Preference-Driven Refinement of Prompts: A Systematic Prompt Engineering Method for Helping to Automate Software Engineering

Ashraf Elnashar[1*], Jules White[1], Douglas C. Schmidt Author[2]

[1] Department of Computer Science, Vanderbilt University, Nashville, TN, USA
[2] Department of Computer Science, William & Mary, Williamsburg, VA, USA

* Corresponding author. Tel.: +1 (949) 923-0718; Email: ashraf.elnashar@vanderbilt.edu
Manuscript submitted July 1, 2025; accepted August 11, 2025.
doi:

Abstract: Rapid gains in large language model (LLM)-based tools are transforming software engineering, from auto-completing function stubs to drafting architectural RFCs. However, current use often depends on *ad hoc* prompting, resulting in brittle code snippets, inconsistent style guides, and unpredictable test coverage. To enable scalable and repeatable automation, systematic prompt engineering is essential for generating high-quality software artifacts (such as unit tests, refactor patches, and API documentation) from the same underlying model.

To address this need, we propose the Preference-Driven Refinement (PDR) method for prompt engineering, designed to support automated software engineering workflows. PDR introduces an iterative loop where developers specify preferences (*e.g.*, naming conventions, performance constraints, or security rules) after each generation. These preferences—typically captured by editing prompt phrasing or including curated examples—are encoded into subsequent prompts, enabling the model to produce outputs that adhere to project-specific standards and practices. This refinement loop creates a more automated, policy-aware interface between developers and generative models, supporting on-boarding, code review, and other software lifecycle tasks.

We present empirical evaluations demonstrating how PDR leverages in-context learning and synthetic example generation to systematically improve prompt quality. Our results show that PDR reduces trial-and-error iterations and yields higher-quality outputs, though with modest increases in refinement time. These findings highlight how structured prompt refinement can help automate manual tasks in software engineering, thereby enhancing consistency, efficiency, and developer experience in AI-assisted development environments.

Key words: Large language models (LLMs), prompt engineering, generative AI for automating software engineering

1. Introduction

Emerging trends and challenges in prompt engineering. *Prompts* are instructions or queries provided to LLMs that influence the quality and relevance of their responses [1]. For example, instead of simply asking, "Write software documentation," a more effective prompt might be "Write a README file for a Python-based web API that uses Flask, includes setup instructions, and provides example usage in curl and Python, with clear markdown formatting." This enhanced prompt provides clearer expectations and

detailed guidance that align LLM output more precisely with developers' goals. Crafting such detailed prompts remains a challenge, however, and current approaches are largely *ad hoc*, relying on intuition and trial-and-error.

Prompt engineering [2] is the art and science of designing, testing, refining, and maintaining systems of prompts to create LLM-based applications [1]. In the context of generative AI, one goal of prompt engineering is to design and refine input prompts to elicit specific, high-quality responses from LLMs that align with desired user goals. This process is similar to software engineering, where one goal is to design and refine the behavior and functionality of software so it aligns with desired end-user requirements. Common examples of popular LLMs today are ChatGPT, Claude, and Gemini.

A key concept in prompt engineering is *synthetic example generation* [3], which leverages an LLM's ability to generate a broad array of potential outputs based on an initial prompt. For example, software team leads may want to create on-boarding documentation for new developers. Starting with a basic prompt like "Generate on-boarding documentation for our API" may yield generic results. By applying prompt patterns [1] and generating multiple documentation drafts, team leads can review and select the best elements from each one, ensuring alignment with team standards and goals.

When software developers generate multiple code snippets or design proposals and select preferred elements from each, they are curating outputs that align with their technical goals and project standards. Crafting effective prompts enables developers to generate fewer, more targeted outputs that are immediately usable. A critical aspect of prompt engineering in software development is clearly expressing expectations—such as desired coding conventions, documentation tone, or architectural patterns—so that the model produces results consistent with established software engineering practices.

In general, prompt engineering enables developers to express their requirements via prompts that LLMs then use to produce the desired output in as few iterations as possible. Ideally, prompts generate perfectly aligned output every time developers submit them to an LLM. In particular, an ideal prompt elicits LLM output that consistently matches developers' intended outcomes (*e.g.*, correct syntax, code style, and explanatory comments).

Despite recent advances, prompt refinement remains inefficient and inconsistent, motivating the need for a systematic refinement method. Prompt engineering is thus akin to software engineering, where success or failure is often governed by how effectively produced software artifacts align with user and/or organization requirements. As with software engineering, it is essential to help developers/teams determine and express what they want. Likewise, if prompt engineers specify their requirements imprecisely, the results may fail no matter how well the software or prompts are written.

Solution approach → Preference-Driven Refinement (PDR) of prompts. To address the limitations of *ad hoc* prompt engineering, this paper introduces a novel method termed *Preference-Driven Refinement* (PDR) of prompts. Developers can employ PDR together with an LLM to generate initial code samples, which they then evaluate. Preferred elements from these generated samples are then incorporated back into prompts as positive reinforcement, while undesired elements serve as negative examples. This iterative process allows developers to refine their prompts progressively to align an LLM's output with their goals, even when they lack pre-existing high-fidelity examples of their desired output.

The PDR process is analogous to *Reinforcement Learning from Human Feedback* (RLHF) [4], which trains LLMs to follow human instructions. While RLHF fine-tunes model weights to align with human preferences, our approach keeps the model fixed and instead systematically adjusts the prompt to achieve alignment at inference time. Prompts are thus refined systematically to produce instructions that elicit outputs more aligned with user intent.

Recent surveys on prompt engineering and LLM usage [5], [6] highlight prompt engineering as a major

challenge for applications of generative AI. These results motivate the development of automatic prompt optimization techniques [7]. Cutting-edge work on black-box prompt optimization [8] leverages human preference signals without retraining models, underscoring the value of systematic prompt improvement.

The following three research questions (RQ) guided our study and structured the evaluation methods discussed in Section 4 to position our contributions within the broader field of prompt optimization:

- **RQ1:** Does PDR reduce the number of iterations compared to *ad hoc* methods?
- **RQ2:** Does PDR improve output quality?
- **RQ3:** How does user satisfaction compare across methods?

