Subjective Code Preferences in Experts and Large Language Models

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have become increasingly popular for coding tasks, with subjective coding preferences being an essential element to adapt to programmers' personal needs. Existing work overlooks such characteristics and mainly focuses on code correctness. In this study, we propose a typification of four subjective coding preference axes — complexity, commenting, modularity, and readability — motivated by common engineering habits and validated by 25 software engineers. We collect a dataset of $\approx 3,000$ paired Python code snippets reflecting these axes, annotated by 73 experts who rate their preferences on a Likert scale. Using our dataset, we study how LLMs handle subjective coding preferences. We present 13 LLMs with pairs of solutions to the same programming task, first as textual descriptions and then as concrete code snippets. We find that models often prefer one option in natural language but the opposite when evaluating code. More consistent models (i.e., those that are coherent in their choices between deeds and words) frequently reveal positional bias: swapping the order of options changes the preferred alternative. We then use the five most consistent models to re-annotate the dataset. Compared to humans, models show polarized Likert distributions and notable divergence in ratings. A case study on GPT-5 reveals reliance on external assumptions and brittle reasoning.[1]

1 Introduction

In recent years, LLMs have become broadly used as coding tools (Ferino et al., 2025). To measure their coding capabilities, there is a need to assess the models' prior knowledge and skills. This is required to differentiate between the system's fundamental understanding of the task and its generalization power (Chollet, 2019), which might be specious at its logical core.

LLM evaluation in the coding domain typically involves benchmarks that target objective aspects such as code correctness and security (Weyssow et al., 2026; Liu et al., 2024). This, however, limits the estimation to a "correct vs incorrect" paradigm and does not explain how LLMs behave when subjective coding preferences come into play. This is especially critical since LLMs can be inconsistent in opinion-based or value-related domains (Xu et al., 2025). Focusing only on objective criteria also contradicts the fundamental definition of a *preference* (Brafman and Domshlak, 2009), which is a set of *alternative* choices, where a more desirable choice precedes a less desirable one. Finally, evaluating code quality does not directly validate the model's core understanding of the task that it is performing (Moskvichev et al., 2023).

In this work, we provide an extensive investigation of LLMs' behavior across subjective coding preferences (Figure 1). First, we introduce a set of coding preferences axes that are subjective (i.e., both preference values produce a working code) but can still be measured and shown from the previous work to be important for software development (Tashtoush et al., 2023; De Souza et al., 2005): commenting, modularity, readability, and complexity. We then generate a novel dataset of ≈ 3000 programming problems with paired solution snippets, where each snippet of a pair represents a different value on our preference axes (e.g., more vs less commented code to solve the same problem). This dataset is then fully annotated by 73 coding experts, using a Likert-6 scale to express their subjective preferences for each pair. Borderline annotations (slight preferences) are enriched by experts with textual rationale.

We use our generated dataset to analyze the behavior and alignment of 13 LLMs in two setups. First, we test if LLMs are consistent in expressing a utility at a textual level (e.g., "a more/less

[1] Data available at https://CodePref-B07D

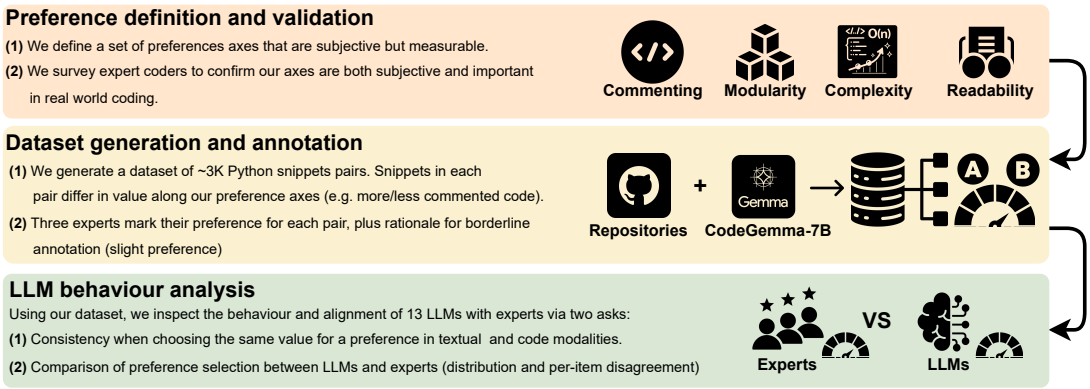

Figure 1: Overview of our dataset construction and experimental pipeline.

commented code") and code level (coding snippet solving the problem with more/less comments). All tested LLMs show severe inconsistency, choosing a specific textual preference, but then picking the opposite coding snippet, and most of the evaluated models exhibit positional bias.

Second, we let LLMs replicate our dataset annotation, and compare their preference distribution with that of experts: most models predominantly choose the extremes of our selection scale), compared to the more balanced one of our experts. GPT-5, the only exception, more closely models the overall human distribution, but still shows severe divergence on a per-item basis. We also run a case study where we qualitatively inspect cases of disagreement between experts and GPT-5, showing reliance on external assumptions and brittle reasoning, culminating in ignoring core knowledge necessary for a correct assessment.

To summarize, this paper provides the following three contributions (Figure 1):

1. A novel dataset of Python coding snippets showcasing subjective preferences, validated and annotated by expert software engineers (SE) (§3).
2. First assessment of LLM consistency over subjective coding preferences, across natural language and code modality (§4).
3. Analysis of the alignment and behavior between LLMs and humans with respect to subjective coding preferences (§5).

2 Related Work

Coding Preferences Previous works related to LLMs' coding preferences (Weyssow et al., 2026; Liu et al., 2024) inspect objective criteria, such as security or correctness (Liu et al., 2024; Weyssow et al., 2026). These studies do not reflect the complexity of decisions and tradeoffs that a developer experiences (Dutta et al., 2025). This tendency is also mirrored in existing datasets like CodeUltraFeedback (Weyssow et al., 2026), CodePrefBench (Liu et al., 2024), and OpenCodeInstruct (Ahmad et al., 2025), which focus on tuning LLMs for software development problems, but not on preference alignment. It results in LLMs developing towards code generation correctness and ignoring humans' priority for a more nuanced and complex approach to software development (Azeem et al., 2025). Some examples of the importance of this multi-criteria analysis include readability's enhancement of software maintainability (Buse and Weimer, 2009; Tashtoush et al., 2023), or development optimization across multiple objectives (Li and Chen, 2024). While some previous research integrates benchmarking to assess coding criteria beyond correctness (Zheng et al., 2024); it does not provide validation from the side of the end users, as opposed to our study.

Human-LLM preferences comparison Being trained on vast amounts of natural language data, LLMs show similar decision heuristics to humans in text generation (Suri et al., 2024), including pre-existing biases and conceptual behavior tendencies. However, this human-like pattern is not exhibited in controlled binary-choice experiments, where the decision-making requires broader narrative support (Jia et al., 2024) and is systematically distinguished by positional bias (Shi et al., 2024). This leads to LLMs being more prone to hesitating and admitting their limits in the presence of a clear dislike (Li

et al., 2024). Additionally, AI systems are argued to better maximize the satisfaction of preferences and uncover hidden contexts by integrating normative reasoning with machine learning (Zhi-Xuan et al., 2025). In this experiment, we enrich borderline preferences with textual rationale to gain a better, first-of-its-kind understanding of how humans and LLMs ground their choices with regard to decision-making in the coding domain.

LLM Alignment on Code and Natural Language Prompt-based inference (Song et al., 2023; Wei et al., 2022) has been shown to enhance LLMs' performance across a variety of coding tasks, mostly generation (Jain et al., 2022; Weyssow et al., 2025) and testing (Kang et al., 2023; Lemieux et al., 2023). LLMs also systematically fail in correctly performing "description-to-code" evaluations due to inherent biases and hallucinations within LLMs (Jin and Chen, 2025). There is no up-to-date proof of the positive contributions of these methodologies to LLMs' alignment between words and coding deeds. We address this in our study by exploring different prompting setups.

3 Data

3.1 Coding Preferences

We define four coding preference axes, designed to be subjective but measurable. This means that choosing one value of the axis does not affect the code correctness; instead, the choice for each preference should be motivated by the individual professional experience, project requirements, or specific industry protocols. For our axes, we model:

Commenting Appropriate code commenting benefits code maintenance, and speeds up debugging. However, excessive or redundant comments can be detrimental. *'Commenting'* utility represents the extent to which a code snippet contains commented lines, including the code documentation (Rani et al., 2023; Perera et al., 2023).

