedback respectively. We use parsing feedback only (P) under multi-step planning and JSON-format language generation as the basis, while showing the Δ of those with other feedback combinations compared to parsing feedback only.

	tool-F1				argname-F1				pass rate			
model	P	PV	PE	PVE	P	PV	PE	PVE	P	PV	PE	PVE
Llama-2-7b	29.78	-2.94	-2.59	-2.58	34.03	2.03	1.24	1.15	28.23	18.14	10.32	13.72
Llama-2-13b	42.27	-3.45	-2.78	-4.57	45.07	3.94	3.08	3.29	38.10	29.93	32.99	23.92
Mixtral-8x7B	66.79	1.18	-0.11	-0.04	72.52	2.00	1.89	2.72	75.74	10.32	8.96	10.77
Gemini-pro	69.38	1.18	-0.11	-0.04	73.37	2.00	1.89	2.72	77.32	13.27	14.06	16.67
GPT-3.5-turbo-0125	80.52	-0.65	-2.80	-2.56	84.86	0.65	-0.92	-0.86	89.46	6.69	7.26	6.92
GPT-4-0125-preview	88.46	-0.60	0.25	-0.91	89.81	-0.18	0.48	0.32	97.73	1.13	-1.25	2.15

Note: we use the experiments with parsing feedback instead of no feedback at all as the baseline to highlight external feedback's effects on tool selection and invocation instead of parsing. We include the results of experiments with no feedback in the Appendix.

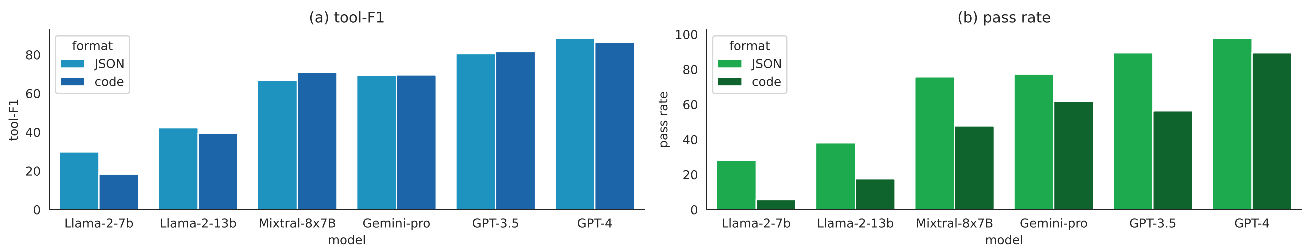

Figure 4. **Comparing plan formats.** We find that all models except for Llama-7-b perform comparably on tool-F1 with JSON-format and code generation. However, JSON-format generation leads to a much higher pass rate across all models.

of predictions that execute successfully without any execution errors.

We report additional metrics, including argvalue-F1, overall plan accuracy, normalized edit distance, and edge-F1 as well as code-specific metrics such as AST accuracy and CodeBLEU in the supplementary material.

3.3. Results

We highlight three key findings and describe each of them in detail:

1. Models consistently perform better on tool-F1 and pass rate under multi-step planning than under step-by-step planning. We find that all large language models achieve higher tool-F1 when they are instructed to perform multi-step planning compared to when they perform step-by-step prediction (Figure 3), and all models except for Gemini-pro achieve a higher pass rate with multi-step planning. Among the 6 models we evaluated, Llama-2-7b, Llama-2-13b, and GPT-3.5 all showcase a large increase (>10%) in performance with multi-step planning compared to step-by-step prediction, with the greatest increase of 21.8% for GPT-3.5.

2. External feedback can improve planning agents' performance on argument name prediction and pass rate. On the effects of external feedback, we find that both verification and execution feedback can lead to slightly better argname-F1 and much higher pass rates (Table 1), indicating that feedback can help models predict correct argument names and generate more executable plans. With feedback, most models can increase argname-F1 by around 1-4% and pass rate by up to 20-30% (Table 1). There are only a few exceptions on GPT-3.5 and GPT-4, which already obtain relatively high performance without feedback and experience around 1% drop in argname-F1 and/or pass rate with feedback (Table 1).

3. Models perform comparably on tool-F1 with JSON-format and code generation but much worse on pass rate with code generation. Our experiments show that while all models except for Llama-2-7b achieve similar tool-F1s (<3% difference) with JSON-format generation and code generation, they all suffer from a large drop in pass rate with code generation (Figure 4). These results suggest that JSON-format generation is preferable to code generation when the executability of generated plans matters.

Upon qualitative analysis, we find common errors that result in the findings above and present examples of these errors in the Appendix.

References

[1] Kelsey R Allen, Kevin A Smith, and Joshua B Tenenbaum. Rapid trial-and-error learning with simulation supports flexible tool use and physical reasoning. *Proceedings of the National Academy of Sciences*, 117(47):29302–29310, 2020. 7

[2] Po-Lin Chen and Cheng-Shang Chang. Interact: Exploring the potentials of chatgpt as a cooperative agent. *arXiv preprint arXiv:2308.01552*, 2023. 7

[3] Carlos G Correa, Mark K Ho, Frederick Callaway, Nathaniel D Daw, and Thomas L Griffiths. Humans decompose tasks by trading off utility and computational cost. *PLOS Computational Biology*, 19(6):e1011087, 2023. 7

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 2

[5] Difei Gao, Lei Ji, Luowei Zhou, Kevin Qinghong Lin, Joya Chen, Zihan Fan, and Mike Zheng Shou. Assistgpt: A general multi-modal assistant that can plan, execute, inspect, and learn. *arXiv preprint arXiv:2306.08640*, 2023. 7

[6] Madeleine Grunde-McLaughlin, Michelle S Lam, Ranjay Krishna, Daniel S Weld, and Jeffrey Heer. Designing llm chains by adapting techniques from crowdsourcing workflows. *arXiv preprint arXiv:2312.11681*, 2023. 7

[7] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training, 2022. 3, 6, 7

[8] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022. 7

[9] Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023. 7

[10] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6700–6709, 2019. 6

[11] Sahar Kazemzadeh, Vicente Ordonez, Mark Matten, and Tamara Berg. ReferItGame: Referring to objects in photographs of natural scenes. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 787–798, Doha, Qatar, 2014. Association for Computational Linguistics. 6

[12] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision*, 123:32–73, 2017. 2

[13] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024. 7

[14] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023. 6

[15] Ning Miao, Yee Whye Teh, and Tom Rainforth. Selfcheck: Using llms to zero-shot check their own step-by-step reasoning. *arXiv preprint arXiv:2308.00436*, 2023. 7

[16] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015. 2

[17] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. 3, 6, 7

[18] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, 2016. Association for Computational Linguistics. 2

[19] Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. Sayplan: Grounding large language models using 3d scene graphs for scalable task planning. *arXiv preprint arXiv:2307.06135*, 2023. 7

[20] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the risks of lm agents with an lm-emulated sandbox, 2023. 2, 6, 7

[21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024. 6

[22] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023. 3, 6, 7

[23] Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: Benchmarking large language models for task automation. *arXiv preprint arXiv:2311.18760*, 2023. 2, 6, 7, 9, 15

[24] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023. 7

[25] Otilia Stretcu, Edward Vendrow, Kenji Hata, Krishnamurthy Viswanathan, Vittorio Ferrari, Sasan Tavakkol, Wenlei Zhou, Aditya Avinash, Emming Luo, Neil Gordon Alldrin, Ranjay Krishna, and Ariel Fuxman. Agile modeling: From concept to classifier in minutes. In *Proceedings of the IEEE/CVF*

International Conference on Computer Vision, pages 22323–22334, 2023. 7

[26] Alane Suhr, Mike Lewis, James Yeh, and Yoav Artzi. A corpus of natural language for visual reasoning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 217–223, Vancouver, Canada, 2017. Association for Computational Linguistics. 6

[27] Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems*, 36, 2024. 7

[28] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023. 3, 6, 7

[29] Imad Eddine Toubal, Aditya Avinash, Neil Gordon Alldrin, Jan Dlabal, Wenlei Zhou, Enming Luo, Otilia Stretcu, Hao Xiong, Chun-Ta Lu, Howard Zhou, Ranjay Krishna, Ariel Fuxman, and Tom Duerig. Modeling collaborator: Enabling subjective vision classification with minimal human effort via llm tool-use. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2024. 7

[30] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. 7

[31] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023. 7

[32] 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 preprint arXiv:2309.10691*, 2023. 7

[33] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. 7

[34] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023. 12

[35] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training, 2023. 14

[36] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. 6

[37] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022. 6

[38] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. 7

[39] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. 3, 7, 12

[40] Jieyu Zhang, Ranjay Krishna, Ahmed H Awadallah, and Chi Wang. Ecoassistant: Using llm assistant more affordably and accurately. *arXiv preprint arXiv:2310.03046*, 2023. 7

[41] Kai Zhang, Lingbo Mo, Wenhu Chen, Huan Sun, and Yu Su. Magicbrush: A manually annotated dataset for instruction-guided image editing. *Advances in Neural Information Processing Systems*, 36, 2024. 2

[42] Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 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*, 2023. 7

4. Supplementary

A. Related work

We situate our work amongst tool-use research.

Planning evaluations. Although many tool-use variants have been proposed, evaluating LLMs on tool-use still lacks a standardized protocol. For instance, VisProg and ViperGPT evaluate their plan's *executions* on vision tasks using a Python-like *code* format [7, 28]. HuggingGPT evaluates only the *plan* accuracy (did the agent choose the right tools) without executing the proposed plans [22]. ToolFormer [21] and ToolLLaMA [17] both use *natural language* instead of *code* to interface with tools; while ToolFormer generates a *multi-step* plan all at once and evaluates the program's *execution*, ToolLLaMA generates the plan *step-by-step*, with *self-feedback* to correct mistakes. ToolLLaMA evaluates only the *plans* while ToolFormer evaluates both *plans* and executions. Unfortunately, no single benchmark evaluates planning agents along this combinatorial design space, which is what we contribute.