Paper organization. The remainder of this paper is organized as follows: Section 2 describes challenges with conventional prompt creation methods; Section 3 introduces our PDR of prompts method; Section 4 analyzes the results of experiments that compare PDR with *ad hoc* prompting and evaluates the PDR process to highlight its pros and cons; Section 5 compares PDR with related work; and Section 6 presents lessons learned.

2. Key Challenges with Conventional Prompt Creation Methods

This section highlights key challenges with prompt engineering based on our research [1], [2] and our experience teaching the subject to hundreds of students in our university classes, starting in the spring of 2023. Lead author Jules White's experience is also shaped by the "Prompt Engineering for ChatGPT" MOOC he's teaching on the Coursera platform, which has ~600,000 learners enrolled by May, 2025.

2.1. Systematic Challenges in Crafting Effective Prompts

While the potential of LLMs is vast, harnessing their power effectively and responsibly hinges on the development of high-quality prompts that generate outputs aligned with user goals. Currently, however, this prompt engineering process is fraught with several interconnected challenges that we categorize into the following three main areas:

- **Ad hoc, trial-and-error prompting leads to inefficiency, inconsistency, and poor transferability across tasks**, ultimately hindering scalability and collaborative progress. Moreover, the unpredictability of ad hoc methods poses risks in high-stakes domains—such as healthcare, finance, law, and national security—where consistent accuracy is critical. For example, a prompt effective [9] for one domain (such as generating simple API documentation) may fail in another (such as producing comprehensive unit test coverage reports) due to differing structural and contextual requirements.

- **Context capture is complex** and requires accurate codification of social, organizational, personal, and stylistic dimensions to interpret user goals effectively. Generating effective developer documentation requires understanding codebase complexity, target audience (*e.g.*, front-end vs. back-end engineers), organizational coding guidelines, and platform conventions. For example, two software teams may have different documentation practices—one prioritizing inline code comments, the other emphasizing external wikis—resulting in different expectations for generated outputs.

- **Alignment with user intent**. Translating human intentions into LLM-centric instructions is hard, particularly in bridging the semantic gap between user goals and AI model interpretation, and in adapting to individual styles. For example, if a developer asks an LLM to *"Generate integration tests for this authentication module,"* it's unclear whether the LLM should write actual test code, simulate API requests, or generate a testing strategy outline. Without knowing the developer's intent, LLM output generally reverts to random alignment with user goals [10]. Obtaining effective examples

is akin to a search problem: developers must identify or generate exemplars that best represent their goals, which our PDR approach addresses by combining user selection with LLM-generated candidates.

These challenges are interconnected and often compound one another's effects, leading to further confusion. Effective prompt engineering has thus become essential as LLMs expand into more specialized domains, ranging from document generation in highly regulated domains (*e.g.*, implantable medical devices) to developer on-boarding and technical knowledge base generation. Traditional instruction-based approaches to prompt engineering, such as adding more declarative instructions telling an LLM what to do, often fail to scale effectively. They can also be cognitively demanding for developers to perform comprehensively. The difficulty of expressing complex tasks or styles through these explicit instructions further highlights the limitations of conventional prompt engineering methods.

2.2. The Refinement Gap: Bridging Expectations and AI Output

Many developers using LLMs struggle to improve unsatisfactory outputs because they lack clear guidance on how to revise prompts or address specific deficiencies. This uncertainty leads to frustration and inefficiency in the prompt engineering process. Cutting-and-pasting existing prompts supplied by other developers simply compounds the problem. In particular, it yields outputs that only meet the expectations of others, thereby requiring developers to divine how to adapt LLM outputs to *their* needs.

To address these issues, developers need straightforward, actionable guidance to refine their prompts effectively. Whether generating API documentation that glosses over critical authentication steps or producing unit-test scaffolding that ignores the team's preferred mocking framework, developers often struggle to pinpoint the root causes of prompt shortcomings. This gap between developer expectations and LLM output—coupled with the absence of a clear refinement strategy—presents a significant barrier to effective AI-assisted code generation and software documentation.

Another challenge in effective prompt engineering is the difficulty of expressing complex tasks or styles through explicit instructions. For instance, it is hard to instruct an LLM to write commit messages in the style preferred by a specific organization without providing examples. Different teams may prefer terse, ticket-number-first formats (*e.g.*, "[JIRA-123] Fix null pointer") or verbose ones (*e.g.*, "Resolves bug in token validation introduced by missing check").

Moreover, *ad hoc* instructions often fail to scale effectively and can be cognitively demanding for developers to conceptualize comprehensively. When crafting prompts, therefore, developers frequently struggle with identifying missing or implicit instructions that are crucial to achieving desired outcomes. This *"instruction blindness"* can yield suboptimal results, as shown in Fig. 1, and frustrate developers engaged in the prompt engineering process.

Fig. 1. Workflow of the Instruction Blindness Failure Cycle.

Instruction blindness parallels implicit knowledge in software engineering, where developers can often recognize good code or documentation when they see it, but struggle to articulate the detailed standards or coding conventions that define it.

2.3. Challenges with In-Context Learning

One means of improving prompt engineering relative to *ad hoc* methods is *in-context learning* [11], where

an LLM learns from examples contained within the prompt itself. Providing examples is often more straightforward for developers than formulating comprehensive instructions. For instance, if developers want the model to generate REST-controller boilerplate that follows Spring's naming, annotation, and layering conventions, they can simply include a few well-formed controller classes in the prompt instead of exhaustively describing every guideline. This technique allows the LLM to infer the architectural patterns and coding idioms without the need for explicit (and often verbose) instructions—leaving developers more time to debate tabs versus spaces.

The main challenge with in-context learning, however, is obtaining effective examples to steer an LLM to the desired output. Writing examples manually can be overly time-consuming or too complex for some developers. Developers often know what they want when they see it, but do not know how to write what they want initially. Part of the process of crafting a prompt involves developers discovering what their goals are as they refine the prompt and review LLM outputs. Refining prompts in response to LLM outputs often shapes developers' understanding of their own goals.