Consider a code snippet $\mathcal{S}$, with $\mathcal{L}_{\text{total}}(\mathcal{S})$ total lines, and $\mathcal{L}_{\text{comment}}(\mathcal{S}) \subseteq \mathcal{L}_{\text{total}}(\mathcal{S})$ commented lines. The commenting score $CS(\mathcal{S})$ is:

$$\text{CS}(\mathcal{S}) = \frac{|\mathcal{L}_{\text{comment}}(\mathcal{S})|}{|\mathcal{L}_{\text{total}}(\mathcal{S})|}.$$

Modularity Code modularity refers to the practice of breaking down a large program into smaller, independent, and reusable units called modules. Making a correct and balanced use of this coding feature can improve code organization, re-usability, and maintainability. Modularity is typically measured by the number of separate modules and functions, divided by the total number of code lines (hug, 1989; Tempero et al., 2023). Specifically, given a snippet $\mathcal{S}$, with $\mathcal{M}(\mathcal{S})$ distinct modules and functions, and $\mathcal{L}_{\text{total}}(\mathcal{S})$ lines of code, the modularity score of $\text{MS}(\mathcal{S})$ is:

$$\text{MS}(\mathcal{S}) = \frac{|\mathcal{M}(\mathcal{S})|}{|\mathcal{L}_{\text{total}}(\mathcal{S})|}.$$

Complexity The degree of complexity of a code can have a profound effect on maintainability (Tashtoush et al., 2023), developer productivity, software quality, performance, and overall business cost. For this preference, we consider coding snippets with opposite complexity codes in terms of big O-notation (Sikka et al., 2020). Consider a program $\mathcal{P}$ with input size n, $T_{\mathcal{P}}(n)$ time complexity, and $S_{\mathcal{P}}(n)$ space complexity functions. The asymptotic complexity class of $\mathcal{P}$ is characterized by:

$$\mathcal{C}(\mathcal{P}) = (\mathcal{O}(T_{\mathcal{P}}(n)), \mathcal{O}(S_{\mathcal{P}}(n))),$$

Readability Code readability is a fundamental coding characteristic (Börstler et al., 2023; Wyrich et al., 2023) that defines developers' preferences and values in structuring their code. However, it is very subjective and experience-based, without an established unified measurement (Scalabrino et al., 2017). One definition of readability is linked to judging how easy a text is to understand (Buse and Weimer, 2009), which can be represented in overt code features (e.g., average or maximal line length, number of loops, etc.). On the other hand, discussions on this topic (Rosenberg et al.; chi, 1994; Oliveira et al., 2024) agree that following the coding standards, cyclomatic complexity, and code maintainability are also associated with the concept of readability. Information on our proposed Readability metric, which combines above-mentioned factors, can be found in the Figure 10 and in the Appendix A.

Axes validation The goal of our work is to investigate preferences that matter in real-world coding. Therefore, we validate our four axes through an online survey with experts (recruitment details in Appendix B; full survey in Appendix C). We recruit 25 coders from Upwork[2] with heterogeneous pro-

[2] upwork.com

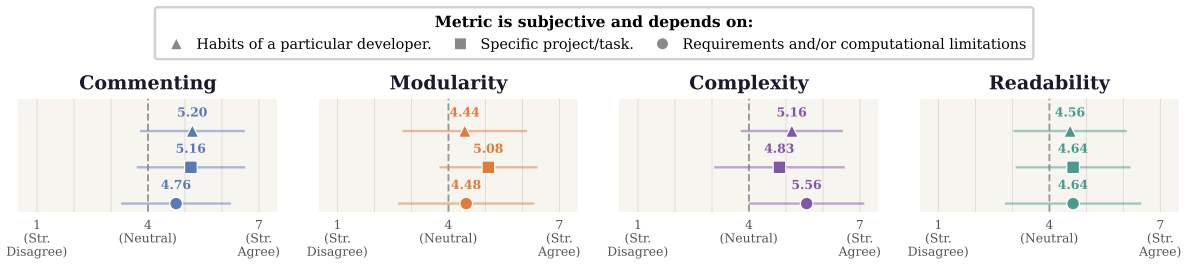

Figure 2: Responses from the experts' survey for each preference: aggregated values, along with context-specific importance (project, requirements, or habits). Appendix C contains the full questionnaire and Likert distribution.

Table 1: Participant background and coding context.

Category	Share
Development background	
Backend development	28%
Frontend development	24%
Full-stack development	20%
Machine learning engineering	16%
Other fields	12%
Coding context	
Company projects	88%
Academic or freelance development	12%

Table 2: Dataset composition by preference axis.

Axis	Portion	Snippets pairs
Commenting	26%	765
Complexity	35%	1,030
Modularity	24%	706
Readability	15%	442
Total	**100%**	**2,944**

fessional background (tab:participant-background). We ask the experts to rate, on a Likert-7 scale, the importance of our four axes and their dependency on personal coding habits, project constraints, and requirements. From results (Figure 2), all metrics are confirmed as influenced by company/project requirements, and experts show average agreement on their importance in different software engineering environments.

3.2 Generating the Dataset

Next, we realize our preferences axes as a dataset $\mathcal{D}$ of paired code snippets

$$\mathcal{D} = \{(p, (s_1, d_1), (s_2, d_2))\},$$

where p is a programming problem description, s_i is a code snippet that solves p, and d_i is a structured description of s_i with respect to a specific evaluation metric ("e.g., minimal VS extensive commenting"). Each pair (s_1, s_2) represents alternative solutions to the same problem p, annotated by their corresponding metric-based characterizations (d_1, d_2).

For Commenting and Modularity, we first extract source code snippets from existing Python programming datasets (Cassano et al., 2023; Muennighoff et al., 2024; Tarun, 2021; Kaggle, 2023). We select snippets that score low on our metrics (e.g, for *Modularity*, we select only those samples that do not contain any modules, thus with $\mathrm{MS}(\mathcal{S}) = 0$). Then, we prompt CodeGemma-7B (Team et al., 2024) to generate the opposite of the given snippet (as in, high-scoring), and manually review the resulting code (prompt in Figure 7 in Appendix D). We manually review the generated snippets to make sure they do not stylistically diverge from the original ones (more details in Appendix E).

For Complexity and Readability, coding pairs are acquired from open-access coding competitions repositories (The Algorithms — GitHub Organization, 2026; GeeksforGeeks, 2025) and are fully human-written; thus, no LLM is used for code generation. The final dataset contains 2944 coding pairs (statistics in Table 2). We also conduct statistical testing, verifying that the produced coding snippets score significantly and consistently different on our metrics (Appendix F).

3.3 Human Preference Annotation

To compare humans and LLMs, we collect human preferences annotations for our dataset. We hire experts from Prolific[3], with a background in software

[3] prolific.com

A — Use minimal commenting

```python
def Fibonacci_Generator(n):
    if n<0:
        print("Incorrect input")

    # First and Second Fibonacci numbers are
    # 0 and 1 respectively
    elif n==1:
        return 0
    elif n==2:
        return 1
    else:
        return Fibonacci_Generator(n-1)
              +Fibonacci_Generator(n-2)
```

B — Use extensive commenting

```python
# Define a recursive function to generate
# Fibonacci numbers
# The function takes an integer `n` as input
def Fibonacci_Generator(n):
    # Handle incorrect input (negative numbers)
    if n<0:
        print("Incorrect input")

    # First and Second Fibonacci numbers are
    # 0 and 1 respectively
    # Base case: if n is 1, return 0
    elif n==1:
        return 0
    # Base case: if n is 2, return 1
    elif n==2:
        return 1
    # Recursive step: return the sum of the
    # previous two Fibonacci numbers
    else:
        return Fibonacci_Generator(n-1)
              +Fibonacci_Generator(n-2)
```

Preferences

Annotator 1	**5**	Llama3-70B	**4**	GPT-5	**3**
Annotator 2	**1**	Qwen3-Coder-30B	**2**	DS-GRM-16B	**2**
Annotator 3	**1**	Claude-4.5	**2**	. . .	**...**

Figure 3: Examples of an instance from our dataset. Given a Python problem (e.g., generating the Fibonacci numbers), we provide (1) two solution snippets varying across a preference axis (e.g., more or less commented); (2) preference annotation from human experts and LLMs.

engineering and a degree in computer science. We verify their ability to differentiate preferences via a qualification task: given 8 coding pairs, experts are asked to mark those with a higher/lower degree of commenting/modularity or are more/less computationally complex. Only annotators with 100% passing rate are accepted.