Tool-use benchmarks. Today, tool-use evaluation is spread out across a number of diverse benchmarks, including HotpotQA, WebShop, GQA, RefCOCO, and NLVR [10, 11, 26, 36, 37]. None of these contains ground truth plans, conflating planning errors with execution error. In other words, it is hard to separate whether an LLM failed to propose the correct plan or whether one of the tools used in the plan failed. In response, recent concurrent efforts have proposed new benchmarks, such as ToolEmu, TaskBench, and GAIA [14, 20, 23]. They do contain ground truth plans but fail to support evaluating plans' execution results (Table 2).

Table 2. Compared to previous tool planning benchmarks, *m&m*'s contains multimodal queries that are more realistic and executable. *: MetaTool only considers Open AI plugins as tools. #: The queries of TaskBench contain textural placeholder of other modality data such as images, while queries of *m&m*'s come with real images.

		ToolBench [17]	ToolEmu [20]	TaskBench [23]	MetaTool [9]	*m&m*'s (ours)
Query	Real multi-modal inputs?	✗	✗	✗#	✗	✓
	Verified by human?	✗	✓	✓	✓	✓
Tools	Are all tools executable?	✓	✗	✗	✓	✓
	Multi-modal models	✗	✗	✓	*	✓
Plan	Format	JSON	JSON	JSON	JSON	JSON/Code
Scale	Number of unique tools	3,451	36	103	390	33
	Number of queries	126k	144	17K	20k	1.5k

Table 3. We list all 33 tools across three categories - ML models, public APIs, and image processing modules - in *m&m*'s.

Tool category	Tool name
ML model	text generation, text summarization, text classification, question answering, optical character recognition, image generation, image editing, image captioning, image classification, image segmentation, object detection, visual question answering, automatic speech recognition
Public APIs	get weather, get location, get math fact, get trivia fact, get year fact, get date fact, search movie, love calculator, wikipedia simple search
Image processing	image crop, image crop top, image crop bottom, image crop left, image crop right, select object, count, tag, color pop, emoji, background blur

Planning strategies. There are multiple strategies for planning. For instance, Psychology literature reveals that people rarely plan tasks in their entirety due to the cognitive cost of planning long-range tasks [3]. Instead, they plan the first couple of subtasks, and execute them before planning the rest [1, 3]. In the tool-use literature, we identify two primary forms of planning strategies: *step-by-step planning* [5, 17, 39] and *multi-step planning* [7, 22, 28]. Similar to people, step-by-step planning generates plans sequentially with one subtask at a time. By contrast, multi-step planning creates the entire plan before executing any subtask. Unfortunately, these two strategies have not been systematically compared; we systematically compare both across multiple open-source and close-source LLMs.

Feedback mechanisms. LLM planners make mistakes, stitching together tools that fail to execute or worse, fail to compile. Although human feedback is one mechanism to align plans with human expectations and preferences [2, 32], they require real users, making evaluation stochastic. However, there have been several automatic mechanisms that can improve plans [31, 40]. For instance, syntactic mistakes can easily be detected using external *verifiers* and can guide planners to iterate on their plans [8, 13, 15, 24]. Others require examining the output of individual subtask *executions* [19, 27, 30, 39, 42]. In this work, we compare plan parsing/verification feedback as well as tool execution feedback.

B. Limitations

There are a few limitations to our benchmark and evaluation. First, *m&m*'s only considers sequential task plans, which represent a majority of real-world user requests. However, some tasks might require dynamic task plans depending on the output for one subtask [6]. Dynamic plans require a more complex tool graph sampling procedure. Second, as our main goal is to study the effects of different planning formulations and types of feedback, we do not investigate another dimension of planning design: prompt style. We use direct and ReACT-style [39] prompting and exclude more sophisticated prompting strategies such as tree-of-thoughts prompting [33, 38]. Third, a few tools in our benchmark are generative, which makes the evaluation of the actual execution results subjective (See Appendix) [25, 29].

C. Additional data

We present more examples of query-plan pairs of *m&m*'s in Figure 5, and a complete list of all 33 tools in Table 3.

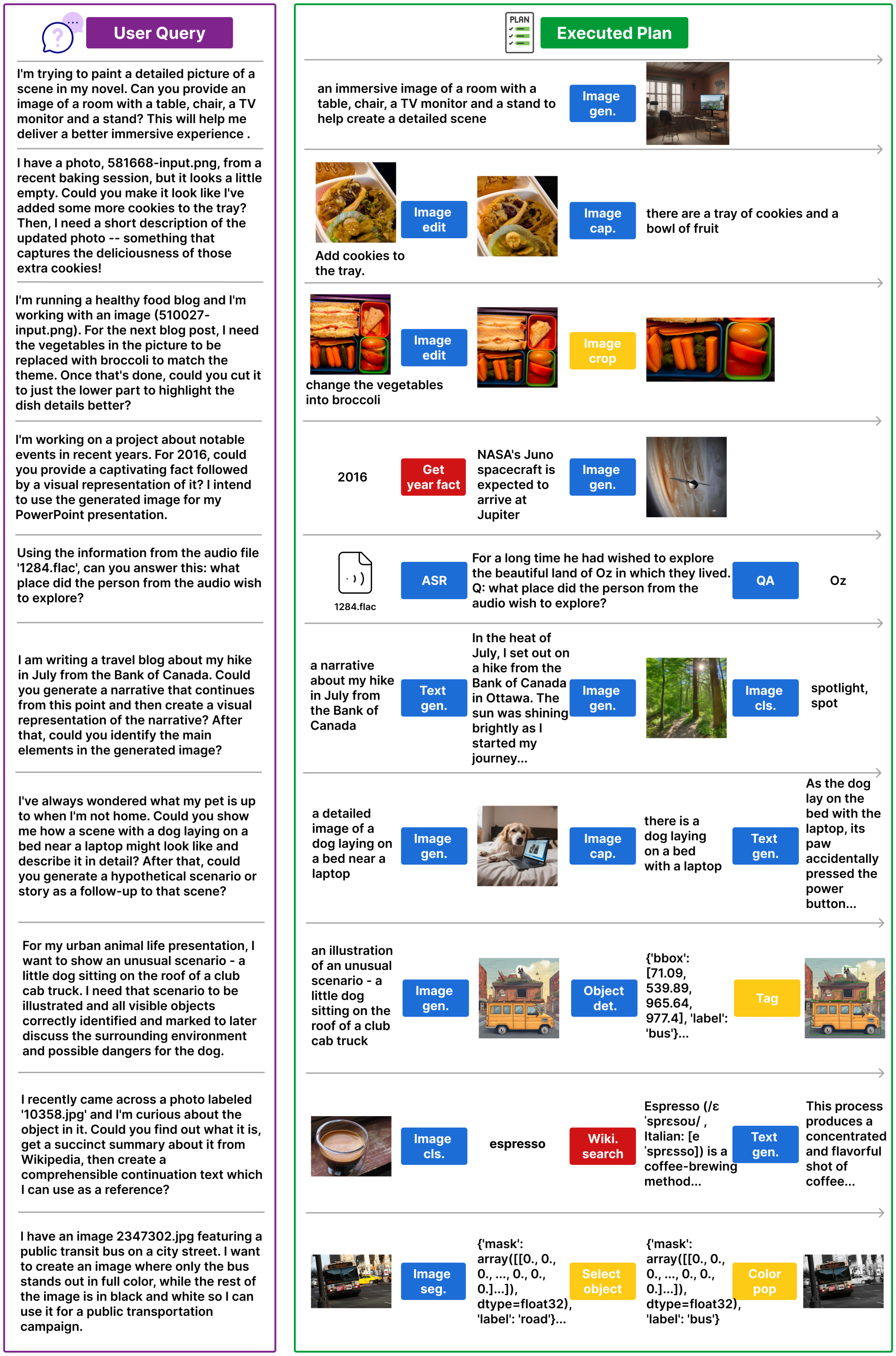

Figure 5. We present additional examples of query-plan pairs along with the execution results of the plans in *m&m*'s.

I have these tools:
image classification: It takes an image and classifies the subject in the image into a category such as cat or dog.
wikipedia simple search: Perform a basic search query on Wikipedia to retrieve a summary of the most relevant page.

Can you write 2 example queries for tasks I can do with a combined workflow of image classification, followed by wikipedia simple search?

There are a few requirements:
1) Each task query should sound natural, represent a realistic use case, and should NOT mention image classification, wikipedia simple search.
2) Each query should be based on these inputs to image classification: {'image': '16611.jpg'} and should explicitly mention these inputs.

Figure 6. **Query generation prompt.** We present the full prompt used for query generation.

