

COPILOT: Controlling Fine-grained Provider Preference in LLM Recommendations

Xiaoyu Zhang^{1,*}, Yimin Ding^{2,*}, Qingshuang Bao², Chenhao Lin^{2,†},
Chao Shen², Bo Yang³, Yang Liu¹

¹Nanyang Technological University ²Xi'an Jiaotong University

³Beijing Forestry University

joshingrain@gmail.com, {2215311791, qs_bao}@stu.xjtu.edu.cn,

{linchenhao, chaoshen}@xjtu.edu.cn, yangbo@bjfu.edu.cn, yangliu@ntu.edu.sg

Abstract

Large language models (LLMs) increasingly serve as recommendation engines, yet existing preference control work primarily focuses on coarse-grained objectives such as safety, helpfulness, or writing style, while overlooking fine-grained *provider preferences*, i.e., user preferences over specific brands, platforms, or services. Such preferences are often multi-intent, context-dependent, and dynamically changing, making them difficult to control with prompting, fine-tuning, or activation steering methods. In this paper, we present COPILOT, a plug-and-play, training-free framework. It first decomposes each prompt into sub-tasks with corresponding user-specific provider preference configurations. Then it uses a diffusion language model to draft sub-task responses with hard preference constraints injected at the initial mask, which are finally completed into fluent answers by the target LLM. Experiments across code and text domains on three target LLMs show that COPILOT effectively improves provider preference satisfaction by up to 58.9% over the strongest baselines while reducing irrelevant provider mentions by up to 87.8% to ensure recommendation quality. This work aims to lay a foundation for LLM recommendations that balance personalized experience with real-world application values.

1 Introduction

Large language models (LLMs) are increasingly used as recommendation agents in domains such as coding assistants, online services, and product selections (Sanner et al., 2023; Wu et al., 2024; Search, 2025). However, current LLM recommendations often fail to reflect users' preferences over specific providers, technology platforms, or brands (Zhang et al., 2025; Deldjoo, 2024). Such mismatches force users to repeatedly revise

prompts or regenerate responses, reducing both recommendation quality and user trust (Srinivasan et al., 2002; Wang et al., 2023b). Aligning LLM recommendations with users' personalized preferences has therefore become a core issue in building trustworthy and usable LLM applications.

Existing work on LLM preference control and alignment (Gao et al., 2024; Ouyang et al., 2022; Lyu et al., 2024; Mysore et al., 2023a) focuses either on coarse-grained objectives such as safety and helpfulness (Ouyang et al., 2022; Gao et al., 2024), or on stylistic dimensions such as tone and writing style (Lyu et al., 2024; Mysore et al., 2023a). Neither addresses the fine-grained **provider preference** behind many real-world requests, where users favor specific brands for online services (Srinivasan et al., 2002) and developers form path dependencies on particular languages, frameworks, and cloud platforms shaped by existing technology stacks, organizational norms, and cost considerations (Stack Overflow, 2025). Unlike stable stylistic preferences, provider preferences span multiple scenarios and candidate providers that evolve over time. Moreover, a single user prompt typically activates only a subset of them, demanding precise activation rather than enumeration. Controlling such multi-intent and dynamic provider preferences in LLM recommendations remains an important yet largely unexplored challenge. Therefore, it calls for a dedicated solution beyond existing coarse-grained alignment methods, enabling commercial and personalized LLM applications.

Existing preference control methods have two limitations when applied to the provider preference. ❶ *Low Reliability in Multi-Intent Contexts*. When a single user prompt contains sub-tasks involving multiple scenarios, existing prompt-based methods may confound the preferences on different providers, leading to unstable and unreliable preference control (Zhang et al., 2025), as illustrated in Fig. 1 (❶). In addition, although gen-

*Equal contribution.

†Corresponding author.

erating multiple candidate results and re-ranking from them can improve the preference control effectiveness (Ramirez et al., 2023; Yun et al., 2025), the sampling process introduces significant latency and computational overhead, making it impractical in real-world applications. ② *Prohibitive Adaptation Cost*. Provider preferences are dynamic, frequently shifting in response to evolving project requirements, budgets, or external environments and events. Existing preference alignment methods based on fine-tuning and steering (Huang et al., 2023; Gao et al., 2024; Cao et al., 2024) require substantial annotated data and heavy training overhead, making it impractical to rapidly adapt to the rich and continuously evolving provider preferences.

To fill this gap, we propose COPILOT, a training-free framework that leverages the collaboration of the diffusion language model (DLM) and LLM to satisfy users’ diverse and dynamic provider preferences. COPILOT consists of three modules, namely *scenario planning*, *draft generation*, and *response completion*. Given a user prompt and a preference configuration, COPILOT first decomposes the prompt into sub-tasks and matches each one with its target preference scenario (§4.1). It then exploits a DLM with constrained decoding to draft sub-task responses in parallel, injecting user preferences into the initial mask as hard constraints (§4.2). The target LLM finally merges these drafts and completes them into a fluent answer (§4.3). Such a design leverages DLM’s advantage in parallel diffusion decoding, accurately imposing fine-grained provider preferences across diverse scenarios, without any additional training. Experiments on the code and text domains across three target LLMs show that COPILOT improves provider preference satisfaction by up to 58.9% over the strongest baselines while preserving recommendation quality.

The contribution of this paper is as follows. ① We formulate the problem of *fine-grained provider preference control in LLM recommendation*, and reveal its importance and challenges in real-world applications, providing a new perspective and direction for future research on personalized LLM and its commercial applications. ② We design a practical training-free provider preference control framework, COPILOT. It innovatively introduces the collaborative workflow of DLM and LLM in the preference control, achieving accurate provider preference control of LLM recommendation results. ③ We evaluate COPILOT on three target LLMs across two recommendation domains, and COPI-

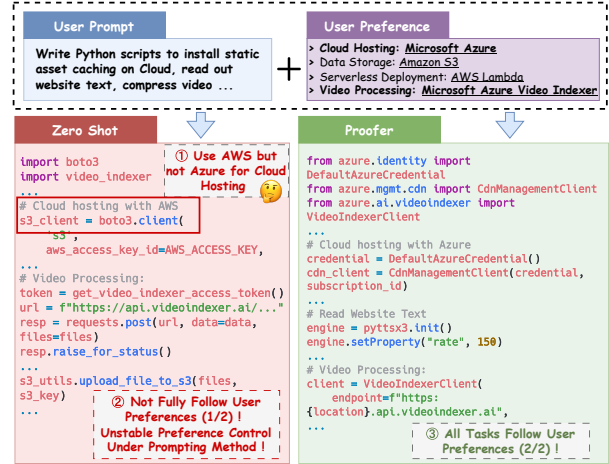


Fig. 1: Motivation example of COPILOT. (Given a user prompt that covers three sub-tasks (‘install static asset caching on cloud’, ‘read out website text’, and ‘compress video’) and a four-entry preference configuration, only the first and third sub-tasks match the *cloud hosting* and *video processing* scenarios in this configuration. Standard zero-shot prompting incorrectly invokes AWS for cloud hosting although the user prefers Microsoft Azure, and grounds only one of the two relevant preferences. COPILOT grounds both relevant preferences correctly (Microsoft Azure for cloud hosting and Microsoft Azure Video Indexer for video processing) and leaves the unmatched sub-task ‘read out website text’ to the target LLM’s default generation.)

LOT outperforms baseline methods in preference control while maintaining high recommendation quality. ④ Our implementation and data are publicly available¹.

2 Background & Related Work

LLM Recommendation. Existing LLM recommendation research can be divided into two categories based on how LLMs are used (Wu et al., 2024). The first category combines LLMs with traditional recommendation models to enhance the performance of the recommendation system (RS) (Wu et al., 2021; Zhang et al., 2022; Yang et al., 2022). Recently, researchers design LLM-Rec, which significantly improves the recommendation quality of RS through four prompting strategies and LLM-enhanced text (Lyu et al., 2024). With the improvement of LLM capabilities, LLMs are gradually evolving from recommendation assistants into recommendation engines that can directly and independently make recommendation decisions (i.e., LLM as RS) (Mysore et al., 2023b; Zhang et al., 2024; Hou et al., 2024; Lin et al., 2024). In recent years, LLM as RS exhibits context understanding and generation capabilities that surpass traditional RS, such as directly recommending code snippets based on user requirements (Liu et al., 2023a; Zhang et al., 2025). This paper pri-

¹ <https://github.com/shiningrain/ProviderPreferenceControl>

marily focuses on the LLM as RS.

Diffusion Language Model. Diffusion models have recently emerged as a promising paradigm in natural language processing, offering distinct advantages over autoregressive models in terms of global planning and generation flexibility (Li et al., 2022; Gong et al., 2023; He et al., 2023). Unlike autoregressive models that generate tokens sequentially from left to right, diffusion language models (DLMs) generate text by iteratively refining the initial masked token into coherent sequences, allowing for bidirectional context modeling and efficient parallel decoding (Han et al., 2023; Gong et al., 2023). Recent studies have demonstrated the superiority of DLMs in complex controllable generation tasks (e.g., code generation) (Singh et al., 2023; Yang et al., 2023). Note that such a masked denoising decoding is a general mechanism rooted in the diffusion paradigm rather than a specific model’s feature. In this paper, we leverage the masked decoding mechanism of DLMs to introduce user preference services as hard constraints in the initial token sequence, thereby effectively and efficiently generating controllable recommendation results.

Stage-wise Problem Solving. Findings in cognitive science reveal that complex generation becomes more reliable when decomposed into distinct stages, as exemplified by the classic process model of writing by Flower and Hayes that frames writing as iterative planning, translating, and reviewing rather than a single pass from intention to text (Flower and Hayes, 1981; Kellogg, 2008). A similar pattern appears in modern LLM workflows, where explicit decomposition, intermediate plans, or scratchpads are introduced to stabilize multi-step generation (Wang et al., 2023a; Shen et al., 2023). Our framework follows the same principle and decomposes recommendation generation into coordinated stages of planning, drafting, and refinement. Note that, different from existing plan-then-generate workflows whose intermediate plans are natural-language texts re-interpreted by the same autoregressive decoder, COPILOT materializes its plan as DLM drafts in which multiple preference anchors are fixed simultaneously in the initial mask. Therefore, the matched provider preferences are structurally bound into the response skeleton rather than left as soft textual instructions.

Personalized LLM. To align LLM responses with human preferences and values, researchers have explored a range of personalization and preference

control techniques (Khanov et al., 2024; Deng and Raffel, 2023; Li et al., 2025; Huang et al., 2025; Mudgal et al., 2024; Wu et al., 2026). One of the mainstream methods is reinforcement learning from human feedback (RLHF) and its variants, which fine-tune the model on large-scale preference datasets to instill general values such as ‘helpfulness’ and ‘harmlessness’ (Ouyang et al., 2022; Gao et al., 2024; Zollo et al., 2025). However, such methods incur substantial training overhead and struggle to adapt to dynamically changing provider preferences. Additionally, inference-stage alternatives steer behavior through carefully designed prompts (Liu et al., 2023b) or activation-layer steering vectors (Cao et al., 2024). Another line of decoding-time alignment methods guides the generation with reward models or classifiers during decoding, such as reward-guided decoding and search (Khanov et al., 2024; Deng and Raffel, 2023; Mudgal et al., 2024) and decoding-time alignment frameworks (Huang et al., 2025; Li et al., 2025). Beyond human preferences, lexically constrained generation methods enforce contextual relevance and output quality at the token level (He, 2021; Zhang et al., 2020; Yun et al., 2025).

In this paper, we study fine-grained provider preference control in LLM recommendation. Compared with the general constraints studied in controllable generation (e.g., lexical or stylistic requirements), provider preferences are unique in that they bind a user-specific scenario-to-service mapping with a concrete realization. Namely, a preference is satisfied only when the matched sub-task actually uses the registered service through its name, SDK, or API calls. Meanwhile, replacing the registered service with an otherwise reasonable alternative still breaks the preference, since such substitutions can violate the procurement, compatibility, or deployment requirements behind the user’s choice.

3 Problem Statement

Different from the prior personalized LLM work that adapts a model toward coarse-grained safety objectives or broad stylistic dimensions, this paper studies **provider preference control**, namely, regulating which concrete service providers, brands, or platforms appear in a recommendation response. Formally, given a user prompt P , a target LLM θ_l , and a preference configuration $\mathcal{C} = \{(s_i, v_i)\}_{i=1}^{|\mathcal{C}|}$ that registers a service v_i from preferred provider for each anticipated scenario s_i , the goal is to pro-

duce a high-quality response R such that every sub-task t whose intent matches the scenario $s_i \in \mathcal{C}$ is realized with v_i , while sub-tasks unmatched by $\mathcal{C}$ are left to the default behavior of θ_l .

$$R^* = \arg \max_R Q(R | P, \theta_l), \quad (1)$$

where the function $Q(\cdot)$ evaluates the given response R from two complementary dimensions.

❶ *Preference adherence dimension* requires every sub-task whose intent matches a registered scenario $s_i \in \mathcal{C}$ to use the preferred service v_i in the answers. ❷ *Response quality dimension* requires the overall response to stay fluent, self-consistent, and informative. Note that this dimension also penalizes hallucinated providers or the enumeration of irrelevant providers.