Preference annotation In the main task (example of the annotation UI in Appendix G, Figure 11), annotators are shown (1) a textual description of a Python problem; (2) two solutions (code snippets) solving it; (3) a clarification of the differences between them at the preference axis level (e.g. "Snippet A/B uses minimal/extensive commenting"); (4) a brief explanation of the axis (e.g. "Commenting refers to the commented of docstring lines..."). Experts choose their preference among the two snippets, based on their working experience, via a Likert-6 scale (0 = Strongly prefer snippet A; 5 = Strongly prefer snippet B). The median option was deliberately omitted to obtain distinguishable preferences between A and B, since, otherwise, humans lean towards the neutral option (Kusmaryono et al., 2022), while LLMs incline towards hesitating (Li et al., 2024). Final numeric preference annotation is done by 73 annotators, where 27 experts annotated ≈176 coding pairs each, and 46 annotated ≈89 coding pairs each. Each coding pair is annotated 3 times by different experts, and code pairs were presented in a randomized order. Figure 3 shows an instance example (including LLMs preferences, which we cover in §5).

Rationale annotation Incorporating additional argumentation for preference data is helpful to uncover more advanced patterns in human and LLMs behaviour, as well as useful in datasets for preference alignment (Just et al., 2024; Liu et al., 2025a), like a dataset curated in this work. Therefore, we collect rationale annotation for coding pairs with midpoint preference choice[4] (in this case, 2 and 3, as for *Slightly prefer A/B*). Existing work either adopts free-text rationale (Gabriel, 2020) or preset templates where the annotators fill in specific blanks (Zhi-Xuan et al., 2025). We adopt both methodologies for completeness. To filter out unreliable annotators who may offload rationale writing

[4]Each pair is re-annotated for rationale by its original annotator.

to LLMs, we adopt a mix of metric checks, manual inspection, and comparison with popular vendor LLMs (see Appendix H). Overall, we obtain rationale annotation from 26 experts over 514 coding snippets.

4 Evaluating LLMs' consistency over subjective preferences

As a first experiment, we assess if LLMs can effectively understand subjective preferences when explicitly expressing them. Given a Python problem z from our dataset, we ask the LLM to choose between two preference options x or y (e.g. extensive or minimal commenting), to solve it. We do this with x, y represented as both text and code. Ideally, we expect the model to be consistent (e.g., "extensive commenting", and the snippet with more comments). This tells us to what extent LLMs recognize the code realization of a textual preference.

We note that our annotators are expected to have this ability, given their work experience. Regardless of this, they also demonstrate it in our qualification task with an acceptance rate of 100% accuracy to proceed with the main annotation task. We also note that human coders, during their job, do not explicitly state a preference: they apply it implicitly. An expert AI assistant, on the other hand, should motivate its choices, and a developer may even state their own preference before, to condition them (e.g. "keep in mind I prefer more densely commented code").

We evaluate 13 LLMs (Appendix G) across four categories. As **baseline** models, we include Llama-3-8B-Instruct (Llama3-8B) and Llama-3-70B-Instruct (Llama3-70B) (Grattafiori et al., 2024), DeepSeek-R1-Distill-Qwen-14B (DS-R1-14B), and Qwen3-8B and Qwen3-32B (Yang et al., 2025). As **coding** models, we use StarCoder2-15B (Lozhkov et al., 2024), DeepSeek-Coder-33B-Base (DS-Coder-33B) (Guo et al., 2024), and Qwen3-Coder-30B-Instruct (Qwen3-Coder-30B) (Yang et al., 2025). As **judge** models, we include Prometheus-13B (Kim et al., 2024) and DeepSeek-GRM-16B (DS-GRM-16B) (Liu et al., 2025b). Finally, we evaluate three **vendor** models: gpt-5-mini (GPT-5) (Singh et al., 2025), Claude-sonnet-4.5 (Claude-4.5) (Anthropic, 2025), and gemini-3-flash-preview (Gemini-3) (Google, 2025). We test three setups:

Baseline (BL): LLM is asked to specify the preferences in Session A, where the model selects a textual option, and Session B, where the model selects a code snippet. Session B is new and holds no memory of Session A.

One-chat (OC): Both choices are made in the same session, ruling out forgetting the textual explanation that can be possible in the *Baseline* setting.

Few-Shot (FS): We introduce 2 input-output examples of paired textual and coding preferences to the prompt.

The specific prompts can be found in Appendix D. Additionally, we re-run each setup by swapping the order of the two snippets to investigate positional bias. We calculate *Consistency Score* (CS) as the correspondence between preference text and code choices:

$$\text{CS} = P(Q_t, Q_c) \sim D\big(\text{LLM}(Q_t) = \text{LLM}(Q_c)\big)$$

where $P(Q_t, Q_c)$ denotes the joint preference over a text query Q_t and a code query Q_c, $\text{LLM}(\cdot)$ is the response of the language model to a given query, and $D(\cdot)$ is an indicator of agreement between the two responses. When we swap the snippets, we calculate *Swap Adjustment Score* (SAS), measuring LLMs' consistency when the order of preferences is shifted. This metric denotes the percentage of cases where the model chooses the same option both in the ordered and the shifted dataset. Thus, such cases, when the choice for the first setup is *A*, and the choice for the shifted setup is *B*, which represent the same unit:

$$\text{SAS} = \Pr\big(\text{LLM}(Q_{\text{ordered}}) \neq \text{LLM}(Q_{\text{shifted}})\big)$$

where Q_{ordered} and Q_{shifted} denote the query with the original and swapped snippet ordering, respectively, and $\Pr(\cdot)$ is the probability over the evaluated cases.

From results (Table 3), models consistently pick contradicting textual and code preferences. This is evident from the generally low CS values. When a very high CS appears, this is due to positional bias. For example, Qwen3-8B, Qwen3-32B, and DS-R1-14B reach $CS > 80$. However, once we swap the order of the snippets, the models do not change their choice, and SAS drops dramatically. Interestingly, CS and SAS show opposite trends in GPT-5 and Gemini-3, showing more positional

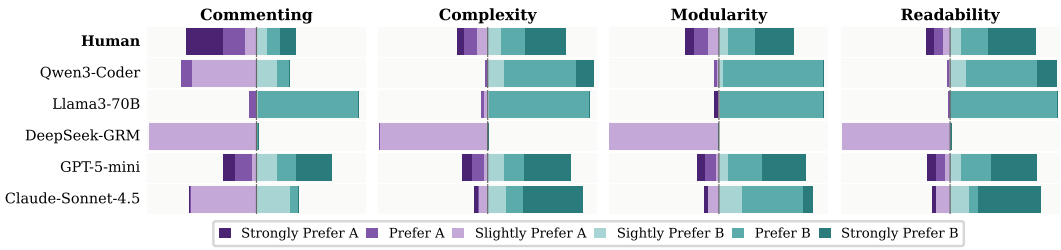

Figure 4: Distribution of preference choices among LLMs and humans.

Open LLMs	BL		OC		FS	
	CS↑	SAS↑	CS↑	SAS↑	CS↑	SAS↑
Llama3-8B	0.35	0.14	0.36	0.15	0.67	0.48
Llama3-70B	0.68	0.55	0.67	0.63	0.72	0.63
Qwen3-8B	0.90	0.00	1.00	0.00	0.96	0.00
Qwen3-32B	0.80	0.20	1.00	0.00	0.81	0.00
DS-R1-14B	0.92	0.00	1.00	0.00	0.99	0.00
DS-Coder-33B	0.53	0.02	0.73	0.04	0.73	0.17
StarCoder2-15B	0.26	0.07	0.52	0.24	0.51	0.03
Qwen3-Coder-30B	0.50	0.40	0.57	0.44	0.63	0.53
Prometheus-13B	0.11	0.00	0.52	0.08	0.99	0.00
DS-GRM-16B	0.49	0.13	0.80	0.17	0.45	0.04
Vendor LLMs	BL		OC		FS	
	CS↑	SAS↑	CS↑	SAS↑	CS↑	SAS↑
GPT-5	0.25	0.63	0.21	0.61	0.29	0.65
Claude-4.5	0.27	0.35	0.15	0.57	0.18	0.25
Gemini-3	0.35	0.55	0.33	0.60	0.30	0.67

Table 3: LLMs' CS (↑) and SAS (↑) across different prompting approaches. We mark **best** and **worst** per column.

consistency in their choices. We also note that OC and FS prompting do not improve the situation: some models benefit from it in terms of CS, but with little-to-no change in SAS.

5 Exploring Preference Behavior among LLMs and Humans

Next, we compare the distribution of preference values between humans and LLMs. To manage the computational and inference cost of our experiments and based on API availability, we select 3 open-weight and 2 vendor LLMs based on combined CS and SAS (Table 3), which is calculated as a summary of these two metrics in the Baseline setup and motivated by the intuition that models should have high CS and high SAS to avoid positional bias. We choose Llama3-70B (baseline model); Qwen3-Coder-30B (coding model); DS-GRM-16B (judge model); Claude-4.5 and GPT-5 (vendor models). We prompt these models by replicating our human annotation setup (Section 3.3), meaning that LLMs are now expressing a preference on a 6-scale, given the two snippets and a description of the axis value.