Table 4. We present the tool-F1, argname-F1 and pass rate of models with various feedback, where P, V, and E represent parsing, verification, and execution feedback respectively. We use no feedback only (N/A) under multi-step planning and JSON-format language generation as the basis, while showing the Δ of those with other feedback combinations compared to no feedback.

	tool-F1					argname-F1					pass rate				
model	N/A	P	PV	PE	PVE	N/A	P	PV	PE	PVE	N/A	P	PV	PE	PVE
Llama-2-7b	27.37	2.41	-0.53	-0.18	-0.18	30.71	3.31	5.34	4.56	4.47	24.83	3.40	21.54	13.72	17.12
Llama-2-13b	40.30	1.97	-1.48	-0.80	-2.60	43.30	1.77	5.72	4.86	5.06	37.30	0.79	30.73	33.79	24.72
Mixtral-8x7B	65.06	1.73	0.88	0.15	2.75	73.00	-0.49	1.12	-0.14	0.85	69.61	6.12	16.44	15.08	16.89
Gemini-pro	68.57	0.80	1.98	0.69	0.76	72.79	0.58	2.58	2.47	3.30	73.92	3.40	16.67	17.46	20.07
GPT-3.5-turbo-0125	79.83	0.68	0.03	-2.11	-1.88	83.94	0.92	1.57	0.00	0.06	88.44	1.02	7.71	8.28	7.94
GPT-4-0125-preview	88.96	-0.50	-1.10	-0.26	-1.42	89.88	-0.07	-0.25	0.41	0.25	97.39	0.34	1.47	-0.91	2.49

D. Dataset generation

It is worth noting that two of the steps in our dataset generation pipeline draw similarities with the recently released concurrent TaskBench [23]. Similar to them, we also sample a subgraph of tools and query generation steps. However, we want to highlight two major differences: first, we leverage real-world examples as inputs to the tool sequences (in contrast to TaskBench's "example.jpg", "example.wav" etc.), which not only leads to a more realistic instantiation of queries but also enables plan execution on actual input which is crucial for studying the role of feedback in planning agents. Second, we use a rule-based program instead of GPT-4 to obtain the ground truth plans based on the sampled tool sequences, which eliminates the possibility of hallucinated and incorrect plans.

Below, we provide additional details about our dataset generation:

D.1. Prompts

We generate the queries with the prompt in Figure 6, and rewrite the argument values of `text generation` and `image generation` with the prompt shown in Figure 7.

D.2. Human verification statistics

The pairwise agreement rates among the 3 annotators are 74.95%, 81.43%, 70.88%, and the average pairwise agreement rate is 75.75% (std=4.34%).

D.3. Data filtering

We perform two types of data filtering on the 1565 human-verified examples: (1) we manually filter out 349 examples with poor execution results, especially those where intermediate tools return wrong or empty outputs (e.g. when `question answering` is the second tool in the sequence and outputs an empty string); (2) we filter out a total of 334 examples whose plans involve `image generation` and have more than 4 unique queries. We perform the second filtering step because of two reasons. First, the frequency of the tools initially follows the distribution in Figure 8 (blue), where `image generation` has a much higher count – 918 – than other tools. Thus, we would like to reduce the frequency of `image generation` in the dataset while maintaining the frequency of rare tools. To achieve this while also preserving the diversity of tool plans, we choose to filter out examples whose plans have 5-10 unique queries, as the average number of unique requests per tool plan before filtering is 4.20. We end up filtering out 40% (or 349) of these examples. After these two filtering steps, we are left with 882 examples in total that follow the

```
# INSTRUCTION #:
A tool node is defined as a dictionary with keys “id” storing its unique identifier, “name” specifying the model to
call, and “args” specifying the arguments needed to make an inference
 call to this tool.
Your task is to rewrite ONLY the 'text' values in the tool nodes 'text generation' and 'image generation' based on
the user request so that they are more concrete and aligned with user'
s intentions.

Below are a few examples:
# EXAMPLES #:
Request: I'm creating an educational video about the world's fastest vehicles and I need material on
watercrafts. Could you provide me with a thorough explanation and some engaging fact
s on What's The Fastest Boat Ever Made?
Nodes: [{'id': 0, 'name': 'text generation', 'args': {'text': "What's The Fastest Boat Ever Made?"}}]
New nodes: [{'id': 0, 'name': 'text generation', 'args': {'text': " a thorough explanation and some engaging facts on
"What's The Fastest Boat Ever Made?"}}]

Request: I would like to create a dynamic visual for my blog post about baseball. The text description I have is
'There is a baseball player who swung for the ball'. Could we use that t
o come up with something eye-catching and fitting for the topic?
Nodes: [{'id': 0, 'name': 'image generation', 'args': {'text': 'There is a baseball player who swung for the ball'}}]
New nodes: [{'id': 0, 'name': 'image generation', 'args': {'text': 'a dynamic and eye-catching image of a baseball
player who swung for the ball'}}]

Request: For a blog topic heading 'What Really Happens When You Flush on an Airplane?', I'm trying to explain
the process visually to my readers. Could you first generate a comprehensiv
e, easy-to-understand description of the process, and then create an illustrative image based on that
description?
Nodes: [{'id': 0, 'name': 'text generation', 'args': {'text': 'What Really Happens When You Flush on an Airplane?'}},
{'id': 1, 'name': 'image generation', 'args': {'text': '<node-0>.te
xt'}}]
New nodes: [{'id': 0, 'name': 'text generation', 'args': {'text': 'a comprehensive, easy-to-understand description of
What Really Happens When You Flush on an Airplane?'}}, {'id': 1, 'n
ame': 'image generation', 'args': {'text': 'an illustrative image based on <node-0>.text'}}]

# REQUIREMENTS #:
1) Besides the argument values of 'text generation' and 'image generation', everything else (including the
nodes' ids and names) must stay the same;
2) The argument value can include reference to last node i's text output as <node-i>.text.
3) You must NOT add or remove any nodes.

Request: "I need to give a quick presentation for kindergarteners on 'Why is the sky blue?'. I don't really have
time to sift through lots of complex information and I need simple, straightforward explanations with a relevant
image that kids can understand. Can you assist me with that?"
Nodes: [{'id': 0, 'name': 'wikipedia simple search', 'args': {'text': 'Why is the sky blue'}}, {'id': 1, 'name': 'text
summarization', 'args': {'text': '<node-0>.text'}}, {'id': 2, 'nam
e': 'image generation', 'args': {'text': '<node-1>.text'}}]
New nodes:
```

Figure 7. **Argument value rewrite prompt.** We present the full prompt used for rewriting the argument values of `text generation` and `image generation`.

distribution in Figure 8 (red).

D.4. Alternative plans

In addition to the one human verified groundtruth plan, we have also generated alternative plans to supplement our evaluation. Concretely, we generate these alternative plans

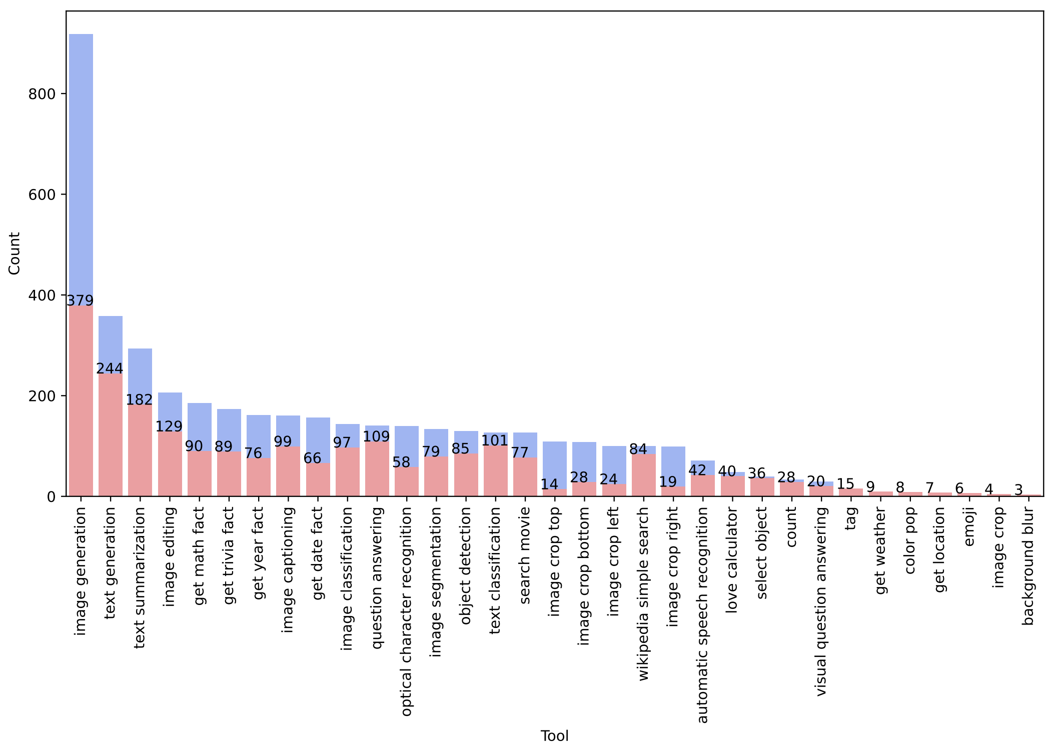

Figure 8. **Tool distribution before and after filtering.**

Table 5. **argvalue-F1.** We present the argvalue-F1 of step-by-step and multi-step planning with JSON-format generation and different types of feedback.

		argvalue-F1			
model	strategy	P	PV	PE	PVE
Llama-2-7b	step-by-step	4.63	8.28	9.68	9.57
	multi-step	10.34	9.88	9.47	10.57
Llama-2-13b	step-by-step	7.10	11.30	12.59	12.64
	multi-step	15.39	17.11	15.84	16.71
Mixtral-8x7B	step-by-step	20.44	24.32	21.77	21.69
	multi-step	36.45	36.70	35.70	36.73
Gemini-pro	step-by-step	32.28	27.81	32.22	31.37
	multi-step	37.22	39.89	36.30	38.33
GPT-3.5-turbo-0125	step-by-step	29.58	28.32	23.61	23.24
	multi-step	45.64	46.54	45.15	45.56
GPT-4-0125-preview	step-by-step	47.37	46.91	34.49	34.84
	multi-step	51.02	51.08	51.70	51.99

in three steps: first, we generate a set of syntactically valid (i.e. the alternative tool's input and output types are correct) and semantically valid (i.e. the alternative tool performs the same functionality as the original tool) alternative tools for each tool in our toolset; second, we manually verify their validity and only keep the human-verified valid tools in the alternative tools set; finally, we compose all valid tools at each position in the plan to obtain all combinations as the total set of valid plans. To generate the syntactically valid tools, we create a graph with both data (including input and output) and tools as nodes, and we obtain the syntactic alternative tools t_o^{alt} of the orig-