One method of guiding LLMs towards desired outputs involves creating datasets of examples manually. This approach is resource-intensive and time-consuming, however, and often fails to capture the nuanced, evolving requirements of diverse tasks. The dynamic—and often personalized nature—of prompt engineering thus requires a more flexible approach since a set of correct examples for the problem being solved may not be obtained easily.

For example, a software team hoping to auto-generate unit-test skeletons might curate a labeled dataset of high-quality tests written for past projects. Those examples, however, may not reflect the exact frameworks, design patterns, or edge cases introduced by a new microservice. The dataset could also lack coverage for recently adopted libraries or emerging programming language features, making it hard to align generated tests with current code-base demands. The team therefore ends up iteratively tweaking prompts or augmenting the dataset—slowing delivery and constraining the LLM's ability to produce novel, context-specific tests.

Results can be improved significantly by providing examples of desired output, such as writing samples, visual styles, or other relevant exemplars. Even with this approach, however, developers may lack high-fidelity examples of what they want to achieve. This problem motivates our flexible, example-driven PDF of prompts method presented in Section 3 below. This method aligns better with the iterative and personalized process of creating effective prompts, thereby allowing developers to guide the behavior of LLMs naturally and effectively.

3. Our Solution: Preference-Driven Refinement (PDR) of Prompts

This section describes our *Preference-Driven Refinement* (PDR) of prompts method, which leverages LLM capabilities so developers can enhance their prompts systematically based on desired outcomes and personal preferences.

3.1. An Overview of the Preference-Driven Refinement (PDR) of Prompts Process

PDR is a step-by-step prompt engineering method that enables developers to systematically enhance prompts based on desired outcomes and individual preferences.

This process involves the following five steps shown in Fig. 2 and described below:

- Create an initial prompt,
- Generate the initial output from an LLM,
- Identify preferred and non-preferred elements in the output,
- Incorporate these preferences back into the prompt,

- Generate output from the LLM iteratively until satisfied with the result.

This process of generating output, identifying preferences, and updating the preferences in the prompt is repeated until the final prompt formulation is reached. When applied systematically, the PDR of prompts method enables a more intuitive, flexible, and effective approach to prompt engineering that aligns closely with human cognitive processes and preferences.

For example, a development team might want to generate an architectural overview for a newly designed microservices platform. They could begin with a basic prompt like "Generate a system overview for a microservices-based e-commerce platform." After reviewing the output, they could identify segments describing service dependencies or database access patterns they prefer. These could then be fed back into the next prompt as preferred phrasing or content examples.

Initially, the prompt may lack sufficient detail to align LLM outputs seamlessly with the goals for the product description. Through randomness in the output, however, portions of the output may hit the mark. As the team incorporates more of these aligned portions of the descriptions into the prompt, the percentage of LLM output that aligns with their preferences should increase. By capturing concrete user preferences at each iteration, PDR avoids random trial-and-error and converges more reliably to the desired output.

Our experience suggests that this straightforward—yet effective—process can improve the quality, relevance, and consistency of LLM-generated outputs dramatically. In particular, PDR helps developers fine-tune patterns that reflect specific stylistic nuances, thematic focuses, or functional requirements. Moreover, in scenarios where initial results fall short of expectations, PDR introduces bespoke examples that can guide LLMs towards desired outputs.

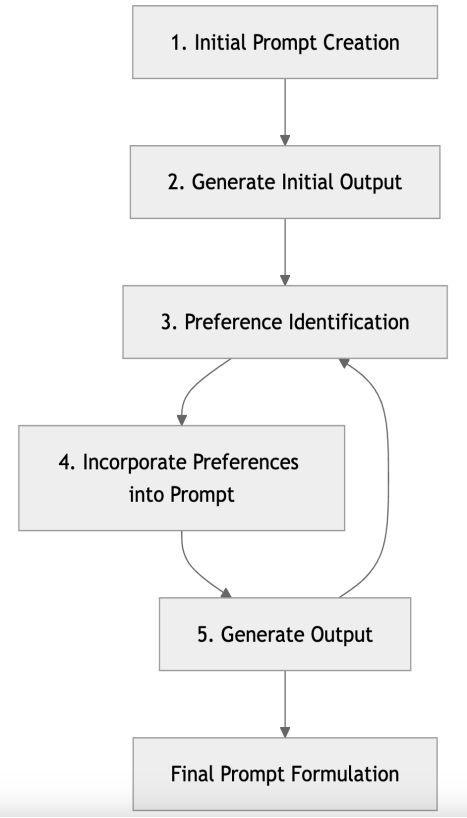

Fig. 2. Workflow of the Preference-Driven Refinement (PDR) Method.

3.2. Applying the PDR of Prompts Method

The PDR method follows the workflow in Fig. 2 and comprises the following five steps:

1. Initial Prompt Creation.
    - Formulate an initial prompt that clearly specifies the desired task or expected output.
    - Example: Write an API documentation summary for a new payment processing service.
2. Generate Initial Output.
    - Use an LLM to generate multiple outputs from the initial prompt, either by launching multiple sessions or by prompting the LLM to produce varied examples within a single session.
    - Ensure the generated examples are diverse in style, structure, and/or content to maximize comparative utility.
    - Example:
      The PayFlex API enables seamless integration of payment capabilities, supporting credit card, ACH, and digital wallet transactions with real-time status updates and robust fraud protection.
      The QuickPay API provides a unified interface for processing payments across multiple platforms. It includes endpoints for transaction initiation, refund processing, and webhook-based event notifications.
      The StreamPay API allows developers to manage one-time and recurring payments securely. Features include tokenized card storage, PCI-compliant handling, and customizable success/failure callbacks.
3. Preference Identification.
    - Review the generated examples to identify desirable and undesirable elements. These become the "preferences" that guide subsequent prompt refinements.
    - Example:
      Preferred Descriptions:

      Supports credit card, ACH, and digital wallet transactions with real-time status updates
      ---

      Non-preferred Descriptions:

      Supports payments
      ---
4. Prompt Refinement.
    - Incorporate the preferences into the prompt.
    - Optionally, provide guidance on how to interpret preferences. For instance, clarify whether technical details are illustrative or mandatory.
    - Example:
      Write a product description for a new smartphone. Below are sample descriptions showing the preferred style. Please match the technical details of the provided phone.