Disagreement We first compute the distribution of human Likert rankings from Section 3.3. For each coding pair i, we define a per-item disagreement score a_i: for human annotators, it is the average pairwise absolute difference across all $\binom{n_i}{2}$ annotator pairs (j, k), where h_{ij} and h_{ik} are their respective Likert ratings; for an LLM, it is the absolute deviation of its response ℓ_i from the human mean m_i. The overall Disagreement D is then the mean of a_i over all N coding pairs:

$$a_i = \begin{cases} \frac{1}{\binom{n_i}{2}} \sum_{1 \le j < k \le n_i} |h_{ij} - h_{ik}| & \text{for human annotation} \\ |\ell_i - m_i| & \text{for LLM annotation} \end{cases}$$

$$D = \frac{1}{N} \sum_{i=1}^{N} a_i$$

Via this metric, we compute D among humans, and between humans and LLMs. We also compute Jensen-Shannon Divergence (lin, 1991) between humans and LLMs. We choose this measure over the others (such as KL divergence (kul, 1951)) as it is symmetric and bounded, offering better interpretation of its values. This is in line with other works in the field (Go et al., 2023; Wang et al., 2024). We intentionally avoid metrics for measuring inter-annotator agreement, as this work focuses on *subjective* and targeting a gold standard for per-item LLM comparison would lead to a paradox.

Results from the distribution (Figure 4) and metrics (Table 4) reveal how critical it is to assess alignment via multiple metrics. GPT-5 achieves the lowest JSD scores, with a distribution of ratings closely mirroring the experts (except for commenting). However, it shows one of the highest D: despite producing realistic aggregate statistics, GPT-5 still frequently deviates from human judgement on individual examples. This indicates that

	Commenting		Complexity		Modularity		Readability		Overall	
Human VS	D	JSD	D	JSD	D	JSD	D	JSD	D	JSD
Human	1.93	–	1.67	–	1.81	–	1.49	–	1.74	–
Llama3-70B	2.29	0.63	1.30	0.55	1.36	0.56	1.18	0.60	1.54	0.57
Qwen3-Coder-30B	1.85	0.51	1.09	0.38	1.05	0.54	1.02	0.36	1.25	0.39
DS-GRM-16B	1.64	0.70	1.92	0.71	1.95	0.73	2.12	0.74	1.89	0.72
GPT-5	2.31	0.26	1.71	0.12	1.79	0.12	1.56	0.04	1.85	0.12
Claude-4.5	1.71	0.58	1.45	0.21	1.46	0.37	1.29	0.26	1.49	0.28

Table 4: Per-metric disagreement ($D \downarrow$) and Jensen-Shannon divergence ($JSD \downarrow$) between annotator pairs. We mark **best** and **worst** values for both D and JSD across metrics.

a model can learn the "shape" of human opinions without learning what drives human decisions on specific cases. Llama3-70B shows the opposite failure mode: systematic distribution biases (JSD), but low D. This shows how a consistent directional bias can accidentally align with human medians. Generally, all open LLMs show extremely skewed distribution; DS-GRM-16B consistently shows the worst results for each metric, which could be attributed to it being the smallest model. Additional statistical analysis of these results is in Appendix J.

Qualitative analysis Subjectivity can be trivial for judgments on Commenting and Modularity. Computational complexity, however, includes more intricate factors like worst-case complexity, expected input distribution, amortized complexity, or computational limitations. To understand the sources of disagreement along this axis, we perform a qualitative analysis of the experts' judgments compared to GPT-5, as it shows the strongest JSD (Table 4). We inspect 20 randomly selected code pairs with at least one human annotator disagreeing with GPT-5. First, we observe *reliance on external assumptions*: GPT-5 (alongside some experts) occasionally presumes additional contexts beyond the problem statement; for example, it favours code with better average case complexity, while experts may consider worst-case complexity (Appendix K, ID 1). Such hidden contexts of preferences can misdirect preference-learning (Siththaranjan et al., 2024), and are of utmost importance in case of expert preferences. Similarly, GPT-5 sometimes relies on auxiliary metrics beyond complexity, like commenting, readability, pure-pythonic vs mixed-implementations, and usage of certain libraries such as Numpy (K, ID 2-4). This sometimes leads to choosing an inefficient implementation due to surface-level simplicity. Finally, we observe consistent *brittle reasoning* in GPT-5: its rationale (K, ID 5) relies on superficial characteristics of the code, or even blatantly ignores known algorithmic complexity bounds, echoing prior findings (Şahinuç et al., 2026).

6 Conclusions

In this work, we inspected the behaviour of LLMs over subjective coding preferences, which are critical in expert coding, and often overlooked by traditional benchmarking. We introduced a novel dataset of coding snippets, that are expert-annotated with subjective preferences. We then used our dataset to identify patterns of behavior and consistency in LLMs. We find that LLMs cannot consistently identify the realization of subjective preferences, and struggle in identifying the same preference value when it's presented as text or code. We also find severe positional bias, showing that even seemingly robust models just learned fixed selection patterns. Vendor models are less affected by positional bias, but still fall far from desirable consistency. Compared to experts, most tested models show extremely polarized distribution, opting almost exclusively for the extreme of the proposed scale. GPT-5 closely resembles the human distribution, but still suffers from consistent per-item divergence. Our qualitative study further reveals problematic patterns in how the model reasons.

We release our dataset to foster research in preference learning and expert coder models. Our dataset can be used to integrate coding preferences into human reinforcement strategies for further development of coding assistants that follow users' personal choices. Additionally, acquired annotations and rationales can serve as a source for extended linguistic and statistical analysis for deeper insights into how LLMs understand code.

7 Limitations

Preference Matching and Preference Aggregation We focus on four coding preference axes. We do understand, however, that subjective preferences may widen into many other axes we did not consider. Some examples of those from the related literature and results of our survey include code *maintainability* and *security*. We also acknowledge that preference axes could aggregate in complex ways, entangling and co-influencing each other. Our current framework represents the single-principal alignment for an AI system to match one unique preference. We reserve the analysis of complex aggregation of multiple axes within single coding snippets for future work.

Programming Languages We limit our study to Python, excluding other programming languages. This choice was motivated by the extensive set of Python resources available online; however, other programming languages should be considered in future developments of our research line.

Model choices The choice of our 13 LLMs is certainly limited, especially when it comes to the rapid development of new models, scales, and training approaches. In this work, we restricted ourselves to those baseline, coding, judge, and vendor models that showed the strongest performance in popular coding benchmarks. This is a conscious choice that lets us analyze the models that are more likely to be used by software engineers, when it comes to both open and vendor LLMs. This was also a necessity to keep a manageable computational requirement and experimental runtime: testing each model on ≈ 3000 paired snippets, in three prompting setups, including the swapped case, imposed us to take this choice. The same logic made us narrow our selection in our second experiment, where we picked the 5 most consistent models. Additionally, this was motivated by the necessity of keeping the inference cost of vendor model within our budget (accounting also for the budge spent in human annotation).

Assumption of expert competence We consider our coding experts as the gold standard when it comes to real-world subjective preferences. However, we do not assume that our experts are always correct. We acknowledge that our qualification task, annotation setup, and sanity checks may be limited and not fully sufficient against all the challenges that come into the implicitly complex domain of subjective data annotation. We assume the trustworthiness of our expert based on our protocol, but also acknowledge that the convoluted nature of expert preference may need deeper analysis.

Application of the chosen binary scenario in coding assistants This work investigates the preliminary task and fundamental premise of the successful implementation of modern coding assistants. More specifically, contemporary agentic practices implicitly or explicitly rely on the agent's ability to understand what is preferred and or dispreferred. In the proposed setup, we directly evaluate if the LLM can understand the human's input. However, we acknowledge the possible limitation of this proxy and do not state that it is a logically sufficient guarantee of the downstream competence of coding agentic systems.

References

1951. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

1989. Why functional programming matters. *The computer journal*, 32(2):98–107.

1991. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151.

1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493.

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. 2025. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*.

Anthropic. 2025. Claude-sonnet-4.5. Large language model.

Samina Azeem, Muhammad Shumail Naveed, Muhammad Sajid, and Imran Ali. 2025. Ai vs. human programmers: Complexity and performance in code generation. *VAWKUM Transactions on Computer Sciences*, 13(1):201–216.

Jürgen Börstler, Kwabena E Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, and 1 others. 2023. Developers talking about code quality. *Empirical Software Engineering*, 28(6):128.

Ronen Brafman and Carmel Domshlak. 2009. Preference handling-an introductory tutorial. *AI magazine*, 30(1):58–58.

Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and 1 others. 2023. Can it edit? evaluating the ability of large language models to follow code editing instructions. *arXiv preprint arXiv:2312.12450*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Sergio Cozzetti B De Souza, Nicolas Anquetil, and Káthia M De Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75.