In real-world applications, the provider preference control task exhibits two unique characteristics that expose the limitations of existing techniques. ❶ *On-demand Activation with Multi-intent*. The preference configuration $\mathcal{C}$ (submitted by users or aggregated from historical conversations) can accumulate dozens of preferences and corresponding scenarios. A single user prompt P typically interleaves multiple sub-tasks and therefore activates only a small subset of $\mathcal{C}$, requiring the model to bind each sub-task to its own matched preference. However, existing prompt-based methods handle this targeted activation poorly and could misroute providers to unrelated sub-tasks. They may even mechanically enumerate every registered service in the response, leading to low response quality. ❷ *High Dynamism*. User preferences for providers fluctuate frequently as personal interests evolve, project budgets shift, or the technology stacks iterate. However, mainstream training-based and steering-based alignment methods rely on large-scale annotated data and a heavy training process. Consequently, it is impractical for them to keep pace with the real-time preference updates that production recommendation systems demand. Therefore, there is an urgent need for a training-free and precise provider preference control method, which is a significant step towards realizing a truly personalized and commercially viable next-generation LLM recommendation system.

4 Design

To overcome existing limitations, we propose COPILOT, a training-free framework. Our design

is inspired by the cognitive process model of writing proposed by the prior work (Flower and Hayes, 1981), which posits that high-quality writing involves three core phases, namely *planning* (organizing ideas and goals), *translating* (converting ideas into text), and *reviewing* (refining the output). Adapting this cognitive process model to recommendation generation, COPILOT implements a ‘Plan-Draft-Complete’ workflow comprising three corresponding modules, as illustrated in Fig. 2.

Specifically, given a user prompt P and a pre-defined preference configuration $\mathcal{C}$, COPILOT aims to produce a high-quality response R that follows the user’s preferences through three coordinated modules. ❶ COPILOT starts by organizing the generation goals (i.e., preferences). The *scenario planning* module first decomposes the prompt P into multiple sub-tasks and matches each sub-task to a specific scenario in $\mathcal{C}$ when one exists, yielding a structured task list $\mathcal{T}$. ❷ Then, in the *draft generation* module, COPILOT translates the goals into drafts. It constructs hard constraints for each sub-task based on the matched preference and employs a DLM to generate the draft responses D containing preference services in parallel. ❸ Finally, COPILOT refines and completes the final output. The *response completion* module merges the drafts D , queries the target LLM to complete the response R that follows the user’s preferences. This hierarchical design enables COPILOT to effectively handle multi-task prompts without additional training, ensuring reliable preference alignment without compromising response quality. The complete algorithm is in §A.1.

4.1 Scenario Planning

The scenario planning module first decomposes complex, multi-intent user prompts into granular sub-tasks that align with the preference configuration $\mathcal{C}$. It outputs a list of sub-tasks, each annotated with a set of *preference anchors*. Specifically, leveraging the in-context learning and reasoning capabilities of LLMs demonstrated in prior work (Wang et al., 2023a; Shen et al., 2023), COPILOT employs a prompting method to guide the model to parse the original input prompt into a sequence of independent sub-tasks, extracting the corresponding description for each sub-task from the original prompt. This decomposition process is grounded in the user’s preference configuration. Note that COPILOT leverages the target model θ_l to add the description of each scenario, avoiding introducing

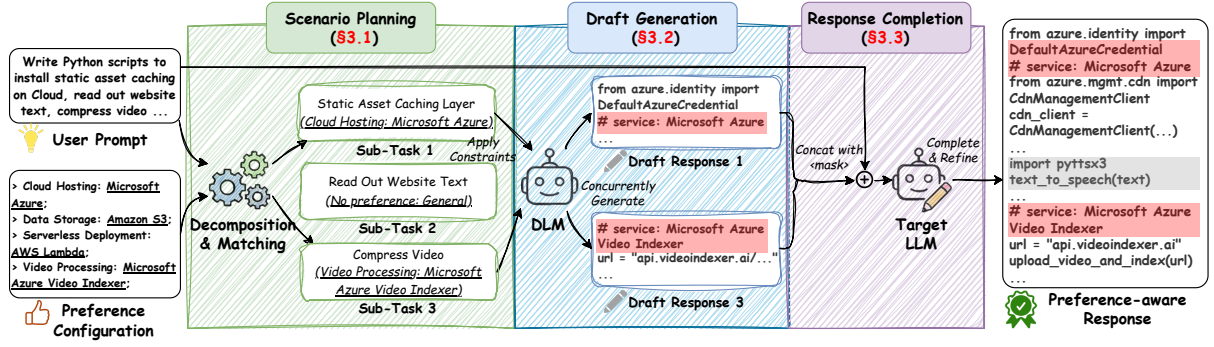


Fig. 2: The workflow of COPILOT. (COPILOT first decomposes the user prompt into sub-tasks and matches each to a scenario in the preference configuration when one applies (e.g., ‘Cloud Hosting’ and ‘Video Processing’), while sub-tasks without a matching scenario are marked as general (e.g., ‘Read out website text’). For each matched sub-task, COPILOT constructs hard constraints and uses a DLM to generate draft responses containing the preferred services in parallel (i.e., Microsoft Azure and Microsoft Azure Video Indexer in red). Finally, COPILOT concatenates the drafts with <mask> tokens, queries the target LLM to complete the missing details (including the subtask with no preference, marked in grey), and returns a high-quality response that matches the user preferences.)

extra sub-tasks in the decomposition process.

Once decomposition is finished, the sub-tasks that semantically match a specific scenario in $\mathcal{C}$ are explicitly tagged with their corresponding preferred services. For each matched sub-task, COPILOT further queries the target LLM to generate a set of preference anchors (e.g., preferred service name and representative API snippets). These anchors are jointly forwarded to the draft generation module to steer the DLM toward the preferred service. Conversely, if a sub-task within the input prompt does not correspond to any defined preference scenario, COPILOT marks it as a general task and does not constrain it in the subsequent generation phase, so that COPILOT will not carry preferences for these parts of the request, and the general utility of the target LLM is preserved.

4.2 Draft Generation

As mentioned in §2, unlike classic LLMs that generate tokens sequentially from left to right, DLMs generate text by iteratively denoising and refining a sequence of noise or masked tokens into coherent text (Li et al., 2022; Han et al., 2023). This bidirectional nature allows DLMs to perform text in-filling, making them ideal for embedding preferences during decoding.

To efficiently generate draft responses that include user-preferred services for each decomposed sub-task, COPILOT leverages the diffusion decoding process of DLMs in this module. Specifically, before the decoding process begins, COPILOT initializes the masked token sequence (e.g., X^0 in Fig. 3) with the preference anchors set produced by the planning module at distinct positions in the sequence. COPILOT marks these anchor tokens as fixed, so the DLM preserves them throughout

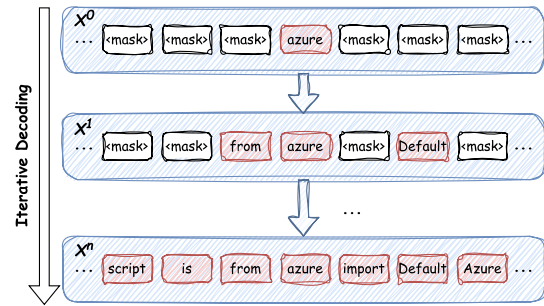


Fig. 3: A demo of iterative decoding process of DLM. Starting from an initial masked token sequence with a fixed token in the middle (i.e., X^0), DLM gradually refines the sequence over multiple denoising steps until generating the final output (i.e., X^n). Red marks the denoised tokens.

denoising while only filling the surrounding mask tokens. During the subsequent denoising steps, the DLM iteratively generates context around these anchored tokens until meeting the stopping criteria, yielding a compact skeleton of the response, such as the structural backbone of a code snippet or the topical outline of a textual answer.

By distributing the preference anchors across several concrete textual positions before full response generation begins, this multi-anchor strategy gives the constraint multiple footholds within the draft and thereby prevents the subsequent completion module from drifting away from the configured preference. By contrast, an autoregressive baseline would have to introduce each anchor at a separate point along a left-to-right trajectory, and the surrounding fluency pressures can easily dilute or override such one-shot commitments (Li et al., 2022). Note that this module aims to effectively generate draft responses that follow the user preferences, and the details and quality of responses will be completed and refined by querying the target LLM in the next module.

4.3 Response Completion

Although the draft responses generated in the previous module contain the correct preference anchors, they may lack semantic fluency and detailed context due to the limited generation capability of the lightweight DLM. Therefore, the response completion module leverages the strong generation capabilities of the target LLM to refine and complete the final response.

Specifically, COPILOT concatenates the draft responses in the sequential order of the sub-tasks and inserts mask tokens between adjacent drafts as explicit slots for missing details. We then construct a composite prompt that feeds the user’s original prompt and the merged draft sequence into the target LLM θ_l , querying the model to replace the mask tokens with response details while preserving the draft contents. This is a classic text completion task for modern LLMs (Donahue et al., 2020; Du et al., 2022), where the drafts serve as a skeleton that outlines the key components of the response, and the masked positions leave room for the LLM to fill in the supporting details and ensure fluency.

This module mirrors the *reviewing* stage of the cognitive process model (Flower and Hayes, 1981), in which the target LLM does not start from scratch but treats the merged drafts as a scaffold and integrates the sub-tasks into a coherent final answer. This decoupled design ensures that the final recommendation results maintain high fluency and quality, without compromising the strict preference alignment established in the drafting module.

5 Experiments

5.1 Setup

Datasets. We evaluate COPILOT on the code and text domains, with one dataset per domain. The code dataset contains prompts adapted from prior work (Zhang et al., 2025) and covers 15 programming scenarios (e.g., cloud hosting and data analysis). The text dataset is curated from public dialogue and recommendation corpora (Budzianowski et al., 2018; Rastogi et al., 2020) and covers 5 everyday recommendation scenarios. Each dataset contains both single-task prompts and multi-task prompts that concatenate up to three sub-tasks.

Our experiments finally include a total of 407 single-task prompts (292 for code and 115 for text) and 1,390 multi-task prompts (650 for code and 740 for text). In addition, to simulate the real-world setting where pre-registered preferences may not

strictly align with every prompt, we generate three preference configurations for each prompt. Each configuration is composed of several verified services and mixes scenarios covered by the prompt with distractor scenarios that are absent from it. Therefore, the methods in experiments must distinguish requested preferences from unrelated ones rather than blindly invoke every registered entry. Pairing each prompt with verified preference configurations yields 5,391 cases, of which 1,221 are single-task cases and 4,170 are multi-task cases. More details are in §A.2.1.

Models. We conduct experiments on three models (i.e., θ_l) that span different open- and closed-source families, with Qwen3-32B (Yang et al., 2025) (hereafter Qwen) and Gemma-3-12B (Team et al., 2025) (hereafter Gemma) as the open-source backbone models and Gemini 3.1 Flash-Lite (hereafter Gemini) as the closed-source model in experiments.

Metrics. Our experiment adopts four metrics that comprehensively evaluate the preference adherence and response quality. Metric details are in §A.2.1.

- *Preference adherence.* Following the prior work (Yun et al., 2025), we report two metrics at different granularities. *Prompt-level Accuracy* (PA) is the fraction of cases where the response satisfies *all* preferences in the prompt, while *Instruction-level Accuracy* (IA) is the fraction of individual preferences (across all cases) that are satisfied. PA measures strict per-case adherence, and IA calculates the adherence at the sub-task level.

- *Response quality.* We report a *Distractor Rate* (DR), the rate at which a response mentions a service from a distractor scenario, i.e., a scenario in $\mathcal{C}$ that is not requested by the current prompt. A low DR means the response consistently focuses on what the user requested, while a high DR means the response also activates distractor entries by listing extra providers without going into detail about the code or recommendation, which can lead to an abnormally high adherence score. Following prior work (Zheng et al., 2023), we introduce *Quality Score* (QS) as an auxiliary metric for response quality assessment. It leverages ‘GPT-5.1-2025-11-13’ model to rate the response on a ‘1-to-10’ scale.

Baselines. We compare COPILOT against four representative training-free baselines (adapted to the provider preference task) that cover the prompting and decoding strategies most commonly used for preference control. Note that our experiment excludes training-based methods since they require heavy training on manual generated dataset for

each preference configuration, which is impractical for the dynamic provider preference setting in this work. ❶ *Zero-shot* prompting appends the full preference configuration to the user prompt and queries the target LLM in a single turn (Yun et al., 2025). ❷ *Grouped* prompting reorganizes the configuration into structured constraint groups before the same single-turn query, which shortens the prompt and reduces cognitive load on the model (Yun et al., 2025). ❸ *Step-wise* prompting partitions the preferences across multiple dialogue turns and elicits the response incrementally, so that each turn focuses on a smaller constraint subset (Yun et al., 2025). ❹ *Constrained* method applies the design of controlled decoding (Lu et al., 2021), expressing each preference as a lexical constraint and biasing token selection toward sequences that satisfy these constraints. Since the Gemini model does not provide logits, our experiments on Gemini do not use constrained decoding that depends on logits.

Implementation. All local experiments run on a Rocky Linux 9.7 server with four NVIDIA L40 GPUs under Python 3.9, and we query Gemini through its public API with the same prompts and preference configurations. COPILOT uses LLaDA-1.5 (Nie et al., 2025) as the DLM in experiments. To improve robustness, we query each case three times and report the mean and standard deviation of each metric. Additional anchor-injection details, prompt templates, and DLM settings are in §A.2.1.