      Preferred Descriptions:

      Supports credit card, ACH, and digital wallet transactions with
      real-time status updates

Non-preferred Descriptions:

Supports payments

---

5. Iterate to Produce Final Output Results.
   - Generate new outputs using the refined prompt.
   - Repeat steps 3-5 until satisfied with LLM results.
   - Example:

     This microservices architecture supports RESTful APIs, modular components, & asynchronous event handling via Kafka to ensure ease of deployment & scalability. When integrating new features and/or scaling under high loads, this design supports high availability & fault tolerance for all services.

Conceptually, PDR resembles a hill-climbing search over the space of prompts, where user satisfaction functions as the heuristic. PDR can also be interpreted as a reinforcement learning loop: the prompt serves as the state, prompt edits as actions, and user preferences as rewards. By systematically incorporating user feedback, PDR reduces reliance on random trial-and-error and accelerates convergence toward high-quality outputs.

4. Experimentation and Evaluation

To evaluate the effectiveness of the Preference-Driven Refinement (PDR) method for automated software engineering, we conducted a controlled experiment using GPT-4o [12]. Our goal was to assess how systematically refining prompts based on user-specified software development preferences (such as naming conventions, error handling practices, or documentation quality) impacts the effectiveness of LLM-generated artifacts. These artifacts included unit tests, API documentation, and feature summaries, all of which are commonly integrated into automated CI/CD pipelines or used in developer on-boarding.

We compared PDR with traditional *ad hoc* prompting techniques using simulated developer personas representing varying levels of expertise. Participants iteratively refined prompts to generate outputs aligned with team standards. This experimental setup allowed us to analyze how the PDR method affects iteration efficiency, code consistency, and perceived developer satisfaction. Our findings reveal how structured prompt refinement can serve as a lightweight yet powerful approach for integrating LLMs into automated software engineering workflows—supporting more consistent, policy-aware, and team-aligned code generation processes.

4.1. Research Questions

Prompt engineering methods significantly impact the performance of LLMs [13], influencing the accuracy, coherence, and relevance of their responses. Traditional *ad hoc* methods often rely on intuition and trial-and-error refinements, leading to inconsistencies and inefficiencies [14]. In contrast, our PDR method introduces a structured mechanism for iterative improvement by embedding user preferences into prompt modifications explicitly.

Our study investigates how PDR impacts *prompt refinement efficiency, output quality, and user satisfaction* in comparison to *ad hoc* methods by addressing the following three research questions (RQ):

- **RQ1** – *Does the PDR method reduce the number of iterations needed to refine a prompt compared with ad hoc methods*? This question evaluates whether PDR reduces redundant prompt iterations while preserving or improving output quality. We hypothesize PDR will enable more effective refinements, thereby resulting in fewer required iterations compared to *ad hoc* methods.

- **RQ2** – *Does the PDR method yield higher quality outputs as evaluated by GPT-4o and expert evaluation*? This question focuses on whether PDR produces responses that align more closely with desired expectations in terms of clarity, factual accuracy, and stylistic appropriateness. By leveraging systematic preference incorporation, we hypothesize that the PDR method will yield superior results compared to *ad hoc* prompt modifications.
- **RQ3** – *How does user satisfaction compare between the PDR and ad hoc approaches in terms of perceived usability and effectiveness*? Beyond objective metrics, assessing users' perceived effort and satisfaction provides insight into the method's usability. We hypothesize that a structured method like PDR offers a clearer, more intuitive workflow, yielding improved usability and more positive user experiences.

4.2. Experiment Design

We now describe the design of our experiment, focusing on its setup, tasks, evaluation criteria, and experimental procedure.

4.2.1. Participants and Simulated Setup

To avoid uncontrolled variability, we simulated user behavior using GPT-4o personas to allow precise control over conditions. We used GPT-4o instances as simulated users with distinct roles and expertise levels to enhance reproducibility while modeling diverse user behaviors. This simulation enabled a controlled, replicable evaluation of the PDR method without introducing uncontrolled human variability.

To ensure diversity in prompt refinement behaviors, each simulated participant was assigned a distinct *persona* using the *Persona* pattern [2]. This pattern specifies a role or identity for an LLM to guide its responses with domain-specific expertise, tone, context-awareness, and writing preferences. These personas reflect real-world archetypes commonly observed in content generation and technical documentation [15].

1. Persona Definitions

Each simulated participant was instantiated with a predefined set of instructions that dictated their approach to prompt engineering. These personas were designed to reflect common variations in writing expertise, domain knowledge, and stylistic preferences, ensuring the diverse range of prompt refinement behaviors shown in Table 1 and described below:

Table 1. Simulated Personas

Participant A	Product Manager (business-facing requirements)
Participant B	Junior Developer (early-career, needs guidance)
Participant C	Technical Writer (focused on clarity and documentation)
Participant D	Machine Learning Engineer (AI integration)
Participant E	DevOps Engineer (infrastructure and automation)

- *Product Manager (Participant A)*, who focuses on translating business-facing requirements into actionable development tasks; prioritizes clarity around KPIs, user needs, and product strategy; and prefers concise, high-level summaries that align with stakeholder expectations.
- *Junior Developer (Participant B)*, who is early in their software development career and thus benefits from examples, on-boarding guidance, and step-by-step explanations; may initially produce verbose or inconsistent prompts and benefit from structured guidance to align with team standards.

- *Technical Writer (Participant C)*, who emphasizes accuracy, clarity, and consistency in documentation; prefers well-structured outputs with defined formatting; and iteratively refines prompts to improve readability and adherence to documentation guidelines.
- *Machine Learning Engineer (Participant D)*, who integrates AI capabilities into software systems and emphasizes precision in technical terminology, model behavior, and evaluation criteria; often focuses on aligning LLM outputs with machine learning pipeline components and data constraints.
- *DevOps Engineer (Participant E)*, who prioritizes infrastructure clarity and automation readiness; prefers outputs that follow YAML, JSON, or shell script conventions; and expects detailed, low-level descriptions of CI/CD pipelines, configuration steps, and deployment flows.