Subhabrata Dutta, Timo Kaufmann, Goran Glavaš, Ivan Habernal, Kristian Kersting, Frauke Kreuter, Mira Mezini, Iryna Gurevych, Eyke Hüllermeier, and Hinrich Schuetze. 2025. Problem solving through human–ai preference-based cooperation. *Computational Linguistics*, 51(4):1337–1372.

Samuel Ferino, Rashina Hoda, John Grundy, and Christoph Treude. 2025. Novice developers' perspectives on adopting llms for software development: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*.

Iason Gabriel. 2020. Artificial intelligence, values, and alignment: I. gabriel. *Minds and machines*, 30(3):411–437.

GeeksforGeeks. 2025. Complete a2z reference for dsa concepts. Last updated: 26 Jul 2025.

Dongyoung Go, Tomasz Korbak, Germán Kruszewski, Jos Rozen, Nahyeon Ryu, and Marc Dymetman. 2023. Aligning language models with preferences through f-divergence minimization. In *Proceedings of the 40th International Conference on Machine Learning*, pages 11546–11583.

Google. 2025. Gemini-3-flash. Large language model.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. In *Neural Information Processing Systems*. Curran Associates.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, YK Li, and 1 others. 2024. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231.

Jingru Jia, Zehua Yuan, Junhao Pan, Paul E McNamara, and Deming Chen. 2024. Decision-making behavior evaluation framework for llms under uncertain context. *Advances in neural information processing systems*, 37:113360–113382.

Haolin Jin and Huaming Chen. 2025. Uncovering systematic failures of llms in verifying code against natural language specifications. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 3819–3823. IEEE.

Hoang Anh Just, Ming Jin, Anit Sahu, Huy Phan, and Ruoxi Jia. 2024. Data-centric human preference with rationales for direct preference alignment. *arXiv preprint arXiv:2407.14477*.

Kaggle. 2023. Leetcode solutions and content kpis.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. 2024. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4334–4353.

Imam Kusmaryono, Dyana Wijayanti, and Hevy Risqi Maharani. 2022. Number of response options, reliability, validity, and potential bias in the use of the likert scale education and social science research: A literature review. *International Journal of Educational Methodology*, 8(4):625–637.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE.

Junlong Li, Fan Zhou, Shichao Sun, Yikai Zhang, Hai Zhao, and Pengfei Liu. 2024. Dissecting human and llm preferences. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1790–1811.

Miqing Li and Tao Chen. 2024. Methodology and guidelines for evaluating multi-objective search-based software engineering. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 707–709.

Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025a. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-sql. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21223–21261.

Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian

Wang. 2024. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025b. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Paakhi Maheshwari. 2024. Code snippets: Insights and readability.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. 2023. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pages 7604–7623.

Delano Oliveira, Reydne Santos, Benedito De Oliveira, Martin Monperrus, Fernando Castor, and Fernanda Madeiral. 2024. Understanding code understandability improvements in code reviews. *IEEE Transactions on Software Engineering*, 51(1):14–37.

DTWS Perera, HTM Premathilake, KPH Thathsarani, RHT Nethmini, DI De Silva, and HMPPKH Samarasekara. 2023. Analyzing the impact of code commenting on software quality. In *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE.

Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, 195:111515.

Linda Rosenberg, Ted Hammer, and Jack Shaw. Software metrics and reliability.

Furkan Şahinuç, Subhabrata Dutta, and Iryna Gurevych. 2026. Reward modeling for scientific writing evaluation. *arXiv preprint arXiv:2601.11374*.

Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE.

Lin Shi, Chiyu Ma, Wenhua Liang, Weicheng Ma, and Soroush Vosoughi. 2024. Judging the judges: A systematic investigation of position bias in pairwise comparative assessments by llms. *arXiv preprint arXiv:2406.07791*.

Jagriti Sikka, Kushal Satya, Yaman Kumar, Shagun Uppal, Rajiv Ratn Shah, and Roger Zimmermann. 2020. Learning based methods for code runtime complexity prediction. In *European Conference on Information Retrieval*, pages 313–325. Springer.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Anand Siththaranjan, Cassidy Laidlaw, and Dylan Hadfield-Menell. 2024. Distributional preference learning: Understanding and accounting for hidden context in rlhf. In *International Conference on Learning Representations*, volume 2024, pages 27448–27471.

Yisheng Song, Ting Wang, Puyu Cai, Subrota K Mondal, and Jyoti Prakash Sahoo. 2023. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *ACM Computing Surveys*, 55(13s):1–40.

Gaurav Suri, Lily R Slater, Ali Ziaee, and Morgan Nguyen. 2024. Do large language models show decision heuristics similar to humans? a case study using gpt-3.5. *Journal of Experimental Psychology: General*, 153(4):1066.

I. Tarun. 2021. Python code instructions 18k dataset.

Yahya Tashtoush, Noor Abu-El-Rub, Omar Darwish, Shorouq Al-Eidi, Dirar Darweesh, and Ola Karajeh. 2023. A notional understanding of the relationship between code readability and software complexity. *Information*, 14(2):81.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, and 1 others. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Ewan Tempero, Kelly Blincoe, and Danielle Lottridge. 2023. An experiment on the effects of modularity on code modification and understanding. In *Proceedings of the 25th Australasian Computing Education Conference*, pages 105–112.

The Algorithms — GitHub Organization. 2026. Open source resource for learning data structures algorithms and their implementation in any programming language.

Chaoqi Wang, Yibo Jiang, Chenghao Yang, Han Liu, and Yuxin Chen. 2024. Beyond reverse kl: Generalizing direct preference optimization with diverse divergence constraints. In *International Conference on Learning Representations*, volume 2024, pages 10450–10480.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. 2026. Codeultrafeedback: An llm-as-a-judge

dataset for aligning large language models to coding preferences. *ACM Transactions on Software Engineering and Methodology*, 35(3):1–36.

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2025. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–25.

Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM computing surveys*, 56(4):1–42.

Ruoxi Xu, Hongyu Lin, Xianpei Han, Jia Zheng, Weixiang Zhou, Le Sun, and Yingfei Sun. 2025. Large language models often say one thing and do another. *arXiv preprint arXiv:2503.07003*.

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*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *arXiv preprint arXiv:2407.11470*.

Tan Zhi-Xuan, Micah Carroll, Matija Franklin, and Hal Ashton. 2025. Beyond preferences in ai alignment: T. zhi-xuan et al. *Philosophical Studies*, 182(7):1813–1863.

A Additional information on coding metrics

Although there is no single objective criteria for readability, discussions on this topic agree on the following characteristics that are usually associated with how easy code is to understand: (1) Following coding standards, linters, pep8, etc.; (2) Cyclomatic complexity, which is a way of evaluating the code based on a built graph and number of nodes, edges and connected components; (3) Code maintainability, including above-mentioned cyclomatic complexity as well as relations between operators and operands; (4) Subjective human perception of how code looks which is based on local factors. These are the surface features that aim to capture human notions of readability and include such traits as number of comments, average line length, and loop count. We propose creating a combined readability model (Figure 10) that, on one hand, contains overt code features, and, on the other hand, represents

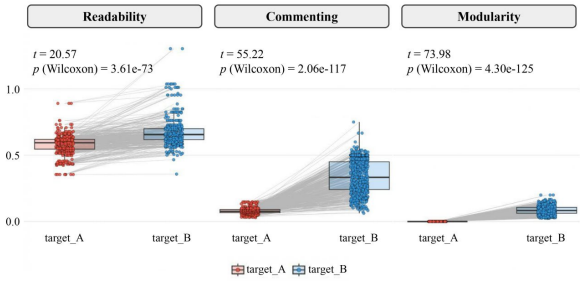

Figure 5: Distribution of code metrics differentiating Code_A and Code_B.

the established concepts of readability. For this purpose, a multiple linear regression is used with code local factors as independent variables and scores acquired by three methods that to some extent reflect readability as dependent variables on this data (Maheshwari, 2024). We then obtain weights for each of the local features, which allow us to reproduce the readability metric on any code snippet of any programming language. To filter code pairs from the dataset for the readability samples, we selected instances that were either more readable and less complex or less readable and more complex, using the above-described metric.

B Ethics Statement

In this section, we briefly discuss the ethical aspects of our experiments.

Ethical Review Before starting our experiments, we consulted our Institution for ethical pre-assessment. Since we do not collect any demographics and personal information about the participants, and crowdworkers do not disclose any data that could identify them, our experiment was confirmed to be ethically compliant without further procedures.

Data Consent For both recruitment processes done in this study, the survey and the main annotation task, we provided instructions explaining how the collected data would be used, the goal of the study, and ways to contact the study executors directly. Participants who wanted to take part in our experiments were required to give us explicit agreement before proceeding.