5.2 Effectiveness of COPILOT

Experiment Design. To study the effectiveness of COPILOT in controlling provider preference, we evaluate COPILOT against four prompting and decoding baselines (*Zero-shot*, *Grouped*, *Step-wise*, and *Constrained*) on the full evaluation dataset described in §5.1, which covers single-task and multi-task prompts across the code and text domains. Table 1 reports the main comparison, where each row corresponds to the results of different methods on different models.

Analysis of Effectiveness. As shown in Table 1, COPILOT achieves the best preference control performance among all methods across three target LLMs. Specifically, COPILOT simultaneously attains high PA, IA, and QS together with low DR, indicating that it can precisely generate high-quality responses containing user-preferred services without introducing irrelevant distractor items. The advantage is particularly pronounced in multi-task settings, where COPILOT improves PA by up to

58.9% and reduces DR by 74.0% compared with the strongest baseline ‘*Grouped*’ (Gemma model and multi-task code dataset).

On the code dataset, COPILOT leads PA and IA across all three target LLMs, and this advantage widens as the prompts shift from single-task to multi-task settings. For instance, on the Qwen model with multi-task prompts, COPILOT reaches 80.3% PA while the second best method ‘*Grouped*’ attains only 60.9%, and on the Gemini model, COPILOT achieves 95.8% PA against 69.3% for the second best method ‘*Step-wise*’. Our analysis indicates that this performance gap is attributed to the precise nature of code generation tasks, where the included service has to surface through a concrete API call or import statement. Once a single prompt mixes multiple sub-tasks, prompting methods easily drop or substitute a preferred provider in the reasoning chain, and the constrained decoding method lacks the global view needed to decide where a service should appear inside a long code skeleton. In contrast, COPILOT embeds preferred services as preference anchors in DLM decoding, so the generated draft can fix the target services before the downstream LLM fleshes out the surrounding code, resulting in strong adherence and low distractor activation.

For the text domain, most baselines push PA and IA close to 100%, which looks like flawless preference control. However, our further analysis reveals that the baselines reach such adherence only by mechanically listing every service in the preference configuration, regardless of whether the user actually requested it. For example, on the Qwen model with single-task text prompts, ‘*Zero-shot*’ achieves the DR of 99.4%, and ‘*Constrained*’ achieves 97.1%. In contrast, COPILOT activates only the preferred services that match the current sub-tasks while leaving distractor scenarios untouched. It gives up a marginal 1.2% in PA under this setting, but earns a steep drop of more than 87.8% in DR. This behavior follows the demands of real applications. As discussed in §2, the platform could register a large pool of preference entries while any single prompt touches only a small subset, so a recommender that activates every service would clutter the response and may breach compliance policies that scope preferences to specific scenarios.

Table 1: Comparison between COPILLOT and baselines across code and text dataset. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable. Bold indicates the best result, italic with underline indicates the second best result, and gray marks the standard deviation.)

Model	Method	Single-task Code				Multi-task Code				Single-task Text				Multi-task Text			
		PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$	QS $\uparrow$	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$	QS $\uparrow$	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$	QS $\uparrow$	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$	QS $\uparrow$
Qwen	Zero-shot	41.7 (0.9)	41.7 (0.9)	45.8 (1.3)	3.6 (2.9)	56.9 (0.1)	73.3 (0.2)	65.6 (0.7)	5.5 (1.3)	99.9 (0.0)	99.9 (0.0)	99.4 (0.0)	7.9 (1.5)	100.0 (0.0)	100.0 (0.0)	98.0 (0.0)	7.3 (1.3)
	Grouped	<u>70.5 (1.2)</u>	<u>70.5 (1.2)</u>	70.3 (1.6)	4.5 (1.3)	<u>60.9 (0.7)</u>	<u>74.8 (0.5)</u>	57.7 (1.1)	4.8 (1.2)	100.0 (0.0)	100.0 (0.0)	<u>59.1 (12.5)</u>	<u>8.8 (1.2)</u>	<u>99.9 (0.0)</u>	100.0 (0.0)	<u>75.7 (2.8)</u>	7.6 (1.5)
	Step-wise	18.9 (0.8)	18.9 (0.8)	<u>17.6 (1.0)</u>	3.9 (1.0)	15.5 (0.5)	23.3 (0.5)	16.9 (1.5)	4.2 (1.2)	100.0 (0.0)	100.0 (0.0)	92.9 (0.0)	6.5 (0.9)	100.0 (0.0)	100.0 (0.0)	90.0 (0.8)	6.7 (1.6)
	Constrained	69.9 (1.1)	69.9 (1.1)	74.3 (1.5)	<u>5.1 (1.2)</u>	56.7 (0.8)	73.1 (0.6)	65.5 (1.8)	<u>5.6 (1.5)</u>	100.0 (0.0)	100.0 (0.0)	97.1 (0.0)	7.6 (2.3)	100.0 (0.0)	100.0 (0.0)	96.7 (0.0)	<u>7.4 (1.9)</u>
	COPILLOT	78.5 (0.6)	78.5 (0.6)	15.5 (0.4)	5.4 (1.4)	80.3 (1.4)	85.3 (1.1)	<u>19.0 (0.1)</u>	5.8 (1.5)	98.8 (1.6)	98.8 (1.6)	7.2 (1.6)	9.5 (0.5)	98.6 (1.4)	<u>99.3 (0.7)</u>	39.8 (0.8)	7.3 (1.3)
Gemma	Zero-shot	72.0 (2.0)	72.0 (2.0)	72.5 (1.3)	<u>4.1 (1.1)</u>	43.6 (0.4)	62.3 (0.4)	53.5 (0.1)	4.5 (1.2)	100.0 (0.0)	100.0 (0.0)	98.3 (1.7)	7.9 (1.1)	100.0 (0.0)	100.0 (0.0)	97.9 (0.9)	<u>7.5 (1.7)</u>
	Grouped	70.1 (1.0)	70.1 (1.0)	75.4 (1.3)	<u>3.7 (0.7)</u>	<u>53.5 (0.7)</u>	<u>70.6 (0.4)</u>	70.5 (0.6)	3.4 (0.9)	100.0 (0.0)	100.0 (0.0)	<u>29.0 (4.3)</u>	8.9 (1.0)	<u>99.9 (0.2)</u>	100.0 (0.1)	<u>67.2 (2.7)</u>	7.8 (1.2)
	Step-wise	37.3 (0.9)	37.3 (0.9)	<u>37.9 (1.6)</u>	3.8 (0.7)	8.4 (0.2)	12.6 (0.3)	10.4 (0.4)	3.7 (0.7)	99.4 (0.5)	99.4 (0.5)	91.3 (0.9)	7.4 (1.5)	<u>99.8 (0.2)</u>	<u>99.9 (0.1)</u>	92.1 (1.7)	6.4 (1.4)
	Constrained	71.2 (1.5)	71.2 (1.5)	73.0 (1.9)	4.0 (1.4)	45.6 (0.9)	63.5 (0.9)	54.5 (0.8)	4.3 (0.9)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	7.8 (1.4)	100.0 (0.0)	100.0 (0.0)	95.8 (0.2)	7.2 (1.3)
	COPILLOT	80.0 (0.5)	80.0 (0.5)	21.7 (1.4)	4.7 (1.6)	85.0 (1.2)	88.8 (0.7)	<u>18.3 (0.5)</u>	4.5 (1.2)	95.4 (3.2)	95.4 (3.2)	25.9 (4.2)	<u>8.3 (1.8)</u>	98.1 (0.8)	99.1 (0.4)	42.6 (0.8)	6.7 (1.4)
Gemini	Zero-shot	56.4 (4.2)	56.4 (4.2)	<u>65.8 (11.0)</u>	4.8 (1.3)	46.1 (2.1)	65.3 (2.6)	<u>66.3 (4.2)</u>	5.4 (1.3)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	8.9 (1.4)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	8.4 (1.5)
	Grouped	<u>64.5 (7.2)</u>	<u>64.5 (7.2)</u>	74.5 (0.0)	4.8 (1.3)	63.4 (3.2)	77.0 (3.3)	73.2 (3.2)	4.8 (1.2)	100.0 (0.0)	100.0 (0.0)	<u>76.4 (2.4)</u>	<u>8.8 (1.3)</u>	100.0 (0.0)	100.0 (0.0)	<u>96.5 (1.2)</u>	7.6 (1.4)
	Step-wise	62.9 (10.5)	62.9 (10.5)	67.5 (4.2)	4.6 (1.6)	<u>69.3 (2.4)</u>	<u>79.2 (2.2)</u>	75.2 (5.2)	<u>5.3 (1.2)</u>	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	7.6 (1.2)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	7.4 (1.1)
	COPILLOT	95.8 (2.4)	95.8 (2.4)	12.5 (3.9)	4.4 (1.7)	95.8 (1.7)	95.8 (1.7)	25.0 (3.5)	4.6 (0.8)	100.0 (0.0)	100.0 (0.0)	18.1 (2.4)	<u>8.8 (0.9)</u>	100.0 (0.0)	100.0 (0.0)	26.4 (2.4)	<u>8.2 (1.0)</u>

Table 2: Ablation study results of COPILLOT on 100 cases of the code dataset on the Qwen model. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable. Bold indicates the best result, and italic with underline indicates the second best result.)

Variant	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$	QS $\uparrow$
COPILLOT	82.6	<u>83.5</u>	19.3	6.0
- w/o Scenario Planning	50.7	63.2	71.0	5.3
- w/o Draft Generation	69.3	78.3	<u>20.3</u>	5.4
- w/o Response Completion	<u>72.0</u>	84.8	23.7	<u>5.7</u>

5.3 Ablation study

Experiment Design. To understand how each module contributes to COPILLOT’s performance, we conduct ablation experiments on 100 cases randomly sampled from the code dataset, using the Qwen model as the target LLM. The experiments include three variants, while each variant disables a single module in COPILLOT while keeping all other modules unchanged. We then compare every variant with the full system of COPILLOT. The three variants work as follows. ❶ ‘- w/o Scenario Planning’ replaces the LLM-based intent planning with a coarse punctuation-based segmentation. The target LLM then matches a preference to each segment, and the labeled segments enter the draft and completion modules. ❷ ‘- w/o Draft Generation’ skips the DLM draft step and gives the planned sub-tasks and matched preferences directly to the target LLM, which then produces the final response. ❸ ‘- w/o Response Completion’ returns the raw DLM draft to the user as the final answer without further refinement. Table 2 reports the ablation results.

Analysis of Module Contribution. As shown in Table 2, removing any single module degrades the overall performance of COPILLOT, which demonstrates that each module makes a clear contribution to preference control and response quality. Specifically, removing the scenario planning module triggers a catastrophic drop across every metric,

while PA falls from 82.6% to 50.7%, IA drops from 83.5% to 63.2%, and DR rises sharply from 19.3% to 71.0%. Our analysis indicates that without precise task decomposition and scenario matching, the system fails to navigate complex prompts with multiple sub-tasks. It will activate many irrelevant preferences by mistake, which leads to a high DR, and misses what the user preferences actually ask for. Additionally, draft generation provides the skeleton that ensures preferences are included. After removing the draft module, although the system’s DR remained relatively low, the PA dropped substantially from 82.6% to 69.3%. This further illustrates that simply prompting the target LLM about the preferences it needs to satisfy is insufficient, and LLMs are prone to forgetting these constraints in the generation of long and complex responses. Furthermore, although removing the response completion module does not lead to a significant decrease in preference adherence, it results in the generation of rough responses, reducing the quality and usability of the recommendation results. Notably, this variant attains a slightly higher IA (84.8% vs. 83.5%), since the raw skeleton preserves the fixed anchors verbatim. However, such verbatim retention in an unfinished skeleton comes at the cost of lower PA, higher DR, and lower QS, reflecting a trade-off between literal anchor retention and end-to-end control quality rather than a stronger preference control. Additional ablation results (e.g., the text-domain ablation) are in §A.2.2.

6 Discussion

Implications. COPILLOT provides a practical solution for LLM recommenders that accurately adheres to users’ fine-grained commercial choices. Beyond its immediate technical performance, our framework has broader implications at three levels.

❶ For end users and their organizations, COPILLOT

LOT reduces the burden of repeatedly correcting responses that ignore the established tech stack or a service provider, improving the efficiency in human-AI collaboration scenarios. It also enables organizations to enforce procurement, security, and compliance policies through a declarative configuration $\mathcal{C}$ rather than through brittle prompts. ② For downstream applications, the training-free design lowers the integration cost of provider-aware automation. Therefore, simply applying different configurations can govern different agents, IDE assistants, and enterprise copilots without retraining the target model. ③ For the ecosystem, reliable preference control opens a new commercial layer for LLM recommendation. The LLM platform could populate a default $\mathcal{C}$ for users who never upload one, turning the configuration itself into a controllable surface for promotion or partner placement. Although this capability necessitates careful consideration of transparency and fairness risks, it outlines a plausible and highly scalable blueprint for monetizing LLM recommendations, potentially yielding substantial benefits in the era when LLMs are replacing traditional recommendation systems. Finally, although this paper instantiates the framework on provider preferences, a realistic scenario with clear commercial value, the underlying mechanism is not tied to providers. Other constraints that are registered in advance and activated on demand (e.g., organization-specific terminology or mandated tool chains) can be controlled in the same plug-and-play manner, which we view as an extra benefit of COPILOT.