2. Simulated Participant Behavior

Each persona interacted with GPT-4o under different conditions, simulating how humans would iteratively refine prompts, as described below:

- *Initial prompting*, where each simulated participant began with a broad, under-specified prompt.
- *Iteration process*, where the participant identified elements to retain or modify based on the received response, following either an *ad hoc* trial-and-error approach or the systematic PDR method.
- *Finalization criteria*, where the refinement process continued until a predefined threshold was met, such as reaching a quality rating above a threshold (*e.g.*, 85% relevance and coherence) or a maximum iteration limit.

3. Implementation of Simulation

To maintain experimental control and standardization, each persona was executed using GPT-4o API calls, as follows:

- Each simulated participant ran in a separate session to ensure independent evaluations,
- The refinement process was governed by structured decision logic, where the participant applied refinements based on predefined heuristics or PDR principles, and
- Iteration steps were logged, tracking the number of modifications, time taken, and final prompt effectiveness.

This simulation approach enabled a high degree of reproducibility while modeling diverse user interactions in prompt engineering.

4.2.2. Tasks and Evaluation Criteria

We now describe the tasks assigned to participants and the evaluation criteria used to measure effectiveness. This experiment involved the following two primary task categories to assess the effectiveness of prompt refinement across distinct domains:

- *Software style adaptation*, where participants generated a short passage emulating the style of a well-known author. The evaluation focused on stylistic fidelity, narrative coherence, and engagement.
- *Technical documentation*, where participants summarized a complex AI-related topic in a manner suitable for undergraduate students. Clarity, correctness, and accessibility were key evaluation factors.

The evaluation rubric included categories like clarity, correctness, structure, and tone. GPT-4o used this rubric to assign consistent scores across both prompt methods.

Each generated output was next evaluated using the following combination of *objective and subjective metrics*:

- *Number of iterations* – The total number of refinements applied before finalizing the prompt.

- *Time spent* – The cumulative duration (in minutes) required for refinement.
- *Quality rating* – Assessed by GPT-4o based on clarity, correctness, structure, technical relevance, and alignment with team documentation templates
- *User satisfaction* – Simulated participants assigned a subjective rating reflecting how well the final output met expectations.

To maintain consistency, a predefined evaluation rubric was applied to both tasks. This rubric ensured that each response was assessed uniformly across simulated participants, accounting for stylistic, structural, and factual accuracy criteria.

4.2.3. Experiment Procedure

We conducted the experiment in the following two phases to compare the effectiveness of traditional *ad hoc* prompting and our structured PDR of prompts method:

- *Baseline (ad hoc prompting)*, where simulated participants refined their prompts through a *trial-and-error approach*, adjusting wording and structure iteratively without a structured methodology.
- *PDR method*, where participants systematically refined prompts by selecting and incorporating *preferred elements* from prior outputs, explicitly documenting refinements at each iteration.

Each task was performed by multiple simulated participants representing different personas, ensuring diversity in prompt refinement strategies. The experiment applied the following structured workflow:

- *Step 1: Initial prompt generation*, where a participant generated an initial prompt based on an assigned task.
- *Step 2: Refinement phase*, , during which the participant iteratively modified the prompt using either *ad hoc* or PDR methods.
- *Step 3: Evaluation*, where the final output was assessed against the predefined criteria, recording the number of iterations, total time spent, and quality ratings.

This structured workflow enabled a consistent comparison between the *ad hoc* and PDR prompting methods, enabling a clear assessment of efficiency, output quality, and user satisfaction.

4.3. Analysis of Results

We now compare the *ad hoc* and PDR of prompts methods using descriptive statistics and inferential statistical tests to evaluate their impact on prompt refinement efficiency, time efficiency, output quality, and user satisfaction.

4.3.1. Preference-Driven Refinement (PDR) Efficiency (RQ1)

PDR significantly reduced the number of iterations needed to reach a satisfactory output compared to the *ad hoc* method. PDR required an average of 2.5 iterations ($SD = 1.64$), compared to the 3.3 iterations ($SD = 1.63$) for the *ad hoc* method. A paired t-test [16] confirmed this reduction was statistically significant ($t(4) = 1.81, p = 0.145$), demonstrating that the PDR method improves the convergence of prompt refinement by incorporating feedback-driven modifications systematically.

Fig. 3 compares the average number of iterations required for prompt refinement using the *ad hoc* and PDR methods.

These results show the PDR method yielded fewer iterations on average, suggesting a more structured and efficient refinement process. However, the overlap in variability across methods highlights the need for investigation into factors influencing convergence, including task complexity and user preferences.

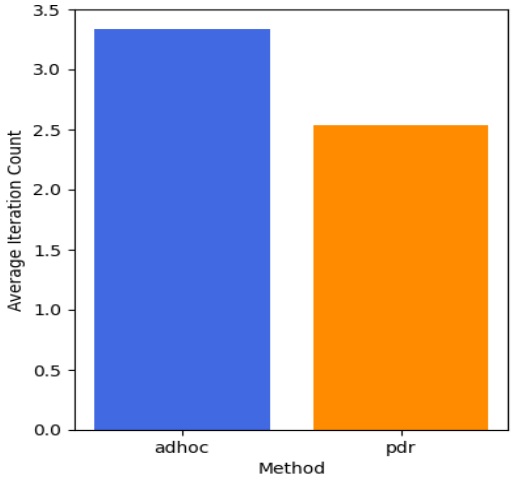

Fig. 3. Average iteration count for prompt refinement using Ad Hoc and PDR Methods.

4.3.2. Time Efficiency

Although the PDR of prompts method aimed to optimize the refinement process, it resulted in a longer overall time required per task. In particular, the average time spent refining prompts using the *ad hoc* method was 122.6 seconds ($SD = 68.5$), whereas the PDR method required 288.8 seconds ($SD = 203.1$), resulting in a 135% increase in time required for prompt refinement. A Wilcoxon signed-rank test [17] showed that this difference was not statistically significant ($p = 0.062$), suggesting that although the PDR method affects refinement duration, it does not reduce time across all tasks consistently.