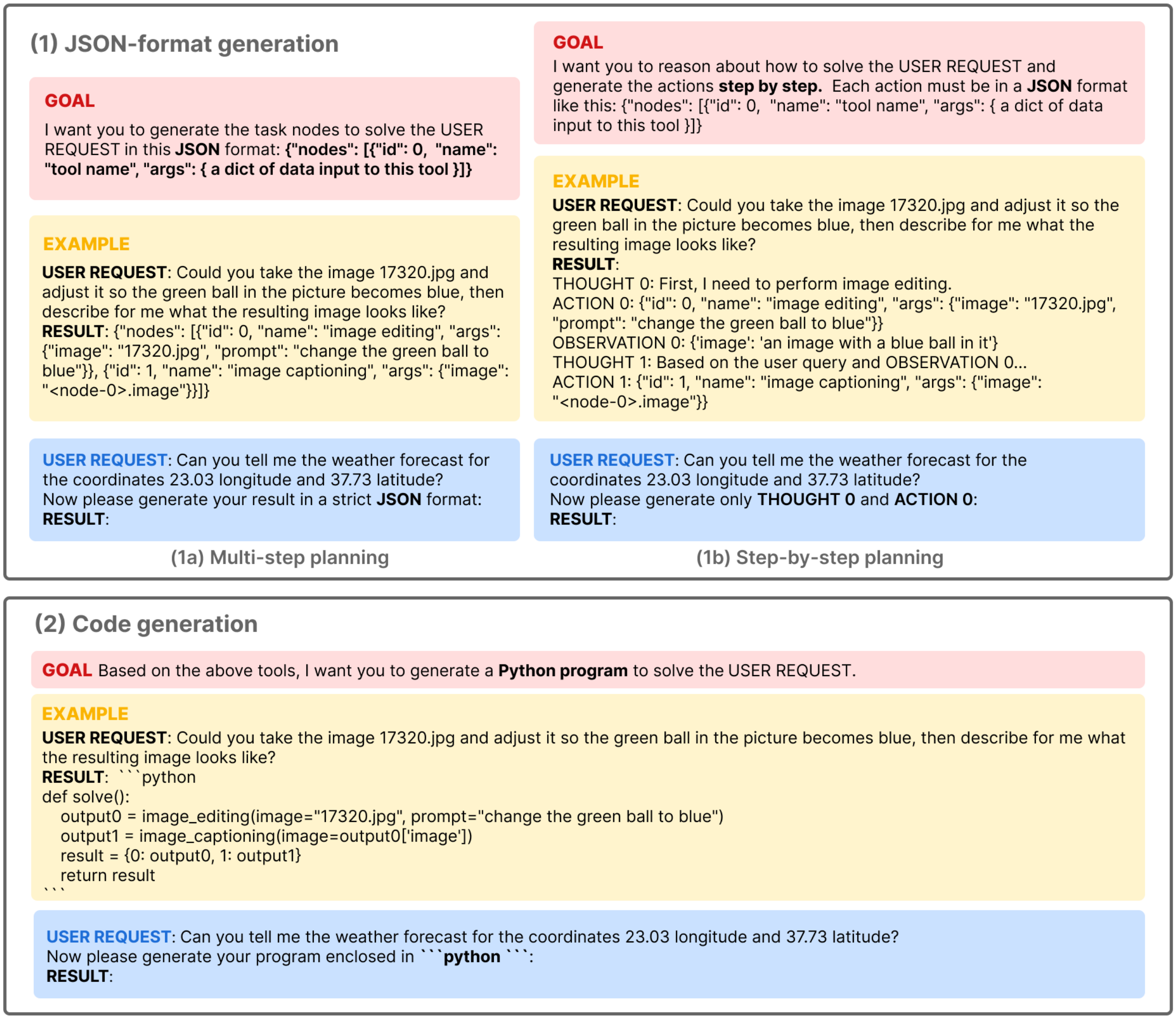

Figure 9. **Illustrating the three main planning setups in our evaluation:** (1a) multi-step and (1b) step-by-step JSON-format language generation [39], and (2) code generation. (Note that the prompts have been simplified for illustration. Please see the Appendix for the full prompts).

inal tool t_o by searching for all possible paths from t_o's input to its output. As for semantic alternative tools, we prompt GPT-4 to generate these for each tool in the toolset. For example, for the plan `image classification` $\rightarrow$ `text generation`, we first obtain alternative tools to each of them. For `image classification`, its syntactic alternative tools include `image captioning` and `visual question answering` as these tools' inputs both include one image and their outputs are a text – the same as `image classification`'s. In addition, GPT-4 identifies `object detection` as a semantic alternative to `image classification`. On the other hand, there are no human-verified alternative tools to `text generation`. Therefore, there are a total of 3 alternative plans to `image classification` $\rightarrow$ `text generation`.

E. Planning agent

To systematically evaluate the design space of planning agents, we design a modular planning system with these components: planning LLM, parser, verifier, and executor. We implement this system with AutoGen's framework [34]. Given the user query, the LLM must iteratively generate and refine the plan. Each iteration involves generating

Table 6. **edge-F1.** We present the edge-F1 of step-by-step and multi-step planning with JSON-format generation and different types of feedback.

		edge-F1			
model	strategy	P	PV	PE	PVE
Llama-2-7b	step-by-step	1.61	2.35	3.98	3.37
	multi-step	12.44	11.61	12.10	11.27
Llama-2-13b	step-by-step	5.74	6.22	6.96	8.22
	multi-step	23.27	23.98	24.00	23.58
Mixtral-8x7B	step-by-step	15.41	21.88	24.00	24.77
	multi-step	55.72	53.10	53.08	53.52
Gemini-pro	step-by-step	41.39	17.86	45.82	45.08
	multi-step	54.98	56.63	53.60	55.22
GPT-3.5-turbo-0125	step-by-step	31.37	27.23	39.40	39.72
	multi-step	69.52	71.03	67.98	69.05
GPT-4-0125-preview	step-by-step	73.68	72.67	68.28	68.12
	multi-step	78.80	78.79	79.47	79.60

Table 7. **Normalized edit distance.** We present the normalized edit distance of step-by-step and multi-step planning with JSON-format generation and different types of feedback.

		Normalized edit distance ↓			
model	strategy	P	PV	PE	PVE
Llama-2-7b	step-by-step	80.39	75.24	76.00	74.55
	multi-step	61.14	64.43	62.82	63.12
Llama-2-13b	step-by-step	72.81	68.57	68.60	67.84
	multi-step	47.57	48.69	49.63	49.73
Mixtral-8x7B	step-by-step	60.81	56.28	56.86	56.78
	multi-step	23.97	25.97	26.64	26.26
Gemini-pro	step-by-step	36.23	47.89	34.70	36.00
	multi-step	28.18	27.34	25.96	24.77
GPT-3.5-turbo-0125	step-by-step	51.46	52.38	47.93	47.44
	multi-step	16.08	15.55	17.44	17.86
GPT-4-0125-preview	step-by-step	14.26	14.70	16.92	16.62
	multi-step	10.96	11.39	10.59	10.81

the whole or a part of the plan and receiving feedback on the generation. Given the raw text output from the LLM planner at the current iteration, *m&m*'s supports the following 3 kinds of feedback (Figure 13):

Parsing feedback. The parser attempts to parse the LLM text output to either JSON or code formats and returns an error message in case of parsing failures.

Plan verification feedback. The verifier checks the parsed output according to pre-defined rules and returns an error message in case of rule violations. Specifically, the verifier checks if the predicted tool exists in our provided tool list, if it forms a valid connection with the previous tool, and if the predicted argument names match the ones specified in the metadata document.

Plan execution feedback. In the case of JSON output, the executor calls the functions with specified arguments in a Python environment and returns the output or execution errors. In the case of code output, the code is directly executed with outputs or errors returned as feedback.

We provide concrete examples of the parsing, verification and Execution feedback in Figure 13.