Efficiency and Applicability. The stronger preference control of COPILOT is achieved by the additional planning and drafting stages, whose overhead remains modest in practice. As detailed in §A.2.6, the median end-to-end latency of COPILOT on the code dataset (224.4s per output on the Qwen model) falls within the range of the baselines (from 168.4 to 237.1s), since the planning and drafting stages account for only a small fraction of the latency (15.0s and 15.7s, versus 190.6s for the final completion), and the lightweight DLM only requires an additional 15 GB checkpoint. Moreover, on the Gemini models, the target-model API charge of COPILOT is even slightly lower than those of the prompting baselines (Table 11). Note that the DLM does not need to match the scale of the target model, since COPILOT only borrows its masked decoding mechanism to embed the matched preferences into a short draft (e.g., 128 tokens), while the

fluency and details of the final response are handled by the target LLM. Exploring how the choice of DLM backbones affects the performance is one potential future work (§A.3). Balancing the overhead and the preference control performance, we believe that COPILOT is best suited for the multi-intent and distractor-heavy prompts that are common in personalized LLM recommendation applications, where simple prompting methods often obfuscate or overactivate preferences (§5.2). For simple user prompts with few registered preferences, falling back to lightweight prompting methods remains a reasonable choice.

7 Conclusion

In this paper, we present COPILOT, a training-free framework for provider preference control in LLM recommendation. COPILOT splits each user prompt into sub-tasks based on the given preference configuration, drafts sub-task responses with a DLM under hard preference anchors, and queries the target LLM to complete the fluent and high-quality answer. Experiments on the code and text datasets show that COPILOT raises the provider preference adherence effect by up to 58.9% compared to baselines while maintaining recommendation quality. We urge the community to recognize fine-grained provider preferences as a valuable objective for LLM alignment and to apply inference-time control methods to explore new commercial opportunities in LLM recommendation.

Limitation

This study aims to design and evaluate a training-free framework for provider preference control in LLM recommendations. Although our dataset contains 1,797 prompts and 5,391 cases (i.e., prompt-configuration pairs) that span 20 scenarios across code and text domains, it still cannot exhaust the prompt distribution observed in real-world LLM recommendations, where user prompts can be longer, noisier, and span more tasks than the three sub-tasks we adopt in the synthesized multi-task set. In addition, whether users upload the preference configuration directly or the platform aggregates it from interaction history, real-world deployment can introduce noise such as overlapping scenario boundaries and ambiguous service descriptions. Regarding the baselines, since provider preferences are naturally expressed as lexical constraints (i.e., service names and APIs), our experiments adopt a lexically constrained decoding pipeline as the decoding-time baseline (§5.1). Other decoding-time alignment methods discussed in §2 typically require a trained reward model or classifier for each preference dimension, which conflicts with the training-free and dynamically changing setting of this work, and we leave the comparison with them under matched training budgets to future work. Moreover, all prompts in our benchmark are in English, and evaluating COPILOT on multilingual prompts, as well as on longer and noisier requests with overlapping or conflicting provider preferences, is an important direction for future work. Note that the purpose of this study is not to provide a universal benchmark for provider preference, but to demonstrate a feasible inference-time control framework. We have compared COPILOT and all baselines on the same prompts and settings. However, the above limitations may still prevent the experimental conclusions from fully holding to other scenarios and tasks. To mitigate these limitations and support the reproduction of future research, we have publicly released the prototype of COPILOT, the dataset, and the necessary experimental results.

Ethical Considerations

This paper presents a training-free framework for controlling provider preference in LLM recommendation, and we do not foresee direct ethical risks introduced by the technique itself. Our experiments use publicly released LLMs (Qwen3-

32B, Gemma-3-12B, and Gemini 3.1 Flash-Lite), the open-source DLM LLaDA-1.5, and prompts adapted from publicly available recommendation datasets (Zhang et al., 2025; Budzianowski et al., 2018; Rastogi et al., 2020), none of which contain personal or sensitive information. The study also does not raise IRB-level concerns. We acknowledge that reliable preference control could be a double-edged blade in applications, since a recommendation platform could populate the preference configuration with promoted or partner services rather than honoring user-uploaded preferences, as we discussed in §6. Mitigating such risks calls for regulation and oversight that ensure the transparency and reliability of the provenance of the preference configuration. This paper focuses on the inference-time technique that controls preferences during generation, which is orthogonal to the above risks arising from the deployment.

Acknowledgments

The authors thank the anonymous reviewers for their insightful feedback and constructive comments. This work was supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM114), the National Natural Science Foundation of China (62521002, U24B20185, 62376210, 62132011). Thanks to the New Cornerstone Science Foundation and the Xplorer Prize. This work was also partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). This research is also part of the IN-CYPHER Programme and is supported by the National Research Foundation, Prime Minister’s Office, Singapore, under its Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

References

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. Multiwoz-a large-scale multi-domain wizard-of-oz dataset for task-oriented dialogue modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural*

- Language Processing*. Association for Computational Linguistics.
- Yuanpu Cao, Tianrong Zhang, Bochuan Cao, Ziyi Yin, Lu Lin, Fenglong Ma, and Jinghui Chen. 2024. Personalized steering of large language models: Versatile steering vectors through bi-directional preference optimization. *Advances in Neural Information Processing Systems*, 37:49519–49551.
- Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the effectiveness of llm-as-a-judge for code generation and summarization. *IEEE Transactions on Software Engineering*.
- Yashar Deldjoo. 2024. Understanding biases in chatgpt-based recommender systems: Provider fairness, temporal stability, and recency. *ACM Transactions on Recommender Systems*.
- Haikang Deng and Colin Raffel. 2023. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11781–11791.
- Chris Donahue, Mina Lee, and Percy Liang. 2020. Enabling language models to fill in the blanks. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 2492–2501.
- Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335.
- Linda Flower and John R Hayes. 1981. A cognitive process theory of writing. *College Composition & Communication*, 32(4):365–387.
- Ge Gao, Alexey Taymanov, Eduardo Salinas, Paul Mineiro, and Dipendra Misra. 2024. Aligning llm agents by learning latent preference from user edits. *Advances in Neural Information Processing Systems*, 37:136873–136896.
- Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. 2023. Diffuseq: Sequence to sequence text generation with diffusion models. In *International Conference on Learning Representations (ICLR 2023)*.
- Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. 2023. Ssd-llm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11575–11596.
- Xingwei He. 2021. Parallel refinements for lexically constrained text generation with bart. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8653–8666.
- Zhengfu He, Tianxiang Sun, Qiong Tang, Kuanning Wang, Xuan-Jing Huang, and Xipeng Qiu. 2023. Diffusionbert: Improving generative masked language models with diffusion models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4521–4534.
- Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. 2024. Large language models are zero-shot rankers for recommender systems. In *European Conference on Information Retrieval*, pages 364–381. Springer.
- James Y Huang, Sailik Sengupta, Daniele Bonadiman, Yi-an Lai, Arshit Gupta, Nikolaos Pappas, Saab Mansour, Katrin Kirchhoff, and Dan Roth. 2025. Deal: Decoding-time alignment for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26280–26300.
- Xuancheng Huang, Zijun Liu, Peng Li, Tao Li, Maosong Sun, and Yang Liu. 2023. An extensible plug-and-play method for multi-aspect controllable text generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15233–15256.
- Ronald T. Kellogg. 2008. [Training writing skills: A cognitive developmental perspective](#). *Journal of Writing Research*, 1(1):1–26.
- Maxim Khanov, Jirayu Burapachee, and Yixuan Li. 2024. Args: Alignment as reward-guided search. In *International Conference on Learning Representations*, volume 2024, pages 17755–17773.
- Bolian Li, Yifan Wang, Anamika Lochab, Ananth Grama, and Ruqi Zhang. 2025. Cascade reward sampling for efficient decoding-time alignment. In *Second Conference on Language Modeling*.
- Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. 2022. Diffusion-llm improves controllable text generation. *Advances in neural information processing systems*, 35:4328–4343.
- Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient fine-tuning for llm-based recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pages 365–374.
- Junling Liu, Chao Liu, Peilin Zhou, Renjie Lv, Kang Zhou, and Yan Zhang. 2023a. Is chatgpt a good recommender? a preliminary study. *arXiv preprint arXiv:2304.10149*.
- Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023b. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9):1–35.

- Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Neurologic decoding:(un) supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human language technologies*, pages 4288–4299.
- Hanjia Lyu, Song Jiang, Hanqing Zeng, Yinglong Xia, Qifan Wang, Si Zhang, Ren Chen, Chris Leung, Jiajie Tang, and Jiebo Luo. 2024. Llm-rec: Personalized recommendation via prompting large language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 583–612.
- Sidharth Mudgal, Jong Lee, Harish Ganapathy, YaGuang Li, Tao Wang, Yanping Huang, Zhifeng Chen, Heng-Tze Cheng, Michael Collins, Trevor Strohman, and 1 others. 2024. Controlled decoding from language models. In *Proceedings of the 41st International Conference on Machine Learning*, pages 36486–36503.
- Sheshera Mysore, Mahmood Jasim, Andrew McCallum, and Hamed Zamani. 2023a. Editable user profiles for controllable text recommendations. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 993–1003.
- Sheshera Mysore, Andrew McCallum, and Hamed Zamani. 2023b. Large language model augmented narrative driven recommendations. In *Proceedings of the 17th ACM conference on recommender systems*, pages 777–783.
- Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, JUN ZHOU, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2025. Large language diffusion models. In *ICLR 2025 Workshop on Deep Generative Model in Machine Learning: Theory, Principle and Efficacy*.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.
- Angela Ramirez, Kartik Agarwal, Juraj Juraska, Utkarsh Garg, and Marilyn Walker. 2023. Controllable generation of dialogue acts for dialogue systems via few-shot response generation and ranking. In *Proceedings of the 24th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 355–369.
- Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8689–8696.
- Scott Sanner, Krisztian Balog, Filip Radlinski, Ben Wedin, and Lucas Dixon. 2023. Large language models are competitive near cold-start recommenders for language-and item-based preferences. In *Proceedings of the 17th ACM conference on recommender systems*, pages 890–896.
- Google Search. 2025. Ai in search. <https://search.google/ai-in-search/>.
- Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugging-gpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.
- Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. Codefusion: A pre-trained diffusion model for code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11697–11708.
- Srini S Srinivasan, Rolph Anderson, and Kishore Ponnavolu. 2002. Customer loyalty in e-commerce: an exploration of its antecedents and consequences. *Journal of retailing*, 78(1):41–50.
- Stack Overflow. 2025. [2025 stack overflow developer survey](#).
- Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, and 1 others. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.
- Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051.
- Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 2609–2634.
- Yifan Wang, Weizhi Ma, Min Zhang, Yiqun Liu, and Shaoping Ma. 2023b. A survey on the fairness of recommender systems. *ACM Transactions on Information Systems*, 41(3):1–43.
- Chuhan Wu, Fangzhao Wu, Tao Qi, and Yongfeng Huang. 2021. Empowering news recommendation with pre-trained language models. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, pages 1652–1656.
- Jialin Wu, Kecen Li, Zhicong Huang, Xinfeng Li, Xiaofeng Wang, and Cheng Hong. 2026. Enchtable: Unified safety alignment transfer in fine-tuned large language models. pages 3358–3376.

- Likang Wu, Zhi Zheng, Zhaopeng Qiu, Hao Wang, Hongchao Gu, Tingjia Shen, Chuan Qin, Chen Zhu, Hengshu Zhu, Qi Liu, and 1 others. 2024. A survey on large language models for recommendation. *World Wide Web*, 27(5):60.
- Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and 1 others. 2025. Unleashing the power of generative model in recovering variable names from stripped binary. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Bowen Yang, Cong Han, Yu Li, Lei Zuo, and Zhou Yu. 2022. Improving conversational recommendation systems’ quality with context-aware item meta-information. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 38–48.
- Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. 2023. Diffusion models: A comprehensive survey of methods and applications. *ACM computing surveys*, 56(4):1–39.
- Yejin Yoon, Jungyeon Lee, Kangsan Kim, Chanhee Park, and Taeuk Kim. 2024. Blendx: Complex multi-intent detection with blended patterns. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 2428–2439.
- Longfei Yun, Letian Peng, and Jingbo Shang. 2025. Ultrabench: Benchmarking llms under extreme fine-grained text generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15438–15453.
- An Zhang, Yuxin Chen, Leheng Sheng, Xiang Wang, and Tat-Seng Chua. 2024. On generative agents in recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and development in Information Retrieval*, pages 1807–1817.
- Song Zhang, Nan Zheng, and Danli Wang. 2022. Gbert: Pre-training user representations for ephemeral group recommendation. In *Proceedings of the 31st ACM international conference on information & knowledge management*, pages 2631–2639.
- Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Qian Wang, Chao Shen, and Yang Liu. 2025. [The invisible hand: Unveiling provider bias in large language models for code generation](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2025, Vienna, Austria, July 27 - August 1, 2025, pages 21376–21403. Association for Computational Linguistics.
- Yizhe Zhang, Guoyin Wang, Chunyuan Li, Zhe Gan, Chris Brockett, and William B Dolan. 2020. Pointer: Constrained progressive text generation via insertion-based generative pre-training. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pages 8649–8670.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.
- Thomas P Zollo, Andrew Wei Tung Siah, Naimeng Ye, Ang Li, and Hongseok Namkoong. 2025. Personallm: Tailoring llms to individual preferences. In *The Thirteenth International Conference on Learning Representations*.