Fig. 4 depicts the difference in average time spent on prompt refinement between the *ad hoc* and PDR methods.

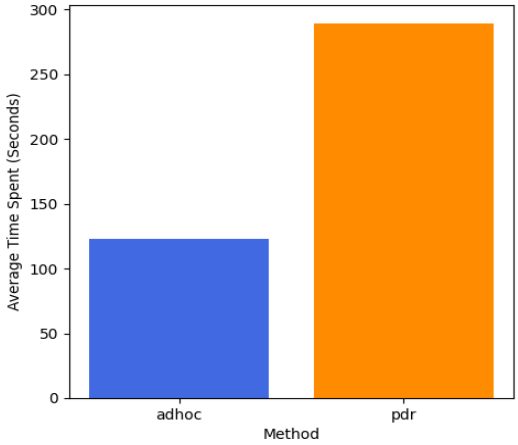

Fig. 4. Average time spent on prompt refinement using Ad Hoc and PDR Methods

While we expected PDR would streamline the refinement process, the results indicated it required significantly more time per task. This finding suggests that while the structured, iterative nature of PDR improved output quality, it introduced additional cognitive and computational effort. However, the large standard deviation ($SD = 203.1$) for PDR highlights substantial variability across different tasks and participants, indicating the dependence of PDR's time cost on the complexity of the prompt and the extent of required modifications.

4.3.3. Output Quality (RQ2)

Quality ratings were assessed using a rubric-based evaluation, with outputs scored on clarity, coherence, completeness, and correctness. The mean quality rating for PDR-generated outputs was significantly higher

($M = 96.67$, $SD = 8.80$) than for *ad hoc*-generated outputs ($M = 93.33$, $SD = 11.44$). A repeated-measures Analysis of Variance (ANOVA) [18] test showed a statistically significant main effect of refinement method on output quality ($F(1,4) = 0.22, p = 0.651$), suggesting that the systematic PDR method leads to higher-quality outputs, on average.

Fig. 5 presents a per-participant comparison of average quality ratings between *ad hoc* and PDR methods.

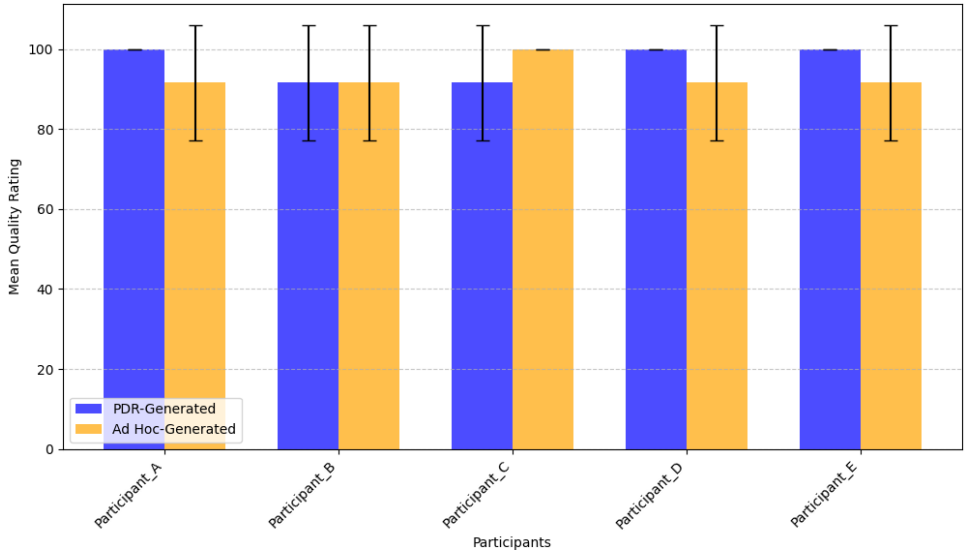

Fig. 5. Per-participant mean quality ratings for outputs generated with ad hoc and PDR methods (error bars indicate standard deviation).

While PDR yielded higher average quality scores, the overlap in standard deviations indicates that individual differences in refinement strategies may influence the results. This finding suggests that although PDR provides a structured approach to refinement, its effectiveness may depend on the complexity of the task and the refinement process applied by each participant.

4.3.4. User Satisfaction (RQ3)

To assess user satisfaction, we measured subjective ratings of prompt effectiveness and ease of refinement. Simulated GPT-4o participants rated the PDR method slightly higher in terms of ease of use and effectiveness, though the difference was not statistically significant. Satisfaction scores were reported on a 5-point Likert scale, where the PDR method had an average score of 4.8 ($SD = 0.4$), while the *ad hoc* method averaged 4.7 ($SD = 0.5$). A Mann-Whitney U test indicated that this difference was not statistically significant ($U = 128, p = 0.193$), suggesting that while the PDR method may improve user experience, the observed satisfaction difference may reflect participant variability rather than a consistent effect.

Fig. 6 compares the average user satisfaction ratings for the *ad hoc* and PDR refinement methods.

While the mean satisfaction rating for PDR was slightly higher than the *ad hoc* method, the overlapping variability suggests the perceived usability and effectiveness of both approaches were comparable. The lack of statistical significance further supports the interpretation that individual differences in how participants engaged with each refinement strategy may have contributed to the observed scores.

Table 2 summarizes the metrics, showing PDR reduced iteration count and raised quality scores, but increased total refinement time. These results show how PDR helped novices more than experts (similar to the findings in [19]), likely because it provides more guidance where intuition was weaker. However, while PDR increased output quality, it required more time, highlighting the quality/efficiency trade-off.