Workload and pay We estimated a maximum time for completing the expert survey of 30 min, and set an hourly rate per participant of 22€.

For the numeric preference annotation, we first conducted a pilot study, where we determined the av-

erage completion time as one hour for ~90 coding pairs. For the qualification task, which consisted of annotating 8 coding pairs, the required time per participant was 15 min and the hourly rate was 28€. For the full-scale annotation task, the hourly rate per annotator was set to 14€.

For the rationale preference annotation, participants are paid approximately 15€/hour, and the estimated (from the pilot study) workload is 6 coding pairs per hour.

Additional information on the recruited experts Other general information about participants of the *survey* includes:

1. 44% have 4-6 years of working experience; 28% have 7-9 years; 20% have 1-3 years; and 8% have 10+ years.

2. 48% spend approximately 3-5 hours a day on working with code; 44% spend 5-8 hours a day; 8% spend 1-3 hours.

C Survey Coding Styles of Software Engineers

Part 1: General information about experience and current work

Q1. How many years of working experience do you have as a developer?

Response options:

(a) 1-3 years
(b) 4-6 years
(c) 7-9 years
(d) 10+ years

Q2. What is your main specialisation?

Response options:

☐ Back-end
☐ Front-end
☐ Full Stack
☐ QA/testing
☐ ML engineering
☐ Other: _____

Q3. What are the main applications of your code?

Response options:

(a) Company projects
(b) Academic projects

Q4. How many hours a day do you spend on working with code on average?

Response options:

(a) 1-3 hours
(b) 3-5 hours
(c) 5-8 hours

Part 2: Coding Comments

Instructions:

You will be asked to rate several coding factors based on how relevant they are in your daily work and your personal experience. For your convenience and clearness, each coding factor is provided with a short definition.

"Coding comments" refer to the commented or docstring lines in code that usually explain what the code does

Example case: some code might contain a high number of comments explaining each step, while another way to implement the same code is with a moderate use of comments in the exceptional cases.

Q1. To what extent do you agree with the following statement regarding the use of coding comments?
1. Using different degree of comments in code (i.e. low or high) depends on the habits of a particular developer
2. In my daily coding activity I personally choose different degree of comments (i.e. low or high) depending on the specific project/task
3. Using different degree of comments in code depends on the requirements and/or computational limitations of the specific project
4. Using different degree of comments in code (i.e. low or high) doesn't depend on the habits of a particular developer

Response options: 1 = Strongly disagree, 2 = Disagree, 3 = Somewhat disagree, 4 = Neutral, 5 = Somewhat agree, 6 = Agree, 7 = Strongly agree.

Q2. Please mark here how you usually balance degree of comments in your code

Response options:

(a) I always use minimal commenting
(b) I mostly use minimal commenting
(c) I equally use minimal and extensive commenting
(d) I mostly use extensive commenting
(e) I always use extensive commenting

Q3. Feel free to share any other thoughts about this coding characteristic (coding comments)

Response options: Open-ended text.

Part 3: Coding Modularity

Instructions:

You will be asked to rate several coding factors based on how relevant they are in your daily work and your personal experience. For your convenience and clearness, each coding factor is provided with a short definition.

"Modularity" refers to the higher or lower usage of smaller self-contained modules (functions and classes)

Example case: some code might contain higher number of smaller self-contained modules, while another way to implement the same scripts is to write monolithic code

Q1. To what extent do you agree with the following statement regarding the use of coding modularity?
1. Modularity in code depends on the requirements and/or computational limitations of the specific project
2. Modularity in code doesn't depend on the habits of a particular developer
3. In my daily coding activity I personally choose different degree of modularity depending on the specific project/task
4. Modularity in code depends on the habits of a particular developer

Response options: 1 = Strongly disagree, 2 = Disagree, 3 = Somewhat disagree, 4 = Neutral, 5 = Somewhat agree, 6 = Agree, 7 = Strongly agree.

Q2. Please mark here how you usually balance modularity in your code

Response options:

(a) I always write monolithic code
(b) I mostly write monolithic code
(c) I equally write monolithic code and code that contains higher number of smaller modules
(d) I mostly write code that contains higher number of smaller modules
(e) I always write code that contains higher number of smaller modules

Q3. Feel free to share any other thoughts about this coding characteristic (modularity)

Response options: Open-ended text.

Part 4: Code Complexity

Instructions:

You will be asked to rate several coding factors based on how relevant they are in your daily work and your personal experience. For your convenience and clarity, each coding factor is provided with a short definition.

"Complexity" refers to the balance between computational resources and the amount of time taken by the code to run.

Example case: some code might be very quick to execute but requires a lot of memory, while another way to implement the same code is to save memory and sacrifice the running time

Q1. To what extent do you agree with the following statement regarding the use of coding complexity?
1. Complexity in code depends on the habits of a particular developer
2. Complexity in code depends on the requirements and/or computational limitations of the specific project
3. In my daily coding activity I personally make decisions regarding code complexity depending on the specific project/task
4. In my daily coding activity my decisions regarding code complexity don't depend on the specific project/task

Response options: 1 = Strongly disagree, 2 = Disagree, 3 = Somewhat disagree, 4 = Neutral, 5 = Somewhat agree, 6 = Agree, 7 = Strongly agree.

Q2. Please mark here how you usually balance complexity in your code

Response options:

(a) I always minimize computational resources and sacrifice speed of code execution
(b) I mostly minimize computational resources and sacrifice speed of code execution
(c) I equally balance computational resources and speed of code execution
(d) I mostly minimize speed of code execution and sacrifice computational resources
(e) I always minimize speed of code execution and sacrifice computational resources

Q3. Feel free to share any other thoughts about this coding characteristic (complexity)

Response options: Open-ended text.

Part 5: Code Readability

Instructions:

You will be asked to rate several coding factors based on how relevant they are in your daily work and your personal experience. For your convenience and clearness, each coding factor is provided with a short definition.

"Readability" refers to how easy code is to understand, "readability-complexity trade-off" implies the balance between how readable a code is and the computational costs to run it

Example case: some very optimised code might be very hard to read without deep understanding of the underlying framework, while another way to implement the same code is to sacrifice computational costs for better code readability

Q1. To what extent do you agree with the following statement regarding the use of coding readability-complexity trade-off?
1. Choices regarding coding readability-complexity trade-off depend on the requirements and/or computational limitations of the specific project
2. In my daily coding activity I personally make decisions regarding readability-complexity trade-off depending on the specific project/task
3. Choices regarding coding readability-complexity trade-off don't depend on the requirements and/or computational limitations of the specific project
4. Choices regarding coding readability-complexity trade-off depends on the habits of a particular developer

Response options: 1 = Strongly disagree, 2 = Disagree, 3 = Somewhat disagree, 4 = Neutral, 5 = Somewhat agree, 6 = Agree, 7 = Strongly agree.

Q2. Please mark here how you usually balance readability-complexity trade-off in your code

Response options:

(a) I always sacrifice computational resources for a more readable code
(b) I mostly sacrifice computational resources for a more readable code
(c) I equally balance computational resources and how readable code is
(d) I mostly sacrifice how readable code is and minimize computational resources

(e) I always sacrifice how readable code is and minimize computational resources

Q3. Feel free to share any other thoughts about this coding characteristic (readability-efficiency trade-off)

Response options: Open-ended text.

Part 6: Conclusion

Q1. What are the other coding factors that you're considering important at your daily work?

Response options: Open-ended text.

Q2. What are the other trade-offs in your daily coding decisions that you face? (i.e. produce memory-efficient code but sacrifice time of execution)

Response options: Open-ended text.

Q3. Please share any other thoughts or feedback about different coding styles and choices that developers come across to

Response options: Open-ended text.

D Prompts

We report the prompts we used in our experiments in Figures 6 to 9.

The selected prompts follow previous research on LLM preference among words and actions in different domains (Li et al., 2024). The selected prompts do not contain any distinctive stylistic features that would drastically influence LLMs' outputs. For example, none of the prompts has an articulated phrasing that is unique to the axes implementation. Nevertheless, the results remain consistent across model families, model sizes, and prompting configurations. This consistency strengthens the reliability of the findings, since it is unlikely that a specific prompt formulation would produce highly similar outcomes across all these conditions. Moreover, the majority of each prompt consists of fixed snippet pairs. Given the autoregressive nature of large language models, this further suggests that the comparatively small variable portion of the prompt is unlikely to substantially alter the model's behaviour.

```
The following two options represent solutions of the same Python
problem z.Which of these two options would you prefer?

Option A: x

Option B: y

Please respond with only A or B.
```

Figure 6: Prompt template to investigate LLMs' coding preferences. x and y are realized either as textual descriptions or code snippets solving a Python problem z.

```
Here is a code snippet $CODE that solves the following Python
problem: $PROBLEM. This code scores very low on $PREFERENCE.

Write the opposite of this code solution for the stated
problem so that your version will score high on $PREFERENCE

Here is the explanation of $PREFERENCE: $EXPLANATION
```

Figure 7: CodeGemma-7B Code Generation Prompt.