Table 8. **Plan accuracy**

Plan accuracy		(tool)				(tool+argname)			
model	strategy	P	PV	PE	PVE	P	PV	PE	PVE
Llama-2-7b	step-by-step	1.13	2.27	3.29	3.29	1.13	2.27	3.29	3.29
	multi-step	4.20	3.40	2.95	4.20	2.95	3.29	2.04	3.51
Llama-2-13b	step-by-step	1.25	3.17	3.74	4.99	1.13	3.17	3.74	4.99
	multi-step	11.90	13.83	10.88	12.13	9.52	13.27	9.98	11.79
Mixtral-8x7B	step-by-step	9.41	14.63	14.06	14.97	9.41	14.63	14.06	14.97
	multi-step	45.80	45.12	45.12	45.35	45.12	45.01	44.90	45.24
Gemini-pro	step-by-step	24.83	10.66	30.27	28.57	24.38	10.66	30.16	28.57
	multi-step	41.84	42.18	40.70	42.40	40.48	42.18	40.59	42.40
GPT-3.5-turbo-0125	step-by-step	19.27	14.97	18.59	19.16	19.27	14.97	18.59	19.16
	multi-step	59.64	60.20	57.48	58.39	59.52	60.20	57.48	58.39
GPT-4-0125-preview	step-by-step	61.68	60.88	51.93	53.17	61.68	60.88	51.93	53.17
	multi-step	70.63	69.50	71.43	70.63	70.63	69.50	71.43	70.63

Table 9. **Δ in plan accuracy considering alternative plans.**

Δ in plan accuracy		(tool)				(tool+argname)			
model	strategy	P	PV	PE	PVE	P	PV	PE	PVE
Llama-2-7b	step-by-step	0.00	0.11	0.11	0.11	0.00	0.11	0.11	0.11
	multi-step	0.79	0.34	0.68	0.57	0.00	0.11	0.11	0.23
Llama-2-13b	step-by-step	0.57	0.57	0.68	0.91	0.45	0.57	0.68	0.91
	multi-step	1.36	1.47	1.47	1.47	0.91	1.36	1.25	1.25
Mixtral-8x7B	step-by-step	0.79	2.15	1.93	2.04	0.79	1.93	1.93	1.93
	multi-step	4.08	3.40	3.74	2.83	3.40	3.40	3.29	2.61
Gemini-pro	step-by-step	1.36	2.83	2.49	1.93	1.36	2.83	2.38	1.93
	multi-step	3.74	2.83	4.65	3.51	3.40	2.83	4.65	3.51
GPT-3.5-turbo-0125	step-by-step	1.02	0.34	1.02	0.68	1.02	0.34	1.02	0.68
	multi-step	3.17	3.06	3.40	3.74	3.17	3.06	3.40	3.74
GPT-4-0125-preview	step-by-step	2.15	1.81	2.95	3.06	2.15	1.81	2.95	3.06
	multi-step	1.81	1.81	1.59	1.59	1.81	1.81	1.59	1.59

We present a side-by-side comparison of (Figure 9) as well as the full prompts used for multi-step JSON-format planning (Figure 10), step-by-step JSON-format planning (Figure 11, excluding details in the TOOL LIST which are the same as the ones in Figure 10) as well as code generation (Figure 12).

F. Qualitative analysis

Through qualitative analysis, we find out the common errors that lead to the findings. First, regarding the performance drop from multi-step to step-by-step planning, we find that, when models are instructed to perform step-by-step prediction, they tend to output "TERMINATE" after they receive positive feedback (e.g. "Parsing/verification/execution succeeded") from the environment, disregarding whether the user request has been fulfilled. This means that they often predict fewer steps than required and miss necessary tools to resolve the requests. (Figure 16 A) As for the mixed and even negative effects of feedback, we learn that this is because models can change some correct tools to the wrong ones or remove them even though the feedback instructs them to only fix the erroneous parts in the plan (Figure 16 B). One way to mitigate this error can be using more fine-grained and localized feedback [35]. Additionally, neither verification feedback nor execution feedback provides useful information on the correctness of the tool selection and increases their performance on tool-F1.

Last but not least, when it comes to code generation vs. json-format generation, we find that one common execution error in code generation is failing to access the output from a tool (Figure 16 C), which can be due to missing the output or accessing the output differently from what the instruction specifies and the tool implementation expects. While the same error also happens to JSON-format generation, it occurs less frequently due to the more rigid structure of JSON.

Table 10. **Plan accuracy considering argument values**

Plan accuracy (tool+argname+argvalue)		exact matching				entailment			
model	strategy	P	PV	PE	PVE	P	PV	PE	PVE
Llama-2-7b	step-by-step	0.57	1.02	1.81	1.59	0.91	1.81	2.95	2.38
	multi-step	0.57	0.34	0.23	0.57	1.02	1.59	0.68	1.59
Llama-2-13b	step-by-step	0.57	1.70	2.04	2.27	0.91	2.49	2.83	3.51
	multi-step	2.04	2.72	2.38	2.49	5.44	7.48	5.78	6.24
Mixtral-8x7B	step-by-step	2.72	5.44	3.51	3.51	6.12	9.86	7.03	7.37
	multi-step	9.75	10.09	9.52	10.77	28.00	29.14	28.68	29.48
Gemini-pro	step-by-step	7.03	5.78	7.48	6.58	15.42	9.52	17.12	15.19
	multi-step	8.39	11.34	9.07	11.45	24.15	27.89	24.83	27.66
GPT-3.5-turbo-0125	step-by-step	6.46	5.33	2.38	2.72	12.93	10.20	7.14	8.05
	multi-step	13.61	14.29	13.61	14.06	34.81	36.85	34.92	35.83
GPT-4-0125-preview	step-by-step	11.68	11.00	6.35	6.24	34.35	32.65	19.73	20.29
	multi-step	14.85	14.97	15.19	15.53	41.04	40.70	43.20	42.97

Table 11. **Code-specific metrics.** We present the AST accuracy and CodeBLEU score of models under multi-step planning with code generation with or without feedback.

	AST accuracy				CodeBLEU			
model	P	PV	PE	PVE	P	PV	PE	PVE
Llama-2-7b	0.00	0.00	0.00	0.00	22.64	21.28	17.58	21.19
Llama-2-13b	0.11	0.23	0.00	0.00	29.96	27.09	20.29	27.62
Mixtral-8x7B	2.04	3.06	4.22	2.30	54.17	48.48	53.01	47.21
Gemini-pro	3.85	5.33	3.74	4.54	62.37	61.13	59.00	59.18
GPT-3.5-turbo-0125	3.29	4.76	3.29	4.42	60.79	60.32	58.96	59.99
GPT-4-0125-preview	4.31	5.10	4.42	5.33	68.52	68.37	68.68	68.51

G. Additional plan evaluation results

Apart from the three main metrics in the main paper, we have also evaluated all six large language models on 10+ other metrics. We report these additional evaluation results below.

G.1. Pass rate vs. tool-F1

While we see generally positive effects of feedback on argname-F1 and pass rate, we also observe that feedback can lead to a small decrease (up to 4.5%) in models' tool-F1. Nevertheless, we note that the decrease in tool-F1 with feedback is a lot smaller compared to the gains in pass rate (Figure 14), which suggests feedback can greatly improve tool invocation at a small cost to tool selection.

G.2. No feedback

In the main paper, we present the results of models with verification and/or execution of feedback (on top of parsing feedback) using the experiment with parsing (P) feedback as a baseline. Here, we report the results using the experiment with no feedback at all as the baseline in Table 4. We see that our main takeaway remains the same with this change: feedback helps improve models' argname-F1 by a small amount and pass rate by a lot, although it can lead to a small decrease in tool-F1. We additionally observe the improvement of verification and/or execution feedback on pass rate is larger than that of parsing feedback.

G.3. Step-level metrics

Besides tool-F1 and argname-F1, we also report the following step-level metrics: argvalue-F1 (Table 5), edge-F1 (Table 6), and normalized edit distance (Table 7). We adapted TaskBench's [23] implementation of these metrics on our benchmark. We caution readers about argvalue-F1 as it is computed based on exact matching to one groundtruth value even though there can be multiple valid values.

G.4. Plan-level accuracy

Since step-level metrics do not take into account the ordering of the predicted tools, we additionally include plan-level accuracy to evaluate the whole plan's correctness (Table 8). We highlight two main variants of plan accuracy in Table 8, where the first one considers a list of tool names as a plan and the second considers a list of (tool name, argument names) tuples as a plan. As there could be multiple valid

Table 12. **Average turn count.** We present the average number of conversation turns in step-by-step and multi-step planning with JSON-format generation and different types of feedback.

		Average # of turns				
model	strategy	N/A	P	PV	PE	PVE
Llama-2-7b	step-by-step	2.00	3.54	4.03	3.26	3.52
	multi-step	1.00	1.10	2.18	1.95	1.99
Llama-2-13b	step-by-step	2.87	2.87	3.09	3.06	2.99
	multi-step	1.00	1.04	1.98	1.91	1.97
Mixtral-8x7B	step-by-step	2.98	6.37	5.55	6.02	6.09
	multi-step	1.00	1.14	2.43	2.74	2.81
Gemini-pro	step-by-step	2.31	3.01	2.28	3.67	3.78
	multi-step	1.00	1.20	1.84	1.80	1.88
GPT-3.5-turbo-0125	step-by-step	2.40	3.39	4.10	5.43	5.30
	multi-step	1.00	1.02	1.36	1.46	1.62
GPT-4-0125-preview	step-by-step	3.22	3.52	3.51	3.59	3.59
	multi-step	1.00	1.00	1.05	1.06	1.07

Table 13. **Average number of input and output tokens**

		Avg # of input tokens					Avg # of output tokens				
model	strategy	N/A	P	PV	PE	PVE	N/A	P	PV	PE	PVE
Llama-2-7b	step-by-step	5497.25	20627.60	22021.08	14356.79	13562.25	108.54	659.02	673.01	436.63	432.34
	multi-step	2184.19	3065.88	10215.74	6792.83	8570.81	273.65	320.95	735.02	478.79	636.73
Llama-2-13b	step-by-step	13084.77	14793.73	13962.84	11498.10	13025.18	535.74	620.00	495.34	446.56	489.17
	multi-step	2184.19	2651.22	8141.48	7375.54	8309.38	326.91	345.01	738.19	648.41	753.93
Gemini-pro	step-by-step	5661.28	7651.78	5653.98	10136.36	10560.46	115.70	171.22	96.98	216.03	232.53
	multi-step	2184.19	3062.00	4962.19	4786.80	5022.53	86.12	155.05	219.64	216.77	225.45
GPT-3.5-turbo-0125	step-by-step	5891.36	8938.04	11693.37	16497.09	15966.33	109.61	189.53	207.51	317.43	318.30
	multi-step	2184.19	2247.54	3199.10	3502.05	4017.90	96.24	99.47	136.24	149.94	166.76
GPT-4-0125-preview	step-by-step	8046.55	8852.87	8832.17	9601.61	9618.19	166.17	172.37	171.03	235.51	236.76
	multi-step	2184.19	2184.19	2318.98	2331.06	2354.78	102.28	103.49	110.55	107.74	111.09

plans of the same query, we have also included the Δ in plan accuracy considering alternative plans in Table 9 and shown that our set of alternative plans can recover 1-5% examples where the models could have output potential valid plans different from the one human-verified groundtruth plan. Finally, we also present the strictest form of plan accuracy, which considers a list of tool names, argument names and values as a plan in Table 10. We note that exact matching gives us (Table 10 left) extremely low scores while using entailment in the case of text values – if the predicted argument text entails the label text – gives us more reasonable scores (Table 10 right).