A Appendices

The appendices are organized as follows:

- §A.1 provides more implementation details of COPILOT, organized around the scenario planning module (§A.1.1), the draft generation module (§A.1.2), and the response completion module (§A.1.3).
- §A.2 provides additional results and case studies to support our analysis and findings in §5, including our experimental environment (§A.2.1), ablation study (§A.2.2), an analysis of planning accuracy and anchor retention (§A.2.3), case studies (§A.2.4), a mocked-execution validation of the code metrics (§A.2.5), and an efficiency and cost analysis (§A.2.6).
- §A.3 discusses the limitations of this work and the corresponding future directions.

A.1 Additional Implementation Details

A.1.1 Scenario Planning Module

General Prompt. Following the prompt design in prior work (Shen et al., 2023), COPILOT decomposes the user request into a structured JSON list of sub-tasks aligned with the scenarios in the preference configuration $\mathcal{C}$. The prompt strictly enforces the output format, accepts a null *task_category* when no listed scenario semantically matches, and forbids the planner from hallucinating or inventing sub-tasks absent from the user prompt.

Prompt for Scenario Planning

[System]
You perform task parsing on user input, producing a JSON list of tasks. Each task object contains two fields, *task_category*, which is either one available scenario name or null, and *task_prompt*, which is a concise description of the sub-task.
The output must be valid JSON without extra text. Each *task_category* must be exactly one of the scenarios listed in the current preference configuration, or null if no scenario clearly matches the sub-task. The *task_prompt* should be concise and specific. The model should not invent tasks that are not present in the user prompt.
[User]
{user_prompt}

Domain-specific Extensions. Relying solely on a shared planner prompt is insufficiently precise, because identical scenario names can imply fundamentally different intents across code generation and text recommendation domains. Therefore, COPILOT augments the shared prompt with a short domain-specific extension that clarifies the semantics of the scenario list and dictates the handling

of unmatched sub-tasks. The code extension emphasizes concrete programming or cloud-service intents and instructs the planner to return a null *task_category* whenever an implementation-related sub-task is not represented in $\mathcal{C}$. The text extension asks the planner to first infer the semantic type of each sub-task, assigning a non-null category only when it directly matches a scenario in $\mathcal{C}$.

Code-domain Planning Extension

[System Extension]
The available scenarios describe programming and cloud-service tasks. Use the scenario descriptions to match each sub-task to the closest available code scenario, such as hosting, storage, database, analytics, machine learning, media processing, or text-to-speech. Assign a non-null *task_category* only when the user request clearly asks for that code scenario. If a sub-task is implementation-related but does not match any available scenario in the current preference configuration, set *task_category* to null. Do not create extra sub-tasks only to satisfy preference entries.

Text-domain Planning Extension

[System Extension]
The available scenarios describe travel and recommendation tasks, such as *taxi*, *train*, *hotel*, *restaurant*, and *attraction*. Before assigning a category, infer the semantic type of each sub-task. Assign a non-null *task_category* only when the semantic type directly matches one of the scenarios in the current preference configuration. If the matching scenario is absent, set *task_category* to null rather than borrowing a broad or unrelated service. A restaurant or hotel name may simply be a destination, landmark, or existing reservation; classify it as *restaurant* or *hotel* only when the user asks to find, compare, book, review, or manage that type of service.

The three aforementioned components form a layered planning interface that effectively decouples schema control from domain knowledge. Specifically, the shared prompt dictates the output schema and constrains the planner to strictly decompose tasks present in the user request. The code extension focuses the planner on programming and cloud-service intents. The text extension restricts it to travel and recommendation intents grounded in contextual semantic roles. Such a design ensures that the downstream draft generator receives a compact list of preference-anchored sub-tasks, while unmatched sub-tasks can safely bypass forced and unrelated provider choices. The selection between the two extensions, along with the matching anchor template and completion instruction, is a deterministic attribute of the preference configuration $\mathcal{C}$ registered by users or the LLM platform, elim-

Algorithm 1 COPILOT Framework

Input: P – the user prompt;
 $\mathcal{C}$ – the user preference configuration;
 θ_d – DLM;
 θ_l – the target LLM;
Output: R – the final preference-aligned response;

```
1: procedure PREFERENCECONTROL( $P, \mathcal{C}, \theta_d, \theta_l$ )
2:   % Phase 1: Task Planning (§4.1)
3:    $\mathcal{T} \leftarrow \text{decomposePrompt}(P, \theta_t)$                                 ▷ Decompose user prompt into a set of sub-tasks
4:    $M \leftarrow \emptyset$ 
5:   for  $t_i \in \mathcal{T}$  do
6:      $s_i, v_i \leftarrow \text{matchScenario}(t_i, \mathcal{C})$                             ▷ Match scenario and preferred service from configuration
7:      $M \leftarrow M \cup \{(t_i, v_i)\}$                                     ▷ Tag sub-task with service preference (if have)
8:   % Phase 2: Draft Generation (§4.2)
9:    $D \leftarrow \emptyset$ 
10:  for  $(t_i, v_i) \in M$  do in parallel                                ▷ Leverage DLM’s parallel decoding capability
11:     $X^0 \leftarrow \text{initialMask}(v_i)$                                     ▷ Initialize masked token sequence with preferred service (if have)
12:     $d_i \leftarrow \text{dlmDenoise}(X^0, t_i, \theta_d)$                         ▷ Generate draft via iterative denoising
13:     $D \leftarrow D \cup \{d_i\}$ 
14:  % Phase 3: Response Completion (§4.3)
15:   $D_c \leftarrow \text{concatenateDrafts}(D)$                                 ▷ Merge drafts in sequential order
16:   $P_r \leftarrow \text{constructCompositePrompt}(P, D_c)$ 
17:   $R \leftarrow \text{completeResponse}(P_r, \theta_l)$                             ▷ Target LLM refines the drafts into fluent response
18:  return  $R$ 
```

inating the need for additional runtime overhead. Concretely, if $\mathcal{C}$ contains any scenario not related to the code domain, COPILOT automatically loads the text-domain extension, anchor template, and completion instruction. Otherwise, it falls back to the code-domain extensions.

A.1.2 Draft Generation Module

Within the draft generation module, COPILOT employs LLaDA-1.5 (Nie et al., 2025), a lightweight diffusion language model, to efficiently generate draft responses that intrinsically embed the user-preferred services. For each preference-bound sub-task, COPILOT constructs a masked generation region following the DLM prompt and injects fixed anchor tokens into that region prior to the denoising phase. Each anchor encapsulates the preferred service alongside a short scenario or API hint, while the surrounding positions are initialized with mask tokens. During the iterative denoising process, the DLM strictly preserves the fixed anchor tokens and synthesizes the rest of the region in parallel. Consequently, the preferred service is firmly grounded in the draft without the risk of competing against an already-fixed left context.

COPILOT utilizes two anchor templates corresponding to the two domain extensions described in §A.1.1. The code template designates the preferred service name as the primary anchor, optionally accompanied by a short API hint. This minimal

constraint is sufficient to bias the draft toward the correct library or cloud service. The text template is longer and structurally hybrid, integrating a semantic scene cue and a service hint alongside the preferred service name. This ensures that the resulting draft accurately reflects the conversational role of each sub-task. The subsequently completed draft is then passed to the target LLM for final response completion. The lengths of the generated region, DLM denoising step counts, and other inference settings used in the main experiments are reported in §A.2.1.

Example Anchor Templates

```
[Code anchor]
[scenario: Cloud Hosting] [preferred service: Microsoft Azure] [API hint: azure.mgmt.cdn]
[Text anchor]
[scenario: hotel] [preferred service: Airbnb] [semantic cue: book a stay] [service hint: accommodation search and booking]
```

A.1.3 Response Completion Module

Composite Completion Prompt. This module leverages a unified prompt designed to guide the target LLM in completing and refining the merged draft response, while preserving the user-preferred services, as shown below. The prompt integrates four critical pieces, namely, the user task description, the matched preferences, the merged draft responses, and the planned sub-tasks. It then instructs

the target model to faithfully preserve the required service choices, fill in missing details, address every planned sub-task, and remove any unresolved draft placeholders.

Prompt for Response Completion

[System]
 You are in the final answer generation stage of a multi-step DLM pipeline.
 Task description: {*user prompt*}
 Required services: {*matched scenario-service preferences*}
 Draft answer: {*DLM draft*}
 Planned subtasks to cover: {*task planning output*}
 Generate the final answer by preserving the required service choices, completing missing details, covering all planned subtasks, and removing unresolved draft placeholders. For code-generation prompts, return a single fenced Python code block when the user asks for Python scripts. For text recommendation prompts, return a clean natural-language plan without code fences.

A.2 Additional Experimental Results

A.2.1 Setup

Metrics Design. Our evaluation adopts a target-versus-distractor setting. For the i -th generation, let C_i denote the preference configuration, where each key is a scenario, and each value is the preferred service for that scenario. Let S_i denote the scenarios requested by the user prompt. We define the applicable preference set as $P_i = S_i \cap \text{dom}(C_i)$, and the distractor preference set as $D_i = \text{dom}(C_i) \setminus S_i$. Thus, a method should satisfy the services in P_i while avoiding unnecessary service mentions from D_i . All generations included in the PA and IA computation have at least one applicable preference. In our evaluation, whether a response satisfies or mentions a service is determined by case-insensitive substring matching, based on a fixed lexicon that covers each service’s name, common aliases, and API or import identifiers. For the code domain, the matching is performed only within the generated code blocks, so a service that is merely mentioned in the surrounding explanations is not treated as satisfied. For the text domain, the matching is performed on the full natural-language response. Note that this lexical rule does not analyze word boundaries or negation, so a service appearing in code comments or negated sentences is still counted as a mention. Similarly, DR only records whether a response mentions any distractor service, without distinguishing how many times it appears. To verify that this simple lexical rule does not bias our conclusions, the mocked-execution experiment

in §A.2.5 further evaluates the responses based only on the runtime SDK-call events of the generated code.

• *Instruction-level and Prompt-level Accuracy.* Following prior work on controlled generation (Yun et al., 2025), we report *Instruction-level Accuracy* (IA) and *Prompt-level Accuracy* (PA). Let $a_{ij} \in \{0, 1\}$ indicate whether the response of generation i correctly satisfies the j -th applicable preference in P_i , and let m be the number of evaluated generations. IA measures the average fraction of applicable preferences that are satisfied:

$$\mathcal{IA} = \frac{1}{m} \sum_{i=1}^m \frac{1}{|P_i|} \sum_{j=1}^{|P_i|} a_{ij}.$$

PA is stricter as it counts a response as correct only when all applicable preferences in the prompt are satisfied:

$$\mathcal{PA} = \frac{1}{m} \sum_{i=1}^m \prod_{j=1}^{|P_i|} a_{ij}.$$

IA captures fine-grained, partial adherence across sub-tasks, while PA measures strict, prompt-level completeness. These two metrics evaluate adherence at complementary granularities, which provides a nuanced understanding of model performance in complex multi-task scenarios.

• *Distractor Rate.* Preference configurations may contain services for scenarios that are not requested by the current prompt. Let $d_i \in \{0, 1\}$ indicate whether the response for generation i mentions at least one service associated with a distractor scenario in D_i . We define the distractor rate (DR) as

$$\mathcal{DR} = \frac{1}{m} \sum_{i=1}^m d_i.$$

Lower DR indicates that a method effectively isolates provider preferences, strictly scoping them only to the scenarios actually requested by the user.

• *Quality Score.* Recently, with the advancement of LLMs’ reasoning and comprehension capabilities, researchers have widely used the LLM-as-a-judge method to evaluate the quality of text and code (Tong and Zhang, 2024; Zheng et al., 2023; Crupi et al., 2025; Xu et al., 2025). Following this line of work, we employ the ‘GPT-5.1-2025-11-13’ model as a method-blind judge. For each evaluated generation, the judge assigns a rating $q_i \in \{1, \dots, 10\}$ based on helpfulness, relevance,

accuracy, fluency, and sufficient detail. We report the average quality score (QS):

$$QS = \frac{1}{m} \sum_{i=1}^m q_i.$$

QS provides an auxiliary but essential quality reference and is always interpreted in conjunction with PA, IA, and DR to ensure that preference alignment does not significantly compromise natural language quality or code utility.

Prompt for Response Quality Evaluation

[System]
Please act as an impartial judge and evaluate the quality of the response provided by an AI assistant to the user question displayed below. Your evaluation should consider factors such as the helpfulness, relevance, accuracy, fluency, and level of detail of the response. When a preference configuration is provided, also consider whether the response uses the requested services without introducing irrelevant services. Begin your evaluation by providing a short explanation. Be as objective as possible. After providing your explanation, please rate the response on a scale of 1 to 10 by strictly following this format, "[[rating]]", for example, "Rating: [[5]]".
[Question]
{question}
[Preference Configuration]
{preference configuration}
[Assistant's Answer]
{answer}

Human Alignment and Cross-Judge Check of QS. Since the reliability of the LLM-as-a-judge protocol may vary across tasks, we further validate the QS metric under our task setting. Two independent, method-blind human annotators score the outputs of 20 code prompt-configuration keys under all five methods (i.e., 100 outputs in total), using a 1-to-10 task-quality rubric aligned with the GPT-based QS dimensions. The human-human agreement is high (Spearman $\rho=0.855$), and the GPT-based QS correlates well with the mean human ratings ($\rho=0.831$). As shown in Table 3, COPILOT receives the highest human rating together with *Constrained*, showing no quality penalty for COPILOT in this sample.