E Details on Dataset Curation and CodeGemma-7B Applications

In this section, we highlight that CodeGemma-7B did not generate the alternative code, but rather a modification that was manually checked to eliminate the potential differences that can be detected as "LLM-generated vs. human-written". Thus, the LLM is used as a verification tool, whereas we are not building a direct connection, stating that judgment is harder than verification. We also validate the generated snippets through controlled runs and compare their output against the starting ones. We obtain only $\approx 14\%$ of snippets with mismatched stderror outputs. Upon manual review, we discover that this is not due to logical or coding inconsistencies, but rather from randomly generated test variables or minor formatting differences. Therefore, we keep all the generated snippets. Considering the amount of manual work for CodeGemma-7B is also not considered a golden standard model for code

```
The following two options represent solutions of the same Python
problem z.Which of these two options would you prefer?

Option A: x

Option B: y

Please respond with only A or B.

Here are some examples:

1) Option A $CODE: x_code; Option B $CODE: y_code; Response:
x_code. Option A $TEXT: x_text; Option B $TEXT: y_text; Response:
x_text.

2) Option A $CODE: x_code; Option B $CODE: y_code; Response:
y_code. Option A $TEXT: x_text; Option B $TEXT: y_text; Response:
y_text.
```

Figure 8: Few-Shot Example Prompt.

```
$PROBLEM_TITLE

$PROBLEM_EXPLANATION

Which of these two codes would you most likely choose in
a daily work as software engineer?

Preference (0 = strongly prefer A, 5 = strongly prefer
B)

$PREFERENCE_EXPLANATION

Option A: $TEXT_A; $CODE_A

Option B: $TEXT_B; $CODE_B
```

Figure 9: Prompt for Human-like Preference Annotation.

generation. To show that CodeGemma-7B still fails at recognizing different preferential modalities, we extend our analysis in Table 3. The results show that the model replicated the inconsistent behavior observed by other models (Table 5).

Open LLMs	BL		OC		FS	
	CS↑	SAS↑	CS↑	SAS↑	CS↑	SAS↑
Llama3-8B	0.35	0.14	0.36	0.15	0.67	0.48
Llama3-70B	0.68	0.55	0.67	0.63	0.72	0.63
Qwen3-8B	0.90	0.00	1.00	0.00	0.96	0.00
Qwen3-32B	0.80	0.20	1.00	0.00	0.81	0.00
DS-R1-14B	0.92	0.00	1.00	0.00	0.99	0.00
DS-Coder-33B	0.53	0.02	0.73	0.04	0.73	0.17
StarCoder2-15B	0.26	0.07	0.52	0.24	0.51	0.03
Qwen3-Coder-30B	0.50	0.40	0.57	0.44	0.63	0.53
Prometheus-13B	0.11	0.00	0.52	0.08	0.99	0.00
DS-GRM-16B	0.49	0.13	0.80	0.17	0.45	0.04
CodeGemma-7B	1.00	0.00	0.30	0.00	0.76	0.32
Vendor LLMs	BL		OC		FS	
	CS↑	SAS↑	CS↑	SAS↑	CS↑	SAS↑
GPT-5	0.25	0.63	0.21	0.61	0.29	0.65
Claude-4.5	0.27	0.35	0.15	0.57	0.18	0.25
Gemini-3	0.35	0.55	0.33	0.60	0.30	0.67

Table 5: LLMs' CS (↑) and SAS (↑) across different prompting approaches. We mark **best** and **worst** per column.

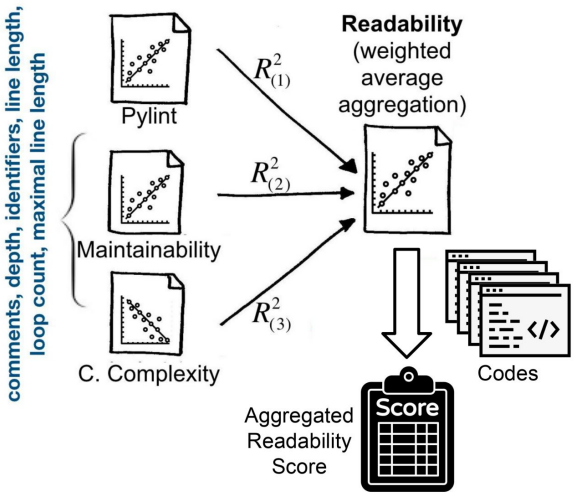

Figure 10: Readability Model Pipeline.

F Validating dataset variety

To inspect subjective preferences in clearly distinguishable scenarios, it is important that our dataset contains sufficiently different coding snippets for each pair. We verify this via a paired Wilcoxon signed-rank test based on the level of commenting (t=55.22; mean-diff=0.260), modularity (t=73.98; mean-diff=0.086), and readability (t=20.57; mean-diff=0.098). We obtain P-values $< 2.2e-16$ for each preference type, proving significant differences (Figure 5). For complexity, the paired t-test was not performed, as these samples are measured in big-O notation, so the benchmarking values are not numerical.

G Annotation Interface

This section demonstrates an example of UI interface for the main annotation task as it was displayed to participants (Figure 11).

H Sanity checks

Rationale annotation sanity check Before acquiring rationale from experts, we conduct a short probing rationale annotation study to understand the potential risks and limits for this kind of experimental setup. We contacted 4 experts who had already done the numeric annotation task, and asked them to ground their previous choices in the written form (2 coding pairs per each annotator). We compare the obtained answers with the responses from ChatGPT and Gemini by BertScore (Zhang et al., 2019), as well as review them manually, to eliminate the use of LLMs for this round of annotation. This verification showed that both open-ended questions and a template can serve as a reliable source for collecting human rationales. In addition, we apply a combination of metrics on the sequences of numeric annotations for each participant to exclude lazy workers or participants whose previous annotations have been done mechanically. Those metrics include: entropy, switch rate (how often an annotator changes their label), switch symmetry (whether the annotator switches symmetrically and does random left-right clicking), percentage of using two midpoint labels together and separately, and frequency of the most repeated annotation sequence. We use the following thresholds for each metric: entropy < 1.43; switch rate $< 80\%$; switch symmetry < 0.375; usage of each midpoint label separately $< 23\%$; usage of midpoint labels together $< 41\%$.

I Experimental Setup

The experimental setup for pairwise and 6-scale preference setting uses a fixed random seed across Python, NumPy, and PyTorch, and enables deterministic CUDA behavior to ensure reproducibility. For some models of larger size (e.g., Llama3-70B, Qwen3-32B, and DS-Coder-33B), inference is run with 8-bit quantization. We use greedy decoding (`do_sample=False`) for deterministic token selection. The remaining settings, such as left padding and a maximum of new generated tokens, are standard implementation choices motivated by enforcing either binary-choice or N-token generation output. In some cases, the `max_new_tokens` hyperparameter was heuristically increased to avoid truncated outputs and ensure the model produced a complete decision.

J Statistics on Preference Behaviour among LLMs and Humans

In this section, we report the results of a bootstrap analysis with 1000 resamples of code pairs, including confidence intervals, standard errors, and cross-LLM as well as LLM-human comparisons for Table 4. For this analysis, we added three worst-performing models across three model families based on the summary of CS and SAS scores, which were Llama3-8B, StarCoder2-15B, and Prometheus-13B.

Analysis showed that across all three evaluation metrics - Disagreement (D), Jensen–Shannon (JS) divergence, and Spearman rank correlation - no model fully replicated human rating behaviour. Overall, the alignment metrics appear stable: standard errors are small (0.019 for Disagreement, 0.018 for Spearman correlation), means match the original estimates to four decimal places, and biases are negligible (0.003), indicating that results are not driven by a few items.

For Disagreement, or Mean Absolute Distance metric, Benjamini–Hochberg (BH)-adjusted paired bootstrap tests (two-sided) indicated that the vast majority of model pairings were significantly differentiated at the $p < 0.01$ level across the overwhelming majority of dimensions, with the small number of pairwise comparisons failing to reach significance (Claude versus GPT-5-mini in readability ($p = 0.850$) and Qwen3-Coder-30B versus Llama-3-70B in modularity ($p = 0.376$)). Moreover, pairwise differences between best- and worst-performing models were large in magnitude. For example, the overall difference between Qwen3-Coder-30B and prometheus-13b-v1.0 was 0.675 [95% CI: 0.723, 0.629].