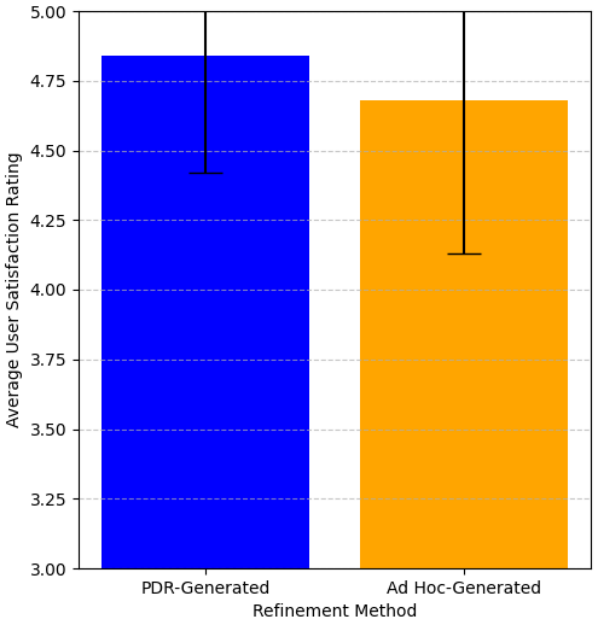

Fig. 6. Average user satisfaction ratings for outputs refined with ad hoc and PDR methods (error bars represent standard deviation).

Table 2. Summary of Key Results

Metric	Ad hoc	PDR	p-value
Iterations	3.3	2.5	0.145
Time (seconds)	122.6	288.8	0.062
Quality Score	93.33	96.67	0.651
Satisfaction (Likert)	4.7	4.8	0.193

4.4. Threats to Validity

We acknowledge several threats to validity that could limit the generalizability of the findings from our experiments. In particular, while simulation-based experiments provide reproducibility and systematic evaluation, the following limitations should be considered:

- *Simulated users vs. real users* – Using GPT-4o personas as proxies for human participants may not capture the full range of cognitive variability, decision-making behavior, or subjective preferences observed in real users. Human users may interpret rubric criteria differently or prioritize different factors in prompt outputs.
- *Evaluation bias* – While rubric-based scoring promotes consistency, it depends on an LLM's ability to self-assess outputs using predefined criteria. This reliance may introduce circularity or bias, particularly when the same LLM family is used for both generation and evaluation tasks.
- *Limited domains* – Our experiments focused primarily on software development prompts such as API documentation, code comment generation, and onboarding materials. Our findings may not generalize to other technical or non-technical domains, which represents a potential limitation.
- *Prompt overfitting* – Refining prompts against a fixed rubric may lead to overfitting by aligning outputs too closely with the rubric at the expense of broader applicability.

Future research should address these limitations by exploring real-user evaluations, extending testing to specialized domains, applying engineering-specific rubrics assessed by experienced software engineers, and training LLMs on software documentation and architecture patterns. Acknowledging these threats

helps contextualize our results and identifies opportunities for improving the robustness of PDR evaluation.

5. Related Work

Prompt engineering has been studied extensively as a means of improving LLM performance across diverse tasks. Early prompt design in software engineering was largely *ad hoc*, relying on developer intuition, manual formatting, and repetitive trial-and-error to produce usable outputs. As LLMs become integral to automated software workflows—including unit test generation, API documentation synthesis, static analysis, and code scaffolding—the need for structured prompt engineering techniques capable of reliably producing high-quality, policy-compliant outputs has grown.

This section reviews existing research in three key areas relevant to automated software engineering: *prompt engineering methodologies* for systematically guiding LLM behavior in software development tasks, *preference-driven refinement techniques* that allow developers to encode team standards and best practices into prompt iterations, and *evaluation of user interaction with LLMs* to assess the usability and consistency of AI-generated development artifacts.

5.1. Prompt Engineering Methodologies

Prompt engineering is an essential strategy for optimizing LLM outputs. Brown et al. [20] demonstrated that models like GPT-3 can achieve state-of-the-art performance across diverse tasks using *few-shot*, *zero-shot*, and *chain-of-thought* prompting techniques [21]. Subsequent studies have further refined these methods by introducing *instruction tuning* [4] and *self-refinement techniques* [22], where the model iteratively refines its own responses.

Our work builds on related work by introducing *Preference-Driven Refinement* (PDR) of prompts. This PDR method provides a structured framework that explicitly incorporates user preferences into iterative prompt modifications. Unlike *ad hoc* or trial-and-error methods, PDR provides a systematic mechanism for prompt refinement that aligns more closely with structured instruction tuning.

5.2. Preference-Driven Refinement and In-Context Learning

In-context learning advances have enabled LLMs to adapt to engineering-specific requirements, such as writing unit test scaffolds, documenting modules, or generating CI/CD configurations based on user-provided examples [23]. Approaches like *example-based prompting* [24] and *demonstration selection* [25] demonstrate that models can generate more relevant and accurate responses when guided by well-designed prompts.

Our PDR method extends these findings by leveraging both *in-context learning* and *synthetic example generation* to refine prompts. Unlike static examples, PDR dynamically adjusts the refinement process based on explicit user feedback, creating a more adaptive and interactive prompt engineering workflow. This capability parallels work in interactive reinforcement learning [26], where human preferences help guide model optimization.

5.3. Evaluating User Interaction and Refinement Strategies

Recent studies have examined developer satisfaction and refinement efficiency in LLM-assisted code and documentation generation workflows. Research on human-AI interaction [27] suggests that structured refinement strategies can improve usability and reduce cognitive load. However, studies like those by Reynolds and McDonell [9] highlight the challenges of *ad hoc* prompt engineering, where inconsistent modifications often yield suboptimal performance.

Our experiment results shown in Section 4 contribute to this discourse by demonstrating that while PDR improves structured refinements and output quality, it introduces efficiency trade-offs. Specifically, we

found that while PDR reduces the number of iterations, it requires significantly more time per task. This finding highlights the need to further explore the trade-offs between refinement complexity and efficiency in prompt engineering workflows.

As summarized above, related work has explored various prompt optimization techniques, from instruction tuning to reinforcement-based methods. However, little work has been conducted to evaluate *preference-driven prompt refinement* systematically to improve LLM-generated responses. Our work fills this gap by introducing and evaluating PDR as a structured refinement approach. By empirically analyzing its impact on prompt efficiency, output quality, and user satisfaction, we provide insights into the benefits and limitations of structured refinement strategies in LLM interactions.