G.5. Code-specific metrics: AST accuracy and CodeBLEU

To evaluate code generation properly, we have also included code-specific metrics such as AST accuracy and CodeBLEU (Table 11). AST accuracy measures if the AST tree of the predicted code is the same as the label code, whereas CodeBLEU measures the similarity of the predicted code to the reference code. We find that feedback, especially verification feedback, can help improve models' AST accuracy but not necessarily CodeBLEU scores.

G.6. Efficiency

Besides models' planning performance, we also kept track of their token usage (Table 13) and numbers of conversation turns (Table 12). As expected, step-by-step planning generally requires more conversation turns and more tokens than multi-step planning. Similarly, feedback also increases token usage.

H. Human evaluation of plan execution results

Since *m&m*'s consists of open-ended queries, which do not always have one single final answer, it is challenging to evaluate the execution results of the plans automatically. Thus, we resort to human evaluation of a small subset of 85 examples with reasonable execution results. Our manual

Table 14. **Execution results accuracy.** We present the execution results accuracy of GPT-4 and Mixtral-8x7B on a selected subset of 85 examples across different setups, including step-by-step and multi-step planning, with JSON-format and code generation, and different types of feedback.

model	strategy	format	feedback	accuracy
Mixtral-8x7B	multi-step	JSON	P	42.94 ± 1.76
GPT-4-0125-preview	step-by-step	JSON	P	49.41 ± 1.18
GPT-4-0125-preview	multi-step	Code	P	61.18 ± 0.0
GPT-4-0125-preview	multi-step	JSON	PVE	64.12 ± 2.94
GPT-4-0125-preview	multi-step	JSON	P	70.00 ± 6.47

evaluation reveals that GPT-4 achieves the best execution accuracy with multi-step planning and JSON-format generation compared to step-by-step planning or code generation (Table 14). Further, we learn that our main metrics, especially pass rate, correlate well with the execution accuracy (Figure 15).

```
# TOOL LIST #:
text generation: It takes an input text prompt and outputs a text that is most likely to follow the input text. Its input includes text, and output includes text.
text summarization: it takes a paragraph of text and summarizes into a few sentences. Its input includes text, and output includes text.
text classification: It takes a text and classifies it into a category in the model's vocabulary (e.g. positive or negative based on its sentiment). Its input includes text, and output includes text.
question answering: It takes a text and a question, and outputs an answer to that question based on the text. Its input includes text, question, and output includes text.
image generation: It takes a text prompt and generates an image that matches the text description. Its input includes text, and output includes image.
image captioning: It takes an image and generates a text caption of the image. Its input includes image, and output includes text.
optical character recognition: It takes an image and outputs recognized texts in the image. Its input includes image, and output includes text.
image classification: It takes an image and classifies the subject in the image into a category such as cat or dog. Its input includes image, and output includes text.
image editing: It takes an image and a text prompt and outputs a new image based on the text. Its input includes image, prompt, and output includes image.
object detection: It takes an image and outputs rectangular bounding boxes of objects detected in the image. Its input includes image, and output includes image, objects.
image segmentation: It takes an image, segments it into different parts, and outputs segmentation masks of any shape for the parts. Its input includes image, and output includes image, objects.
automatic speech recognition: It takes an audio file and produces a transcription of the audio. Its input includes audio, and output includes text.
visual question answering: It takes an image and a question about the image, and generates an answer to the question. Its input includes image, question, and output includes text.
image crop: It takes an image and 4 numbers representing the coordinates of a bounding box and crops the image to the region within the box. Its input includes image, object, and output includes image.
image crop left: It takes an image, crops and keeps the left part of the image. Its input includes image, and output includes image.
image crop right: It takes an image, crops and keeps the right part of the image. Its input includes image, and output includes image.
image crop top: It takes an image, crops and keeps the top part of the image. Its input includes image, and output includes image.
image crop bottom: It takes an image, crops and keeps the bottom part of the image. Its input includes image, and output includes image.
background blur: It takes an image and one or multiple objects in the foreground, and returns an image where the backgroud is blurred. Its input includes image, object, and output includes image.
color pop: It takes an image and one or multiple objects, and returns an image where only the object is colored and the rest is black and white. Its input includes image, object and output includes image.
count: It takes a list of objects and returns the count of the objects. Its input includes objects, and output includes number.
tag: It takes an image and a list of objects with their bounding boxes and classes, and tags all the objects Its input includes image, objects, and output includes image.
select object: It takes a list of objects, and selects the object based on the input object name. Its input includes objects, object_name, and output includes object.
emoji: It takes an image and the bounding box coordinates of one or multiple objects, and replaces the object with an emoji (e.g. angry/flushed/crying/dizzy/sleepy/grimacing/kissing/smiling_face, alien, ghost, goblin etc). Its input includes image, object, emoji, and output includes image.
get date fact: It provides interesting facts about dates. Its input includes date, and output includes text.
get year fact: It provides interesting facts about years. Its input includes year, and output includes text.
get math fact: It provides interesting math facts about numbers. Its input includes number, and output includes text.
get trivia fact: It provides interesting trivia facts about number. Its input includes number, and output includes text.
love calculator: Enter your name and the name of your partner/lover/crush to find Love compatibility & chances of successful love relationship. Its input includes first_name, second_name, and output includes number.
get location: Convert a city name or address to geographical coordinates using OpenStreetMap's Nominatim API. Its input includes city, and output includes lon, lat.
search movie: Retrieve basic movie information, including title, year, genre, and director. Its input includes movie_title, movie_year, and output includes text.
get weather: Provides weather forecast data based on specific geographical coordinates. Its input includes lon, lat, and output includes objects.
wikipedia simple search: Perform a basic search query on Wikipedia to retrieve a summary of the most relevant page. Its input includes text, and output includes text.

# GOAL #: Based on the above tools, I want you to generate the task nodes to solve the # USER REQUEST #. The format must be in a strict JSON format, like: {"nodes": [{"id": an integer id of the tool, starting from 0, "name": "tool name must be from # TOOL LIST #", "args": { a dictionary of arguments for the tool. Either original text, or user-mentioned filename, or tag '<node-j>.text' (start from 0) to refer to the text output of the j-th node. }}]}

# REQUIREMENTS #:
1. the generated tool nodes can resolve the given user request # USER REQUEST # perfectly. Tool name must be selected from # TOOL LIST #;
2. The arguments of a tool must be the same number, modality, and format specified in # TOOL LIST #;
3. Use as few tools as possible.

# EXAMPLE #:
# USER REQUEST #: "Based on reading the article titled 'Would you rather have an Apple Watch - or a BABY?', generate an extended paragraph on the topic."
# RESULT #: {"nodes": [{"id": 0, "name": "text generation", "args": {"text": "an extended paragraph on the topic: Would you rather have an Apple Watch - or a BABY?"}}]}
# EXAMPLE #:
# USER REQUEST #: "Could you take the image, specifically 'image 17320.jpg', and adjust it so the green ball in the picture becomes blue, then describe for me what the resulting
image looks like?"
# RESULT #: {"nodes": [{"id": 0, "name": "image editing", "args": {"image": "17320.jpg", "prompt": "change the green ball to blue"}}, {"id": 1, "name": "image captioning", "args: {"image": "<node-0>.image"}}]}
# EXAMPLE #:
# USER REQUEST #: "Could you provide a brief summary of the key points discussed in the audio file '1995-1826-0002.flac' about John Taylor and his interest in cotton? And then, can you also help me create a vivid illustration based on the key points?"
# RESULT #: {"nodes": [{"id": 0, "name": "automatic speech recognition", "args": {"audio": "1995-1826-0002.flac"}}, {"id": 1, "name": "text summarization", "args": {"text": "<node-0>.text"}}, {"id": 2, "name": "image generation", "args": {"text": "a vivid illustration based on <node-1>.text"}}]}

# USER REQUEST #: "I need to give a quick presentation for kindergarteners on 'Why is the sky blue?'. I don't really have time to sift through lots of complex information and I need simple, straightforward explanations with a relevant image that kids can understand. Can you assist me with that?"
Now please generate your result in a strict JSON format:
# RESULT #:
```

Figure 10. **Multi-step planning prompt.** We present the full prompt used for multi-step planning.