In addition, we conduct a provider-aware cross-judge check on the same outputs with two external LLM judges. DeepSeek-v4-flash ranks *Zero-shot* first and COPILOT second (6.70 vs. 6.45), while Kimi-k2.6 ranks *Constrained* first and COPILOT second (5.55 vs. 5.45). Namely, COPILOT is ranked second under both external judges, while the top-ranked method is judge-dependent. There-

Table 3: Task-specific human-alignment results of the QS metric on 100 outputs (20 code prompt-configuration keys under five methods) on the Qwen model. (Bold indicates the best result in each column.)

Method	GPT QS [†]	Human QS [†]
COPILOT	6.10	4.58
<i>Constrained</i>	5.90	4.58
<i>Zero-shot</i>	5.85	4.50
<i>Grouped</i>	4.75	3.90
<i>Step-wise</i>	3.90	3.00

fore, we treat QS as an auxiliary quality measure and base the main preference-control conclusions of this paper on PA, IA, and DR.

Dataset Construction. To comprehensively evaluate the effectiveness of COPILOT across different domains, we construct a diverse benchmark covering both the code and text domains. The dataset construction process consists of two phases, namely source prompt curation and preference configuration generation.

• **Source Prompt Curation.** For the code domain, we adopt the dataset released in prior work (Zhang et al., 2025). We extract a subset with a total of 292 single-task prompts covering 15 distinct scenarios, which represent programming scenarios such as web development, data analysis, and machine learning. This curation strategy reliably mirrors the real-world requirements of developers across diverse technical scenarios. For the text domain, we curate a dataset from public dialogue and recommendation datasets (Budzianowski et al., 2018; Rastogi et al., 2020), involving 115 single-task prompts spanning 5 distinct recommendation scenarios. These scenarios are chosen to test the capability to handle everyday decision-making tasks.

To further evaluate preference control effectiveness on multi-task prompts, we query the LLM (i.e., GPT-5.1) with selected single-task prompts to construct a multi-task set based on the prompting framework in prior work (Yoon et al., 2024). To avoid an intractable combinatorial explosion of synthesized multi-task prompts, we restrict the concatenation to a maximum of three single-task prompts from different scenarios. In this process, the retained multi-task prompts cover four coding scenarios (*Cloud Hosting*, *Data Analysis*, *Text-to-Speech*, and *Video Processing*) and four text scenarios (*hotel*, *restaurant*, *taxi*, and *train*), ultimately yielding 1,390 multi-task prompts (650 for code and 740 for text). In addition, we randomly sample 100 prompts and invite two experts with at least five years of experience in both AI and software

engineering (SE) fields to conduct a manual evaluation. Both experts confirm that all LLM-combined prompts faithfully align with the semantics and tasks of the original single-task prompts.

The following prompt is used only during dataset construction to combine multiple single-task prompts into a natural multi-task prompt.

Prompt for Multi-task Prompt Synthesis

[System]
You are a native English speaker. Combine the following sentences as one paragraph naturally.

[User]
{single-task prompt 1}
{single-task prompt 2}
{single-task prompt 3, optional}

• **Preference Configuration Generation.** To ground our benchmark in realistic commercial constraints, we construct the preference configurations by expanding upon open-sourced data from existing work (Zhang et al., 2025; Budzianowski et al., 2018; Rastogi et al., 2020). We first collect and verify the services and providers for each task scenario in our dataset, ensuring that each scenario has at least two corresponding real services. Specifically, we invite two experts with at least five years of experience in the SE field to manually search and verify the services for each scenario, ensuring the reliability and relevance of the selected services. For any inconsistencies, a third expert is consulted to make the final decision. We ultimately established 45 scenario-service entries for the code domain and 15 services for the text domain. For example, in the *machine learning deployment* scenario of the code domain, we select services such as Amazon SageMaker, Google AI Platform, and Microsoft Azure Machine Learning. Then, for single-task prompts, we create cases by pairing each prompt with verified preference configurations, leading to a total of 1,221 single-task cases (876 for code and 345 for text). For multi-task prompts, exhaustively enumerating all provider combinations would result in an evaluation space of exponential scale. Therefore, we randomly sample one target service as the preferred service for each sub-task to form a preference configuration and repeat this process three times for each prompt to ensure statistical diversity. This results in 4,170 multi-task cases (1,950 for code and 2,220 for text). Overall, the benchmark contains 1,797 prompts and 5,391 cases (i.e., prompt-configuration pairs); with three repeated generations per case, the final

evaluation contains 16,173 responses from these generations.

Example of the Single-task Prompt and Preference Configuration

[Prompt]
Book for 1 person for 4 nights starting from Saturday. If 4 nights is not available, how about 2 nights? Give me a step by step plan by using a certain tool.

[Configuration]
{taxi: Addison Lee, hotel: Airbnb, attraction: Tripadvisor}

[Main Scenario]
hotel

Example of the Multi-task Prompt and Preference Configuration

[Prompt]
Write Python scripts to: install a caching layer in front of static assets to distribute them across multiple geographic locations; read out text content from websites and documents to aid visually impaired users; and reduce video file size while maintaining acceptable quality.

[Configuration]
{Video Processing: Microsoft Azure Video Indexer, Data Storage: Amazon S3, Serverless Deployment: AWS Lambda, Cloud Hosting: Microsoft Azure}

[Main Scenarios]
Cloud Hosting, Text-to-Speech, Video Processing

Baseline Implementation Details. This subsection complements the baseline descriptions in §5.1 by reporting implementation choices that are not visible from the main text. All baselines share the same target LLMs, decoding hyperparameters, and chat templates as COPILOT’s response completion module, ensuring a rigorous and fair environment where the only differences between baselines are the prompt and decoding strategies described below.

• **Zero-shot** and **Grouped** both issue a single-turn query to the target LLM. The difference is purely on the prompt side. *Zero-shot* naively concatenates the entire preference configuration at the end of the user request as unstructured service requirements, while *Grouped* systematically reorganises the same configuration into a per-scenario structured list before appending it.

• **Step-wise** unfolds the response over two dialogue turns rather than a single turn. The first query sends the user prompt together with the full preference configuration and asks the target LLM to write a concise structural plan for satisfying the task and service constraints, without generating the final answer. The second query then provides the original

prompt, the same preference constraints, and the plan generated in the first step, and asks the model to produce the final user-facing response while following that plan. Thus, this baseline tests whether explicit planning inside the target LLM itself can improve preference following without introducing an external DLM draft.

- *Constrained* applies the positive lexical-constraint mechanism of NeuroLogic decoding (Lu et al., 2021). Given that our target LLM and other baselines use sample-based decoding (temperature, top- k , and top- p), directly applying the original NeuroLogic implementation (beam search) would make the evaluation unfair. Forcing this particular baseline to operate under beam search would inevitably confuse the measured constraint validity with a fundamental shift in the decoding paradigm. We therefore preserve the theoretical principle of positive lexical constraints but replace the heavy-weight beam-search machinery with a lightweight logits processor designed to integrate seamlessly into the existing sampling pipeline. Concretely, for each case, we extract the preferred service names, tokenize both their original and lower-cased forms, and treat the subsequent token IDs as positive lexical constraints. The logits processor scales the unnormalised probabilities of constraint tokens by 1.5 before sampling, and the input prompt remains identical to *Zero-shot*, featuring a service-requirement block appended to the user request. In addition, the baseline shares the same hyperparameters (maximum new tokens, temperature, top- k , and top- p) as the other local baselines to ensure a rigorously controlled comparison.

Models and Implementation. We run all local experiments, including COPILOT and every baseline, on the same Rocky Linux 9.7 server equipped with four NVIDIA L40 GPUs running a Python 3.9 environment. For the target LLMs, we use Qwen3-32B (Yang et al., 2025) or Gemma-3-12B (Team et al., 2025) on this local server, and access Gemini through its public API with the same prompts and preference configurations used in the local runs, which keeps the cross-family comparison strict. The diffusion language model is LLaDA-1.5 (Nie et al., 2025), which we apply uniformly across all COPILOT configurations. All open-source checkpoints are loaded through HuggingFace Transformers with their released tokenizers and chat templates, without any fine-tuning or quantization. For Qwen3-32B, we use half-precision inference, deactivate the default Qwen thinking mode in the chat

template, and generate with sampling enabled, a maximum of 2,048 new tokens. The temperature parameter is 0.7, top- k parameter is 50, and top- p is 0.95. For Gemma-3-12B, we use the same decoding parameters and switch the model dtype to bfloat16, which is the stable inference setting for this model family on our GPUs. For LLaDA-1.5, we follow the released masked-denoising inference interface and use bfloat16 weights, deterministic denoising with temperature 0.0, and without classifier-free guidance.

Building on the DLM draft mechanism described in §A.1.2, the draft generator exposes two groups of inference parameters. The first group controls the spatial distribution of anchors inside the generated region. The leading anchor is placed at the start of the region, and each subsequent anchor follows with a fixed token interval. This spacing naturally accommodates short connective tokens between adjacent anchors. We use a 6-token gap for semantic anchors and an 8-token gap for code-skeleton anchors. The second group dictates the region size and denoising budget: the main code configuration uses a 128-token generated region with 64 denoising steps, while the hybrid configuration mixes semantic and API anchors, COPILOT expands the region to 192 tokens with 96 denoising steps to accommodate the longer anchor list.

To stabilise the evaluation, we query each case (i.e., each pair of prompt and configuration) three times and report the arithmetic mean of each metric together with error bars that capture the standard deviation across the repeated generations.

A.2.2 Ablation Study

Impact of COPILOT Implementations. To complement the module-level ablation in §5.3, we extend the core ablation with two fine-grained variants that alter the implementations of the planning and drafting modules. The experiment reuses the same Qwen model and the same 100 random cases from the code dataset as Table 2, so the first four rows of Table 4 reproduce the main-table numbers on PA, IA, and DR, and the last two rows are the added variants. The - *DLM drafts for all subtasks* variant sends both preference-bound and no-preference sub-tasks through the DLM draft generator. The - *w/o scenario descriptions* variant keeps the same scenario names but removes the descriptions from the planning prompt.

We can observe that indiscriminately adding every sub-task to the DLM draft generation will

Table 4: Extended ablation results of COPILOT on 100 cases of the code dataset on the Qwen model. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable.)

Variant	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$
COPILOT	82.6	83.5	19.3
- w/o Scenario Planning	50.7	63.2	71.0
- w/o Draft Generation	69.3	78.3	20.3
- w/o Response Completion	72.0	84.8	23.7
- DLM drafts for all subtasks	75.3	81.3	23.3
- w/o scenario descriptions	62.7	73.2	21.0

Table 5: Comparison between the DLM draft and a lexically guided AR draft on 300 prompt-configuration keys of the code dataset on the Qwen model. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable.)

Variant	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$
COPILOT (DLM draft)	82.6	83.5	19.3
- Lexically guided AR draft	75.3	79.2	21.7
- w/o Draft Generation	69.3	78.3	20.3

degrade COPILOT’s performance. Forcing unconstrained (no-preference) sub-tasks through this phase increases the likelihood of accumulating unnecessary, service-specific noise, which reduces PA from 82.6 to 75.3 and increases DR from 19.3 to 23.3. Meanwhile, PA remains significantly higher than the 69.3 achieved by completely removing DLM (no draft generation). Therefore, such a performance comparison suggests that the optimal implementation strategy is to limit drafts to preference-constrained subtasks. Second, removing scenario descriptions weakens the COPILOT’s ability to map a user request to the appropriate service-bearing scenario before drafting begins, and PA falls from 82.6 to 62.7 and IA from 83.5 to 73.2. Such a degradation underscores that scenario descriptions supply essential semantic grounding for accurate scenario matching.

Controlled Autoregressive Draft Replacement. To directly examine whether the DLM is necessary in the draft generation module, we conduct a controlled experiment that replaces only the DLM draft with a lexically guided autoregressive (AR) draft produced by the target Qwen3-32B model, while keeping the planning module, the completion module, all prompts, and the evaluation pipeline unchanged. The AR replacement receives the same anchor snippets and applies a token-level lexical bias toward the preferred services during drafting. This experiment reuses the same 300 prompt-configuration keys as Table 2 (i.e., 100 code prompts, three configurations per prompt, and one generation per pair), and Table 5 reports the results.

Table 6: Text-domain ablation results of COPILOT on 300 outputs per variant on the Qwen model. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable.)

Variant	PA (%) $\uparrow$	IA (%) $\uparrow$	DR (%) $\downarrow$
COPILOT	100.0	100.0	26.7
- w/o Scenario Planning	99.0	99.5	90.0
- w/o Draft Generation	100.0	100.0	26.0
- w/o Response Completion	100.0	100.0	22.0

As shown in Table 5, the AR draft improves PA and IA over removing the draft stage entirely, which confirms the value of drafting itself. However, compared with this AR variant, the original DLM draft raises the values of PA and IA by 7.3% and 4.3% while lowering DR by 2.4%. To understand where this gap comes from, we manually check all 703 AR drafts generated for the matched sub-tasks, and find that every one of them already contains the full name of the configured provider. This indicates that the AR drafts do not lose the preferred services at the drafting stage, and the gap arises in the later completion stage. An AR draft mentions each service at only one position along its left-to-right generation, and such a single mention is easily rewritten or dropped when the target LLM completes the response. In contrast, the DLM draft fixes the service name and its related anchors at multiple positions of the skeleton simultaneously (§4.2), making the preferences much harder to lose during completion.