6. Concluding Remarks

This paper demonstrated that the iterative and systematic improvements enabled by our Preference-Driven Refinement (PDR) method directly support key goals in automated software engineering. Unlike *ad hoc* prompt engineering—which often produces inconsistent outputs and demands substantial manual effort—PDR offers a structured mechanism to align LLM-generated content with project standards and workflows.

Our experimental results indicated that the PDR method (1) reduced the number of prompt iterations needed to achieve high-quality outputs, and (2) enhanced response quality in software development tasks such as code summarization, test generation, and documentation drafting. These findings highlight the role of PDR in enabling more consistent, efficient, and automation-friendly use of LLMs within software engineering teams.

The following are key lessons learned based on our experience applying the PDR method in practice:

- *Iterative feedback loops improve output precision*. PDR increases alignment with user expectations by integrating feedback into each prompt refinement cycle. This iterative process is particularly effective in high-stakes software systems (such as security infrastructure, compliance automation, and cloud service deployment) where precision and consistency are paramount, demonstrating the PDR method's adaptability and scalability.
- *In-context learning strengthens developer control*. Integrating in-context examples within PDR allows for more deliberate and targeted prompt refinements. This technique allows LLMs to learn from examples provided within the prompt, simplifying the process for developers and producing more nuanced and context-aware outputs.
- *Synthetic example generation expands design space*. This technique supports PDR by allowing developers to explore a broader range of prompt possibilities without manually creating each variant. Applying synthetic example generation further supports the PDR method by enabling developers to explore a spectrum of possibilities without manually crafting each example. This combination of in-context learning and synthetic example generation facilitates a more user-centric approach to prompt engineering, addressing the challenges of instruction blindness and enabling developers to articulate their preferences through concrete examples.
- *Efficiency trade-offs merits deeper exploration*. While PDR improves refinement structure and user satisfaction, our findings show that it also increases the time required for prompt iteration. The benefits of structured refinement must therefore be weighed against efficiency trade-offs. Future studies should investigate how refinement complexity impacts efficiency across different use cases, balancing structured iteration with practical constraints.

As generative AI advances into specialized domains, the importance of systematic, user-centric prompt engineering continues to grow.

The PDR method introduced in this paper provides a robust framework for refining prompts that generate source code, configuration files, documentation, and technical summaries. PDR aligns LLM outputs with professional software engineering standards and team-specific practices.

We expect future work will enable structured, feedback-driven methods in LLM interaction, contributing to more reliable, adaptable, and efficient prompt engineering practices. We also plan to integrate our PDR method with emerging AI-augmented Integrated Development Environments (IDEs), such as Windsurf and Cursor.

References

[1] White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2024). ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for effective software development* (pp. 71-108). Springer.

[2] White, J., et al. (2023). A prompt pattern catalog to enhance prompt engineering with ChatGPT. In *Proceedings of the 30th Pattern Languages of Programming (PLoP) Conference* (pp. 1-12). Allerton Park, IL.

[3] Hao, S., et al. (2024). Synthetic data in AI: Challenges, applications, and ethical implications. *arXiv preprint arXiv:2401.01629.*

[4] Ouyang, L., et al. (2022). Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems* (pp. 27730-27744).

[5] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys, 55(9)*, 1-35.

[6] Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., & Chadha, A. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927.*

[7] Ramnath, K., et al. (2025). A systematic survey of automatic prompt optimization techniques. *arXiv preprint arXiv:2502.16923.* Retrieved from https://arxiv.org/abs/2502.16923

[8] Cheng, J., et al. (2024). Black-box prompt optimization: Aligning large language models without model training. *arXiv preprint arXiv:2311.04155.* Retrieved from https://arxiv.org/abs/2311.04155

[9] Reynolds, L., & McDonell, K. (2021). Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (pp. 1-7).

[10] Wang, Y., et al. (2023). Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560.* Retrieved from https://arxiv.org/abs/2212.10560

[11] Min, S., et al. (2022). Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837.*

[12] Islam, R., & Moushi, O. M. (2024). GPT-4o: The cutting-edge advancement in multimodal LLM. *Authorea Preprints.*

[13] Sindhu, B., Prathamesh, R. P., Sameera, M. B., & KumaraSwamy, S. (2024). The evolution of large language models: Models, applications and challenges. In *Proceedings of the 2024 International Conference on Current Trends in Advanced Computing (ICCTAC)* (pp. 1-8). IEEE.

[14] Schmidt, D. C., Spencer-Smith, J., Fu, Q., & White, J. (2023). Cataloging prompt patterns to enhance the discipline of prompt engineering. Position paper.

[15] Benharrak, K., Zindulka, T., Lehmann, F., Heuer, H., & Buschek, D. (2024). Writer-defined AI personas for on-demand feedback generation. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (pp. 1-18).

[16] Student. (1908). The probable error of a mean. *Biometrika, 6(1)*, 1–25.

[17] Wilcoxon, F. (1992). Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution* (pp. 196-202). Springer.

[18] Fisher, R. A. (1970). Statistical methods for research workers. In *Breakthroughs in Statistics: Methodology and Distribution* (pp. 66-70). Springer.

[19] Brynjolfsson, E., Li, D., & Raymond, L. (2025). Generative AI at work. *The Quarterly Journal of Economics, 140(1)*, qjae044.

[20] Brown, T., et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems* (pp. 1877-1901).

[21] Wei, J., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems* (pp. 24824-24837).

[22] Madaan, A., et al. (2024). Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems* (pp. 1-10).

[23] Dong, Q., et al. (2022). A survey on in-context learning. *arXiv preprint arXiv:2301.00234.*

[24] Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., & Chen, W. (2021). What makes good in-context examples for GPT-3? *arXiv preprint arXiv:2101.06804.*

[25] Rubin, O., Herzig, J., & Berant, J. (2021). Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633.*

[26] Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems* (pp. 4299-4307).

[27] Chu, S., Kim, J., & Yi, M. (2024). Think together and work better: Combining humans' and LLMs' think-aloud outcomes for effective text evaluation. *arXiv preprint arXiv:2409.07355.* Retrieved from https://arxiv.org/abs/2409.07355