CVPR
#16

```
# TOOL LIST #:
text generation: It takes an input text prompt and outputs a text that is most likely to follow the input text. Its input includes text, and output includes text.
text summarization: it takes a paragraph of text and summarizes into a few sentences. Its input includes text, and output includes text.
......
search movie: Retrieve basic movie information, including title, year, genre, and director. Its input includes movie_title, movie_year, and output includes text.
get weather: Provides weather forecast data based on specific geographical coordinates. Its input includes lon, lat, and output includes objects.
wikipedia simple search: Perform a basic search query on Wikipedia to retrieve a summary of the most relevant page. Its input includes text, and output includes text.

# GOAL #: Based on the above tools, I want you to reason about how to solve the # USER REQUEST # and generate the actions step by step.

# REQUIREMENTS #:
1. The thoughts can be any free form texts to help with action generation;
2. The action must follow this JSON format strictly: {"id": an integer id of the tool, starting from 0, which should be the same as the id of the ACTION "name": "tool name must be from # TOOL LIST #", "args": { a dictionary of
arguments for the tool. Either original text, or user-mentioned filename, or tag '<node-j>.text' (start from 0) to refer to the text output of the j-th node. }};
3. The arguments of a tool must match the number, modality, and format of the tool's arguments specified in # TOOL LIST #.
# EXAMPLE #:
# USER REQUEST #: "Based on reading the article titled 'Would you rather have an Apple Watch - or a BABY?', generate an extended paragraph on the topic."
# RESULT #:
THOUGHT 0: First, I need to perform text generation.
ACTION 0: {"id": 0, "name": "text generation", "args": {"text": "Would you rather have an Apple Watch - or a BABY?"}}

# EXAMPLE #:
# USER REQUEST #: "Could you take the image, specifically 'image 17320.jpg', and adjust it so the green ball in the picture becomes blue, then describe for me what the resulting
image looks like?"
# RESULT #:
THOUGHT 0: First, I need to perform image editing.
ACTION 0: {"id": 0, "name": "image editing", "args": {"image": "17320.jpg", "prompt": "change the green ball to blue"}}
OBSERVATION 0: {'image': 'an image with a blue ball in it'}
THOUGHT 1: Based on the user query and OBSERVATION 0, then, I need to perform image captioning.
ACTION 1: {"id": 1, "name": "image captioning", "args": {"image": "<node-0>.image"}}

# EXAMPLE #:
# USER REQUEST #: "Could you provide a brief summary of the key points discussed in the audio file '1995-1826-0002.flac' about John Taylor and his interest in cotton? And then, c
an you also help me create a vivid illustration based on the key points?"
# RESULT #:
THOUGHT 0: First, I need to perform automatic speech recognition.
ACTION 0: {"id": 0, "name": "automatic speech recognition", "args": {"audio": "1995-1826-0002.flac"}}
OBSERVATION 0: {'text': 'John Taylor, who had supported her through college, was interested in cotton.'}
THOUGHT 1: Based on the user query and OBSERVATION 0, then, I need to perform text summarization.
ACTION 1: {"id": 1, "name": "text summarization", "args": {"text": "<node-0>.text"}}
OBSERVATION 1: {'text': 'John Taylor was interested in cotton.'}
THOUGHT 2: Based on the user query and OBSERVATION 1, then, I need to perform image generation.
ACTION 2: {"id": 2, "name": "image generation", "args": {"text": "a vivid illustration based on <node-1>.text"}}

# USER REQUEST #: "I need to give a quick presentation for kindergarteners on 'Why is the sky blue?'. I don't really have time to sift through lots of complex information and I need simple, straightforward explanations with a relevant image that kids can understand. Can you assist me with that?"
ction of it?
Now please generate only THOUGHT 0 and ACTION 0 in RESULT:
# RESULT #:
```

Figure 11. **Step-by-step planning prompt.** We present the full prompt used for step-by-step planning.

```
# TOOL LIST #:
text_generation(text) → text: It takes an input text prompt and outputs a text that is most likely to follow the input text.
text_summarization(text) → text: it takes a paragraph of text and summarizes into a few sentences.
text_classification(text) → text: It takes a text and classifies it into a category in the model's vocaburary (e.g. positive or negative based on its sentiment).
question_answering(text, question) → text: It takes a text and a question, and outputs an answer to that question based on the text.
image_generation(text) → image: It takes a text prompt and generates an image that matches the text description.
image_captioning(image) → text: It takes an image and generates a text caption of the image.
optical_character_recognition(image) → text: It takes an image and outputs recognized texts in the image.
image_classification(image) → text: It takes an image and classifies the subject in the image into a category such as cat or dog.
image_editing(image, prompt) → image: It takes an image and a text prompt and outputs a new image based on the text.
object_detection(image) → image, objects: It takes an image and outputs rectangular bounding boxes of objects detected in the image.
image_segmentation(image) → image, objects: It takes an image, segments it into different parts, and outputs segmentation masks of any shape for the parts.
automatic_speech_recognition(audio) → text: It takes an audio file and produces a transcription of the audio.
visual_question_answering(image, question) → text: It takes an image and a question about the image, and generates an answer to the question.
image_crop(image, object) → image: It takes an image and 4 numbers representing the coordinates of a bounding box and crops the image to the region within the box.
image_crop_left(image) → image: It takes an image, crops and keeps the left part of the image.
image_crop_right(image) → image: It takes an image, crops and keeps the right part of the image.
image_crop_top(image) → image: It takes an image, crops and keeps the top part of the image.
image_crop_bottom(image) → image: It takes an image, crops and keeps the bottom part of the image.
background_blur(image, object) → image: It takes an image and one or multiple objects in the foreground, and returns an image where the backgroud is blurred.
color_pop(image, object) → image: It takes an image and one or multiple objects, and returns an image where only the object is colored and the rest is black and white.
count(objects) → number: It takes a list of objects and returns the count of the objects.
tag(image, objects) → image: It takes an image and a list of objects with their bounding boxes and classes, and tags all the objects
select_object(objects, object_name) → object: It takes a list of objects, and selects the object based on the input object name.
emoji(image, object, emoji) → image: It takes an image and the bounding box coordinates of one or multiple objects, and replaces the object with an emoji (e.g. angry/flushed/crying/dizzy/sleepy/grimacing/kissing/smiling_face, alien, ghost,
goblin etc).
get_date_fact(date) → text: It provides interesting facts about dates.
get_year_fact(year) → text: It provides interesting facts about years.
get_math_fact(number) → text: It provides interesting math facts about numbers.
get_trivia_fact(number) → text: It provides interesting trivia facts about number.
love_calculator(first_name, second_name) → number: Enter your name and the name of your partner/lover/crush to find Love compatibility & chances of successful love relationship.
get_location(city) → lon, lat: Convert a city name or address to geographical coordinates using OpenStreetMap's Nominatim API.
search_movie(movie_title, movie_year) → text: Retrieve basic movie information, including title, year, genre, and director.
get_weather(lon, lat) → objects: Provides weather forecast data based on specific geographical coordinates.
wikipedia_simple_search(text) → text: Perform a basic search query on Wikipedia to retrieve a summary of the most relevant page.

# GOAL #: Based on the above tools, I want you to generate a python program to solve the # USER REQUEST #.

# REQUIREMENTS #:
1. the generated program can resolve the given user request # USER REQUEST # perfectly. The functions must be selected from # TOOL LIST #;
2. The arguments of a function must be the same number, modality, and format specified in # TOOL LIST #;
3. Use as few tools as possible.

# EXAMPLE #:
# USER REQUEST #: "Based on reading the article titled 'Would you rather have an Apple Watch - or a BABY?', generate an extended paragraph on the topic."
# RESULT #: ```python
def solve():
   output0 = text_generation(text="an extended paragraph on the topic: Would you rather have an Apple Watch - or a BABY?")
   result = {0: output0}
   return result
```
# EXAMPLE #:
# USER REQUEST #: "Could you take the image, specifically 'image 17320.jpg', and adjust it so the green ball in the picture becomes blue, then describe for me what the resulting image looks like?"
# RESULT #: ```python
def solve():
   output0 = image_editing(image="17320.jpg", prompt="change the green ball to blue")
   output1 = image_captioning(image=output0['image'])
   result = {0: output0, 1: output1}
   return result
```
# EXAMPLE #:
# USER REQUEST #: "Could you provide a brief summary of the key points discussed in the audio file '1995-1826-0002.flac' about John Taylor and his interest in cotton? And then, can you also help me create a vivid illustration based on the ke
y points?"
# RESULT #: ```python
def solve():
   output0 = automatic_speech_recognition(audio="1995-1826-0002.flac")
   output1 = text_summarization(text=f"{output0['text']}")
   output2 = image_generation(text=f"a vivid illustration based on {output1['text']}")
   result = {0: output0, 1: output1, 2: output2}
   return result
```

USER REQUEST #: "I need to give a quick presentation for kindergarteners on 'Why is the sky blue?'. I don't really have time to sift through lots of complex information and I need simple, straightforward explanations with a relevant image that kids can understand. Can you assist me with that?"
Now please generate your program enclosed in ```python ```:
```

Figure 12. **Code generation prompt.** We present the full prompt used for code generation.
```