Text-domain Ablation. To complement the code-domain ablation in §5.3, we further evaluate the three module-level variants on the text dataset with 300 outputs per variant. In this experiment, the ‘- w/o Response Completion’ variant removes the full completion stage and retains only a lightweight cleanup that turns the draft into a readable response. Table 6 reports the results.

The results on the text dataset are highly consistent with those on the code dataset (Table 2). Removing the scenario planning module again causes the most severe degradation, where DR jumps from 26.7% to 90.0%, mirroring the sharp DR increase from 19.3% to 71.0% observed in the code domain. Note that PA and IA stay close to 100% for all text variants. This is because mentioning a preferred service in a natural-language response is much easier than embedding it into executable code, so the module effects on the text dataset mainly appear in DR rather than in adherence.

Impact of DLM Settings. To select effective default parameters for the DLM in COPILOT, we con-

duct a grid search on the gap between adjacent fixed anchors (i.e., ‘Gap’, {4, 6, 8} tokens), the generated region length (i.e., ‘Length’, {128, 192, 256} tokens), and the number of denoising steps (i.e., ‘Steps’, {64, 96, 128}). The sweep reuses the same Qwen target LLM and LLaDA-1.5 draft model as the main runs. We evaluate each parameter pair on 54 cases sampled from the code dataset and 108 cases sampled from the text dataset, with all other pipeline components held fixed. Table 7 reports the resulting PA, IA, and DR.

The results in Table 7 demonstrate the effectiveness of default parameter selections detailed in §A.2.1. The choice of DLM default settings is driven by the necessity to maximize preference adherence ($\mathcal{PA}$ and $\mathcal{IA}$) while strictly minimizing the activation of distractor scenarios ($\mathcal{DR}$). Specifically, while some larger configurations (e.g., gap 8, length 256, 128 steps in the code domain) yield marginally higher adherence metrics, they incur a substantial increase in distractor rate (DR rises to 20.4%). Conversely, minimizing DR may cause adherence to collapse, as seen when the setting of gap 8, length 256, and 64 steps reduces DR to 11.1% but drops PA and IA below 77% in the code domain. Balancing these metric results, we choose the gap 8, length 128, and 64 steps as the default settings in the code domain and gap 6, length 192, and 96 steps as the default settings in the text domain, achieving a high preference adherence while maintaining a low distractor rate.

A.2.3 Planning Accuracy and Anchor Retention

The reliability of COPILOT depends on two soft steps in the pipeline, i.e., whether the planning module activates the correct preferences, and whether the completion module preserves the injected anchors. This subsection quantifies both of them on the Qwen code experiments.

Planning Accuracy and Error Propagation. We first evaluate how accurately the planning module matches sub-tasks to preference scenarios, using the labeled scenarios of each prompt in our dataset as the ground truth. Specifically, the recall measures the proportion of requested scenarios that are successfully matched by the planning module, while the precision measures the proportion of matched scenarios that are indeed requested by the prompt, both computed over all matching results. We also report the fraction of generations whose plans mistakenly include at least one distractor sce-

nario (‘Distractor plan’). Furthermore, to show how planning errors affect the final responses, we divide the generations into two groups by whether their plans are fully correct, and report PA, IA, and DR for each group separately. Table 8 summarizes the results on the multi-task splits with two sub-tasks (MT2, 1,350 generations) and three sub-tasks (MT3, 4,500 generations).

As shown in Table 8, the planning module is reliable on our benchmark, achieving a recall above 98% and a precision above 92% on both splits. The group-wise comparison further shows that correct planning is important for the final responses. For example, on the MT2 split, DR stays at 8.0% when the plan is fully correct, but rises to 79.8% once the plan contains errors, and PA drops from 92.5% to 69.7% accordingly. This observation is consistent with that of module-level ablation study in §5.3, where removing the planning module causes the largest degradation.

Anchor Retention after Completion. Recall that the completion module only instructs the target LLM to preserve the draft contents (§4.3) without a hard guarantee. Therefore, we further measure how well the injected anchors survive the completion stage. We classify all 13,521 anchors injected in the code experiments according to how they are realized in the final fenced code, where ‘Literal’ means the anchor text appears verbatim, ‘Alias/API’ means the anchored service appears as one of its recognized aliases or API identifiers, and ‘Undetected’ covers the remaining cases. Table 9 reports the results.

Overall, 85.6% of the injected anchors survive the completion stage, either verbatim or as a recognized alias/API of the anchored service, and the retention is stable across the single-task and multi-task splits. The remaining 14.4% may come from anchors that are removed or rewritten during completion, as well as from aliases that our lexicon-based detector fails to recognize. Note that these results describe the offline behavior of the pipeline, and COPILOT does not verify the anchors at inference time. Therefore, adding a lightweight verification or repair step after completion is a potential future direction.

A.2.4 Case Study

To empirically demonstrate the superiority of COPILOT in enforcing fine-grained preference constraints while keeping response quality, we present two cases on code and text datasets.

Table 7: Draft DLM parameter sensitivity on code and text subsets. All values are percentages. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable.)

Gap	Length	Code									Text								
		64 Steps			96 Steps			128 Steps			64 Steps			96 Steps			128 Steps		
		PA $\uparrow$	IA $\uparrow$	DR $\downarrow$	PA $\uparrow$	IA $\uparrow$	DR $\downarrow$	PA $\uparrow$	IA $\uparrow$	DR $\downarrow$	PA $\uparrow$	IA $\uparrow$	DR $\downarrow$	PA $\uparrow$	IA $\uparrow$	DR $\downarrow$	PA $\uparrow$	IA $\uparrow$	DR $\downarrow$
4	128	88.9	89.8	14.8	85.2	86.1	14.8	83.3	85.2	14.8	86.1	90.7	56.5	87.0	91.2	53.7	88.0	92.1	54.6
4	192	85.2	86.1	14.8	83.3	84.3	16.7	79.6	80.6	14.8	87.0	91.7	54.6	89.8	93.5	54.6	86.1	90.7	58.3
4	256	88.9	89.8	18.5	88.9	89.8	22.2	83.3	84.3	13.0	87.0	91.2	55.6	89.8	93.5	57.4	90.7	94.0	54.6
6	128	87.0	89.8	20.4	85.2	87.0	18.5	88.9	89.8	20.4	88.9	93.1	56.5	86.1	90.3	54.6	88.9	93.1	58.3
6	192	83.3	85.2	16.7	85.2	87.0	16.7	90.7	90.7	16.7	88.0	91.7	59.3	88.9	93.5	59.3	89.8	93.5	54.6
6	256	83.3	84.3	18.5	83.3	84.3	14.8	87.0	88.0	14.8	88.0	92.1	58.3	90.7	94.0	56.5	90.7	93.5	50.9
8	128	87.0	88.9	13.0	90.7	91.7	16.7	85.2	86.1	18.5	88.0	92.6	55.6	86.1	91.2	54.6	90.7	94.0	58.3
8	192	81.5	83.3	14.8	92.6	94.4	20.4	81.5	82.4	14.8	88.0	91.7	57.4	88.0	92.1	52.8	91.7	94.9	60.2
8	256	75.9	76.9	11.1	81.5	82.4	18.5	96.3	97.2	20.4	89.8	93.5	55.6	86.1	90.3	58.3	88.9	93.1	60.2

Table 8: Planning accuracy and its downstream effect on the code dataset on the Qwen model. (MT2 and MT3 denote the multi-task splits with two and three sub-tasks. The last three rows compare the generations whose plans are fully correct with those whose plans contain errors.)

Metric	MT2	MT3
Recall (%)	98.6	98.2
Precision (%)	92.6	93.7
Distractor plan (%)	7.9	12.9
PA (%), correct / error plan	92.5 / 69.7	80.5 / 55.8
IA (%), correct / error plan	92.5 / 69.7	85.9 / 69.7
DR (%), correct / error plan	8.0 / 79.8	12.3 / 72.9

Table 9: Anchor retention after the completion stage on 13,521 injected anchors of the code dataset on the Qwen model. (Each anchor is classified by how it is realized in the final fenced code.)

Split	Literal (%)	Alias/API (%)	Undetected (%)
Single-task	54.2	32.8	13.1
Multi-task-2	52.3	38.8	8.9
Multi-task-3	52.7	31.7	15.6
Pooled	52.9	32.7	14.4

Case 1 on Code Dataset. We extend the same code case described in §A.1 (Example Intermediate Results) with responses from baselines and COPILOT to illustrate how COPILOT changes the service choices in a multi-task response. The prompt, preference configuration, and main scenarios follow the multi-task example shown in §A.2.1, where Microsoft Azure and Microsoft Azure Video Indexer are the requested target services, and Amazon S3 and AWS Lambda are distractor entries.

As observed below, the *Zero-shot* baseline response illustrates an uncontrolled and naive assimilation of the preference configuration. It treats the distractor entries as mandatory architectural components and introduces AWS CloudFront, Amazon S3, and AWS Lambda in addition to the relevant Azure services. We also include the response case from *Grouped* prompting, the strongest baseline according to the multi-task Code preference-control metrics in Table 1. Compared to *Zero-shot*, *Grouped* prompting method correctly uses Microsoft Azure

for the caching sub-task and Microsoft Azure Video Indexer for the video sub-task. However, it still erroneously assigns the text-extraction sub-task to AWS Lambda, even though *Serverless Deployment* is a distractor scenario rather than the requested main scenario. This demonstrates that simply structuring the prompt is insufficient to prevent cross-scenario contamination in multi-intent prompts.

Zero-shot Baseline Response Case

Creating a comprehensive Python solution that integrates **Microsoft Azure Video Indexer** for video processing, **Amazon S3** for data storage, **AWS Lambda** for serverless deployment, and **Microsoft Azure** as the cloud hosting platform, while also implementing **caching with CloudFront (or similar)**.

```
config.py
AZURE_VIDEO_INDEXER_API_
KEY = 'your-azure-video-indexer-api-
key'
AWS_ACCESS_KEY_ID = 'your-
access-key'
AWS_SECRET_ACCESS_KEY =
'your-secret-key'
S3_BUCKET_NAME = 'your-bucket-
name'
CLOUDFRONT_DISTRIBUTION_ID
= 'your-distribution-id'

s3_utils.py
import boto3
def upload_file_to_s3(file_path, key):
s3_client.upload_file(file_path, con-
fig.S3_BUCKET_NAME, key)
def invalidate_cloudfront_cache(keys):
cloudfront = boto3.client('cloudfront')
cloudfront.create_invalida-
tion(DistributionId=config.CLOUDFRONT_-
DISTRIBUTION_ID, ...)
```

Grouped Baseline Response Case

The grouped response first uses the relevant Azure preference for the caching layer:

Caching Layer for Static Assets (Cloud Hosting: Microsoft Azure)

This script sets up a basic Azure CDN using the Azure SDK for Python.

```
from azure.identity import DefaultAzureCredential
from azure.mgmt.cdn import CdnManagementClient
from azure.mgmt.resource import ResourceManagementClient
resource_client = ResourceManagementClient(
    credential, subscription_id)
resource_group_name, location = ...
cdn_client = CdnManagementClient(credential, subscription_id)
cdn_client.profiles.begin_create(resource_group_name, profile_name, ...)
```

Video File Size Reduction (Video Processing: Microsoft Azure Video Indexer)

This script uses Azure Video Indexer to reduce video file size while maintaining acceptable quality.

However, it then applies the distractor *Serverless Deployment* preference to the text-reading sub-task:

Text Extraction from Websites and Documents (Serverless Deployment: AWS Lambda)

This script extracts text from various document formats and web pages and is designed to run as an AWS Lambda function.

```
import boto3
def lambda_handler(event, context):
    s3 = boto3.client('s3')
    bucket = ...
    event['Records'][0]['s3']['bucket']['name']
    key = ...
    event['Records'][0]['s3']['object']['key']
    response = s3.get_object(Bucket=bucket, Key=key)
```

Conversely, COPILOT first systematically maps the caching sub-task to *Cloud Hosting* and the video-compression sub-task to *Video Processing*, while correctly leaving the text-reading sub-task unconstrained. It then drafts the structural skeleton with Microsoft Azure and Microsoft Azure Video Indexer as fixed service anchors. The final COPILOT response strictly keeps these selected services in the generated scripts while implementing the unconstrained accessibility sub-task with provider-neutral open-source tools (e.g., PyPDF2 and pyttsx3). This confirms that COPILOT successfully achieves on-demand preference activation. It fulfills requested constraints with high fidelity without introducing distractor providers into irrelevant sub-tasks,

thereby guaranteeing both adherence and overall code utility.

COPILOT Response Case

```
azure_cdn_setup.py
from azure.identity import DefaultAzureCredential
from azure.mgmt.cdn import CdnManagementClient
from azure.mgmt.storage import StorageManagementClient
from azure.mgmt.storage.models import StorageAccountCreateParameters, Sku, SkuName
from azure.mgmt.cdn.models import ProfileInner, EndpointInner, OriginInner
credential = DefaultAzureCredential()
cdn_client = CdnManagementClient(credential, subscription_id)
storage_client = StorageManagementClient(credential, subscription_id)
storage_client.storage_accounts.begin_create(resource_group, storage_account_name, storage_params).wait()
cdn_client.profiles.begin_create(resource_group, cdn_profile_name, cdn_profile).wait()
cdn_client.endpoints.begin_create(resource_group, cdn_profile_name, cdn_endpoint_name, endpoint).wait()
```

```
azure_video_optimization.py
from azure.ai.videointerpreter import VideoIndexerClient
from azure.ai.videointerpreter.models import VideoCreateParameters, VideoIndexerMediaContent
client = VideoIndexerClient(
    endpoint=f"https://{location}.api.videointerpreter.ai",
    credential=credential,
    account_id=account_id
)
with open(video_file_path, "rb") as video_file:
    video_content = VideoIndexerMediaContent(content=video_file.read())
upload_result = client.videos.upload_video(video_content=video_content, video_name="OptimizedVideo")
video_id = upload_result["id"]
client.videos.begin_index_video(video_id=video_id, video_create_parameters=VideoCreateParameters(...)).wait()
client.videos.download_video_file(video_id=video_id, video_file_type="Mp4", output_path=output_file_path)
```

```
text_to_speech.py
import requests
from PyPDF2 import PdfReader
import pyttsx3
def extract_text_from_pdf(pdf_path):
    with open(pdf_path, "rb") as file:
        reader = PdfReader(file)
    def text_to_speech(text):
```

```
engine = pyttsx3.init()
engine.say(text)
engine.runAndWait()
```

Case 2 on Text Dataset. A prevalent failure mode among prompt-based baselines is the tendency to indiscriminately satisfy all provided preferences, including those completely irrelevant to the user’s immediate request, leading to a high distractor rate. For example, the following single-task prompt only asks for a hotel booking, but the global preference configuration also injects distractor *taxi* and *attraction* entries.

Text Case Study

[Prompt]
book for 1 person for 4 nights starting from Saturday. If 4 nights is not available, how about 2 nights. Give me a step by step plan by using a certain tool.
[Preference Configuration]
{taxi: Addison Lee, hotel: Airbnb, attraction: Tripadvisor}
[Main Scenario]
hotel

As demonstrated in the ‘Zero-shot Response Case’ below, the baseline model successfully incorporates Airbnb for the hotel booking, thereby superficially achieving a perfect Preference Adherence (PA) score. However, in its blind attempt to satisfy the entire configuration, the model forces Addison Lee for transportation and Tripadvisor for attractions into the response. As a result, the baseline achieves a high preference adherence at the expense of a soaring distractor rate and severely degraded conversational utility.

Zero-shot Response Case

Okay, I can help you plan a solo trip using Addison Lee for taxis, Airbnb for accommodation, and Tripadvisor for attractions. Here’s a step-by-step plan, assuming you’re flexible on location since you did not specify a city. I will use a combination of manual searches and suggest tools to streamline the process. I will structure this with both a 4-night and a 2-night option.

Phase 2: Airbnb Accommodation

1. Go to Airbnb: <https://www.airbnb.com/>. 2. Enter the destination city. 3. Enter the dates: for the 4-night option, Saturday to Wednesday; for the 2-night option, Saturday to Monday. 4. Select one guest. 5. Apply filters that are useful for solo travelers, such as price, type of place, neighborhood, Wi-Fi, kitchen, and laundry. 6. Browse listings, read reviews carefully, especially reviews from solo travelers, and shortlist favorites. 7. Book the accommodation once a suitable listing is found.

Phase 3: Attraction Planning with Tripadvisor

1. Go to Tripadvisor: <https://www.tripadvisor.com/>. 2. Enter the destination city. 3. Open the “Things to Do” tab. 4. Filter by category, rating, price, and time needed. 5. Save candidate attractions to a Tripadvisor list. 6. Read reviews to check safety, navigation, and suitability for solo travel. 7. Create a rough daily itinerary and check opening hours. 8. Book tickets in advance when necessary.

Phase 4: Transportation with Addison Lee

1. Use the Addison Lee website or app: <https://www.addislee.com/>. 2. If flying in, book an airport transfer in advance. 3. Estimate fares for expected trips around the city. 4. Book rides through the Addison Lee app or website as needed. 5. Also check public transportation options, since they may be more cost-effective.

Conversely, the COPILOT response demonstrates precise, on-demand preference control. It successfully anchors Airbnb into the hotel planning steps while completely suppressing the unrequested *taxi* and *attraction* entries. By structurally decoupling intent planning from preference injection, COPILOT strictly confines its generation to the user’s actual constraints. This case underscores the advantage of COPILOT that it can not only control the provider preference required by users, but also preserve the natural quality and relevance of the recommendation.

COPILOT Response Case

Here’s the plan to book your hotel using Airbnb:
First, I will use Airbnb to search for hotels for 1 person for 4 nights, starting this Saturday. I will use the Airbnb website or app search interface to find available properties that meet these criteria. I’ll then filter these properties using Airbnb’s built-in website filters to ensure they are suitable. This search will prioritize properties listed on Airbnb.
If the initial search for 4 nights on Airbnb yields no available options, I will then check for 2-night availability on Airbnb, still starting this Saturday. Again, I’ll use the Airbnb search interface and built-in filters to find suitable properties. This will ensure we explore all possibilities within your preferred service, Airbnb, while respecting your request for a minimum of 2 nights.

A.2.5 Mocked Sandbox Execution

To rigorously verify whether COPILOT’s recommendation results reflect executable behavior rather than incidental service mentions hidden within comments or unreachable dead code, we conduct an advanced mocked execution experiment. We sample 300 multi-task cases from the code dataset, utilizing Qwen as the target LLM to maintain consistency with the primary evaluations. Our evaluation pipeline executes each extracted Python snippet within a lightweight, isolated sandbox environ-

Table 10: Mocked execution comparison between COPILOT and baselines on 300 multi-task cases on Qwen. (The symbols $\uparrow$ and $\downarrow$ separately indicate whether a higher or lower value of a specific metric is preferable. Bold indicates the best result, italic with underline indicates the second best result.)

Method	R-PA (%) $\uparrow$	R-IA (%) $\uparrow$	R-DR (%) $\downarrow$
Zero-shot	<u>19.7</u>	<u>43.2</u>	53.3
Grouped	18.7	40.3	41.0
Step-wise	9.7	21.3	<u>19.7</u>
Constrained	19.0	42.0	49.7
COPILOT	46.3	62.0	18.3

ment. This sandbox systematically stubs out major cloud SDKs (AWS, Google Cloud, and Azure) alongside network, file system, and subprocess interfaces. Consequently, every library import, client instantiation, SDK method invocation, and shell command is captured as a deterministic runtime event log, safely bypassing actual external API calls. For responses containing multi-file artifacts, we filter out non-executable components (e.g., Dockerfiles, dependency manifests, and YAML configurations), isolating only the Python source files into an ephemeral directory alongside the testing and mocked modules. The testing subsequently executes top-level statements under strict exception isolation and systematically invokes generated functions using dummy arguments. This robust design ensures that provider-dependent code paths are thoroughly exercised, even if the generated script is structured as an importable library rather than a standalone procedural program. Table 10 compares COPILOT and the baselines on runtime preference adherence, where R-PA, R-IA, and R-DR are the execution-level equivalents of PA, IA, and DR, computed strictly from the verifiable sandbox event logs.

The runtime metrics expose a sharp gap between COPILOT and every prompt-only baseline. *Zero-shot* and *Constrained* instantiate distractor providers on more than half of the responses, pushing R-DR to 53.3% and 49.7% while R-PA stagnates below 20%. *Grouped* only trims R-DR to 41.0%, more than double the value of COPILOT, and its R-PA still tracks the naive zero-shot baseline. *Step-wise* attains the lowest R-DR (19.7%) among the baselines, yet its R-PA collapses to 9.7%, indicating that the multi-turn planner suppresses preferred and distractor providers alike rather than enforcing genuine preference control. In contrast, COPILOT delivers the best R-PA (46.3%) and R-IA (62.0%), outperforming the strongest baseline by a factor of at least 2.3, and simultaneously secures the lowest R-DR (18.3%)

Table 11: Efficiency and cost comparison between COPILOT and baselines. ('Baseline Min' and 'Baseline Max' report the lowest and highest values among the four baselines on each metric. All rows except the last are measured on 300 aligned code outputs on the Qwen model, while the last row reports the target-model API generation charge on the recorded Gemini code panel with 216 outputs per method.)

Detail	COPILOT	Base. Min	Base. Max
Target-LLM calls / output	2	1	2
Median latency / output (s)	224.4	168.4	237.1
Avg. input tokens / output	2385.9	143.8	979.9
Avg. output tokens / output	1991.4	1500.4	1830.6
Additional checkpoint storage (GB)	15	0	0
Gemini API charge (USD)	0.431	0.480	0.571

without sacrificing preferred-provider coverage. Such results highlight that the design of COPILOT extends beyond injecting preferred service names into prose or comments. It effectively embeds the requested services into executable code snippets while keeping distractor providers out of the runtime environment.

A.2.6 Efficiency and Cost Analysis

To quantify the computational overhead introduced by the additional planning and drafting stages, we measure the end-to-end efficiency of COPILOT and all baselines on an aligned panel of 300 code outputs (i.e., the same 100 cases with three configurations as §5.3) with the Qwen model. For each method, we record the number of target-LLM calls and the median wall-clock latency per output, and we further estimate the average input and output tokens of the target LLM with its tokenizer. Since not every metric is separately logged for each baseline, Table 11 reports each metric of COPILOT together with the lowest and highest values among the four baselines, illustrating where COPILOT falls within the baseline range.

As shown in Table 11, the median latency of COPILOT (224.4s) falls within the range of the baselines, only 3.7% above *Constrained* (216.4s) and 5.4% below *Grouped* (237.1s), while it is slower than *Zero-shot* (185.0s) and *Step-wise* (168.4s). Within COPILOT, the per-output median latencies of the three modules are 15.0s for scenario planning, 15.7s for the batched DLM drafting, and 190.6s for the final completion. Thus, the two additional stages account for only a minority of the overall latency, which is dominated by the response generation of the target LLM. For the storage overhead, COPILOT only requires an additional 15 GB memory for loading the DLM, substantially smaller than the 62 GB memory of the target model (Qwen3-32B). Note that the token counts are tokenizer-based estimates rather than

API billing records, the DLM denoising tokens are excluded, and we did not measure the throughput or peak GPU memory.

We further summarize the monetary cost on the recorded Gemini code panel (the last row of [Table 11](#)), which contains 8 prompts from each of the single-task, multi-task-2, and multi-task-3 splits with three configurations and three repeats (i.e., 216 outputs per method). Based on the logged rates of USD 0.375 per million input tokens and USD 2.25 per million output tokens, the target-model generation charge of COPILOT is USD 0.431 (432 calls), which is slightly lower than the USD 0.480–0.571 of the prompting baselines (216–432 calls), since the drafts constrain the completion stage and shorten the redundant enumeration of configured services in responses. Note that this comparison does not include local DLM computation (which is much smaller and cheaper than the target model) and the evaluator calls shared across all methods. This comparison mainly highlights that COPILOT will not lead to higher overhead on the target model.

A.3 Limitations and Future Work

Dataset Distribution. The raw prompts in our benchmark are adapted from prior recommendation work ([Zhang et al., 2025](#); [Budzianowski et al., 2018](#); [Rastogi et al., 2020](#)) and the synthesized multi-task prompts concatenate at most three sub-tasks per prompt. This construction may not fully reflect the distribution of real-world conversation logs, where the topic coverage, sub-task length, and writing style of user prompts could be more diverse. Nevertheless, this limitation does not weaken the conclusions of this paper, since COPILOT and all baselines are evaluated on the same prompt set and preference configurations, so the reported numbers describe the performance gap between methods rather than the scenario coverage of the dataset. A natural extension is to mine prompts from real conversational logs of LLM recommendation services, in which longer prompts and noisier scenario boundaries can stress the planning and anchoring stages of COPILOT more aggressively. We see this direction as a complement to, rather than a replacement for, the controlled evaluation conducted in this paper.

Faithfulness of Preferences. Our evaluation assumes the preference configuration $\mathcal{C}$ faithfully reflects user intent, whether the user uploads $\mathcal{C}$ directly or the platform aggregates $\mathcal{C}$ from interaction history. In real deployment, both channels

could introduce noise (e.g., overlapping scenario boundaries, stale entries from outdated tech stacks, and underspecified service descriptions whose intended scope is unclear). However, this paper does not aim to investigate how much noise exists in real-world preference configurations, and our experiments compare methods under a shared $\mathcal{C}$, so the relative comparison between COPILOT and the baselines is unaffected by the absolute quality of $\mathcal{C}$. Future research could design preference debugging and merging methods that detect contradictions, deduplicate overlapping entries, and surface stale items, so that the upstream $\mathcal{C}$ remains a faithful description of user intent before any control method is invoked. Building such tools is largely orthogonal to the inference-time control problem studied here, and we view the two lines of work as complementary.

Dependence on the Underlying DLM. COPILOT currently adopts LLaDA-1.5 ([Nie et al., 2025](#)) as the backbone DLM in the draft generation module. Diffusion language modeling is still a fast-moving area, and stronger backbones with larger parameter scales, faster sampling, and richer training data are likely to appear in the near future. Whether the absolute numbers in [Table 1](#) carry over to those backbones cannot be answered by our current experiments alone. The qualitative design of COPILOT remains model-agnostic, since the planning, anchoring, and completion stages do not rely on any LLaDA-specific behavior beyond bidirectional masked decoding. Re-running the pipeline on stronger DLMs, and quantifying how much of the observed gain originates from the framework versus from the backbone, is a clear next step. We release the scripts, prompts, and datasets to support such reproduction and exploration by future research, and we encourage the community to swap in newer DLMs as they become available.