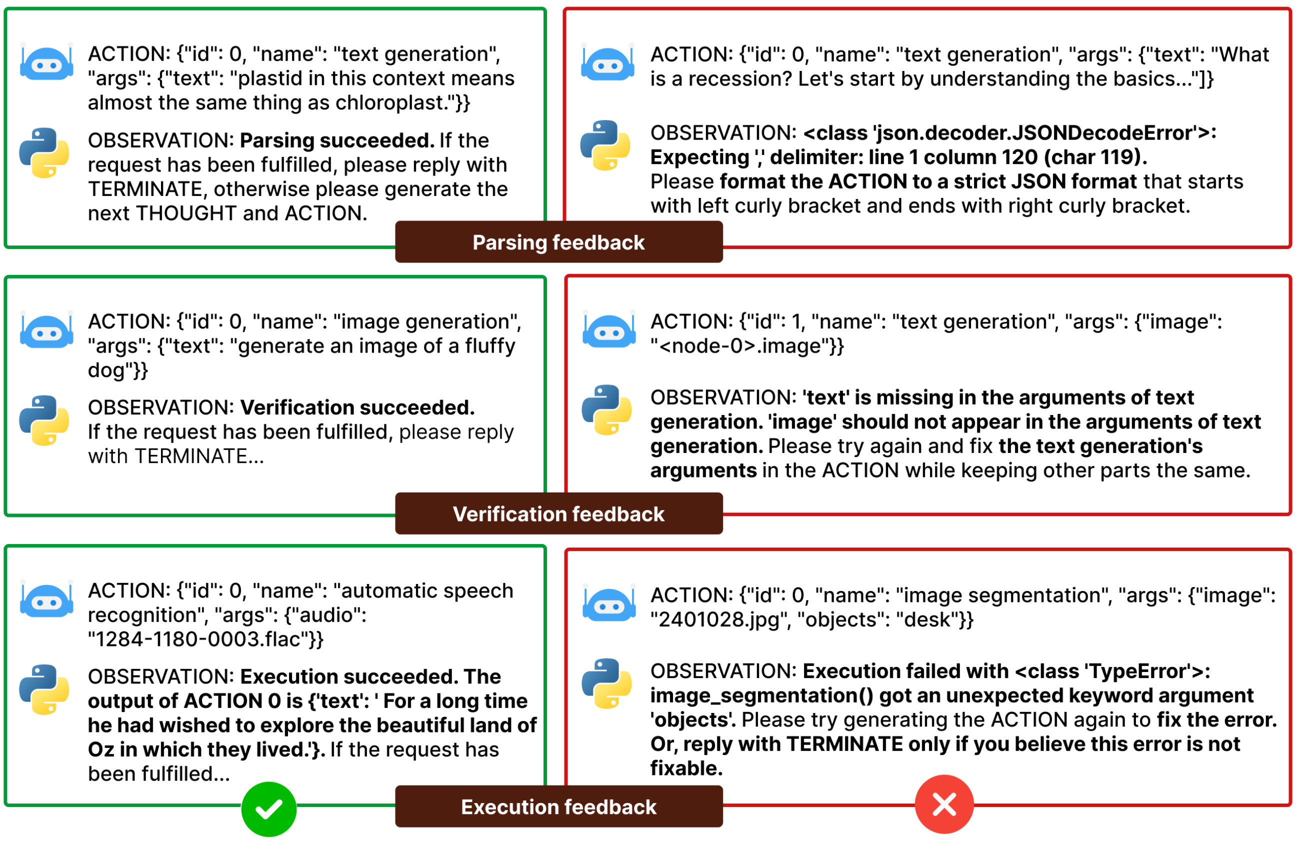

Figure 13. **Types of feedback.** We present examples of parsing, verification, and execution feedback in both success and failure cases.

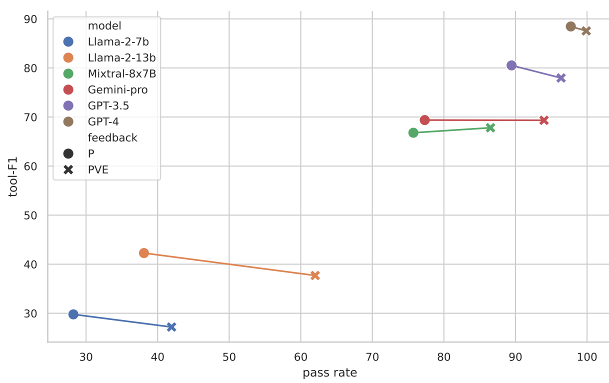

Figure 14. **Comparing without vs. with feedback.** P = parsing feedback, PVE = parsing, verification, and execution feedback. We find that feedback greatly improves planning agents' pass rates across different model sizes, especially for Llama-7b, Llama-13b, and Gemini-pro. However, feedback can also harm models' tool prediction performance and decrease their tool-F1 by up to 5%.

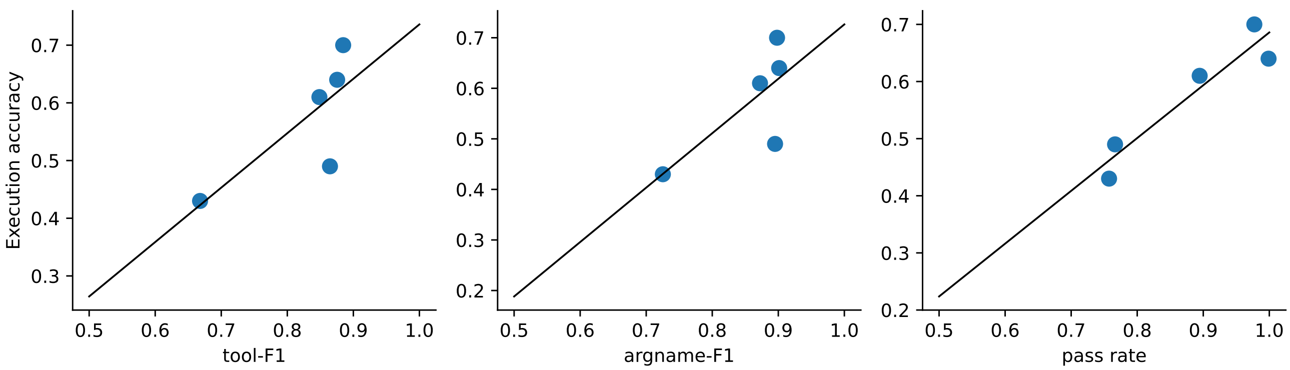

Figure 15. **Execution results accuracy vs. other metrics.** We find a strong positive correlation between the execution results accuracy and our main metrics, including tool-F1, argname-F1, and pass rate, with Pearson correlation coefficients $r = 0.77, 0.73, 0.95$ respectively.

User request	Label	Prediction
A. Missing tool(s) due to early termination	**Step-by-step / JSON / Parsing**	
I have an image labeled '86582-input.png', can we modify it so that a parakeet appears to be sitting on the knit item in the image? Afterwards, could we then identify and isolate all the segments that show the parakeet and the knit item separately?	{'id': 0, 'name': 'image editing', 'args': {'image': '86582-input.png', 'prompt': 'a parakeet should be sitting on the knit item'}} {'id': 1, 'name': 'image segmentation', 'args': {'image': '<node-0>.image'}}	THOUGHT 0: First, I need to perform image editing. ACTION 0: {"id": 0, "name": "image editing", "args": {"image": "86582-input.png", "prompt": "add a parakeet sitting on the knit item"}} -- OBSERVATION: Parsing succeeded. If the request has been fulfilled, please reply with TERMINATE, otherwise please generate the next THOUGHT and ACTION. -- Great, I'm glad the parsing was successful! Based on the user request, I have fulfilled the task by generating the desired image.
B. Change/removal of correct tools	**Multi-step / JSON / Execution**	
I'm a bit curious about two characters from my favorite movie, Jack Twist and Ennis Del Mar. Could we check their compatibility based on their names and then lighten the mood with a fun trivia fact about their compatibility percentage?	{'id': 0, 'name': 'love calculator', 'args': {'first_name': 'Jack Twist', 'second_name': 'Ennis Del Mar'}} {'id': 1, 'name': 'get trivia fact', 'args': {'number': '<node-0>.number'}}	{"id": 0, "name": "love calculator", "args": {"first_name": "Jack Twist", "second_name": "Ennis Del Mar"}}, {"id": 1, "name": "get trivia fact", "args": {"number": "<node-0>.output"}}]} -- Execution failed with <class 'KeyError'>: 'output'. Please try generating the # RESULT # again to fix the error. Or, reply with TERMINATE only if you believe this error is not fixable. -- {"id": 0, "name": "love calculator", "args": {"first_name": "Jack Twist", "second_name": "Ennis Del Mar"}}, {"id": 1, "name": "love calculator", "args": {"first_name": "Jack Twist", "second_name": "Ennis Del Mar"}} -- Execution succeeded. The output is {'number': '79', 'message': 'All the best!'}.
	Multi-step / JSON / Verification	
I have a picture in the filename '07600.jpg' and I need to understand what's in the image. Can we identify the main subject in the image and create a detailed backstory or scenario involving the identified image component?	{'id': 0, 'name': 'image classification', 'args': {'image': '07600.jpg'}} {'id': 1, 'name': 'text generation', 'args': {'text': 'a detailed backstory or scenario involving the identified component <node-0>.text'}}	{"id": 0, "name": "image classification", "args": {"image": "07600.jpg"}}, {"id": 1, "name": "image captioning", "args": {"image": "<node-0>.text"}} -- image captioning should not follow image classification, because it does not make sense to feed 'the text label for the class' into image captioning, which expects the input to be 'an image to be captioned'. Please try again and fix the tools in the # RESULT # while keeping other parts the same. -- {"id": 0, "name": "image captioning", "args": {"image": "07600.jpg"}} -- Verification succeeded.
C. Failure to to access a tool's output	**Multi-step / Code / Parsing**	
I'm writing a storyline for a new Italian cookbook. Right now, I only have this line: 'The Italian way with the knife is done.' Could you help me generate a few paragraphs that could logically follow this line and then evaluate the tone of the generated text to ensure it instills positivity and excitement in the readers?	output0 = text_generation(text="Generate a few paragraphs that could logically follow this line: 'The Italian way with the knife is done.'") output1 = text_classification(text=output0['text'])	output0 = text_generation(text="The Italian way with the knife is done.") output1 = text_classification(text=output0) -- Parsing succeeded.
I'm planning a vacation to Phoenix. Could you tell me what the weather forecast is likely to be for this city?	output0 = get_location(city="Phoenix") output1 = get_weather(lon=output0['lon']}, lat=output0['lat'])	lon, lat = get_location(city="Phoenix") output0 = get_weather(lon=lon, lat=lat) -- Parsing succeeded.

Figure 16. We present examples of three common errors (A-C) in step-by-step planning and multi-step planning with JSON-format generation as well as in code generation with various feedback types.