For Jensen-Shannon divergence, distributional differences were also almost universally significant after BH correction (adjusted $p = 0.002$ in nearly all comparisons), making this metric the most statistically decisive of the three metrics in separating model performance, with only isolated exceptions such as the DeepSeek-GRM-16B versus starcoder2-15b comparison in readability ($p = 0.108$) and complexity ($p = 0.661$).

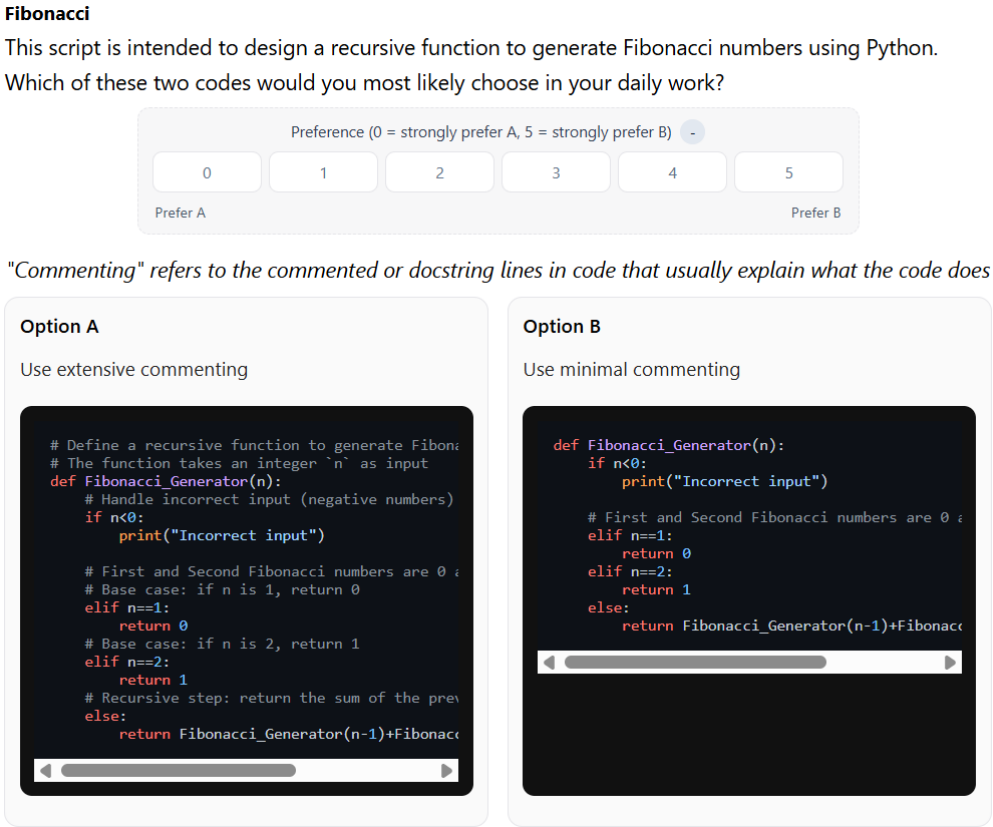

Figure 11: A full example of the expert preference annotation interface.

For Spearman correlation, Claude leads (= 0.447, 95% CI [0.418, 0.476], SE = 0.018), followed by QwenCoder (= 0.416, 95% CI [0.384, 0.449], SE = 0.016); their difference is borderline significant (difference = 0.031, 95% CI [0.062, 0.000], p = 0.050), with Claude's clearest advantage in readability (= 0.316 vs. 0.022, difference = 0.294, 95% CI [0.429, 0.162], p = 0.010). For a fuller picture regarding the confidence intervals, biases, standard errors, and significance scores, refer to Figures 12 to 17.

K Examples for Qualitative analysis

ID 1. Text_A: More computationally complex code

Text_B: Less computationally complex code

Annotator_Score_1: 1

Annotator_Score_2: 3

Annotator_Score_3: 1

GPT5_Score: 0

Ann_Comments_1:

Ann_Comments_2: I would choose Option B due to its greater efficiency and lower time complexity. While this function wouldn't most likely be used many times, by adding a large input on Option A the code would run recursively too many times, wasting time and resources.

Ann_Comments_3:

GPT5_Comments: I picked 0 (strongly prefer A) because Code A is minimal, self-contained, and directly returns the gcd with very little implementation overhead, making it easy to read, verify, and use for typical small-to-moderate integers. Although the subtraction form can be asymptotically worse than the division-based/extended variant, Code B does extra work to produce Bezout coefficients and has more branching/sign handling (higher constant overhead), so unless you specifically need the coefficients or must handle very large numbers, the simpler implementation is preferable in the time/resources tradeoff for common use.

Problem: This Python code solves the algo-

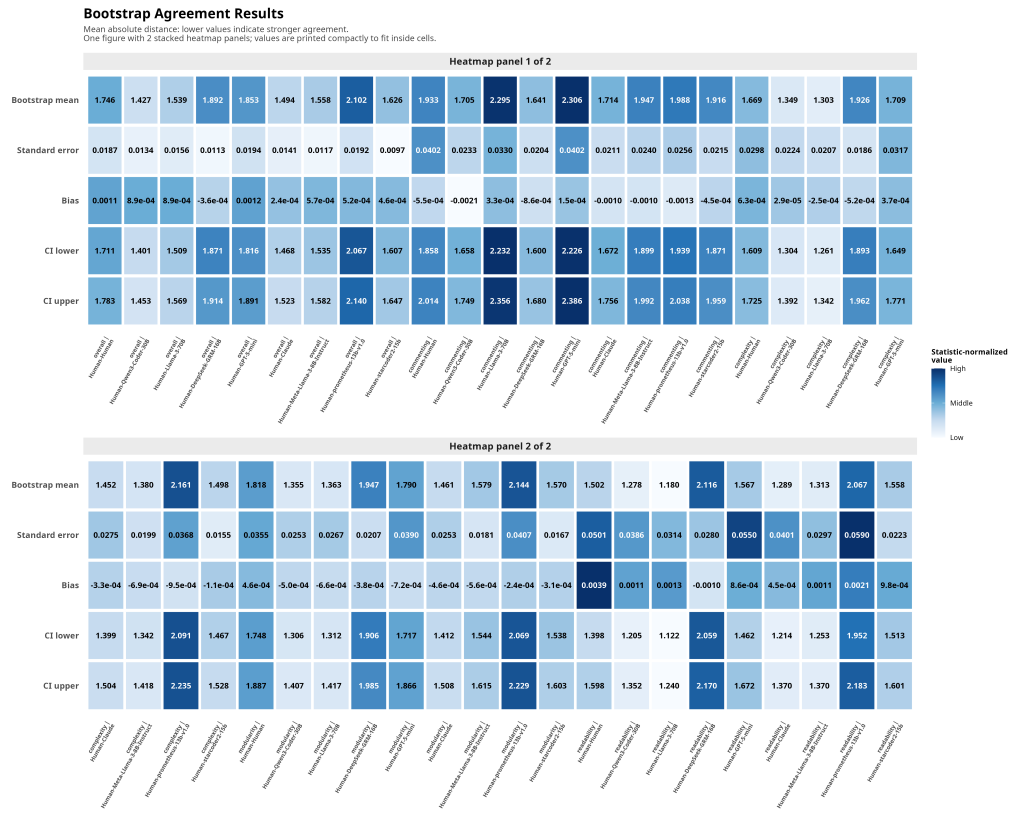

Figure 12: Human-LLM Bootstrapping Statistics for Disagreement metric.

rithmic problem of GCD that involves computing the greatest common divisor of two integers using Euclid's algorithm and returning the result.

ID 2. Text_A: More computationally complex code

Text_B: Less computationally complex code

Annotator_Score_1: 2

Annotator_Score_2: 3

Annotator_Score_3: 5

GPT5_Score: 5

Ann_Comments_1: Even if more complex, the comments are much more precise. If the algorithm works, and especially is a known algorithm, then its okay to use it anywhere, as long as it is commented properly like in option B.

Ann_Comments_2: Option B shows the characteristics of dynamic programming: it uses overlapping subproblems and optimal substructures. It stores the dynamic programming parts in the dp array. It ends with a reconstructing step in the while loop where the string is again put together.

On the other hand, Option A does not reconstruct the string, nor is it dynamic programming. It is, as the first comment says, just a naive recursive implementation without any memoization. This makes it heavily computationally intense (exponentially).

However, option B is not perfect since it lacks online commenting that helps to explain the steps (overlapping subproblems, optimal substructures, reconstruction).

Ann_Comments_3:

GPT5_Comments: I picked 5 (strongly prefer B) because the dynamic-programming implementation runs in O(mn) time and is practical for real inputs, whereas the naive recursive version is exponential and infeasible except for tiny strings. Code B also reconstructs the subsequence and handles edge cases cleanly (at the cost of O(mn) memory), making it far superior in the time/resources

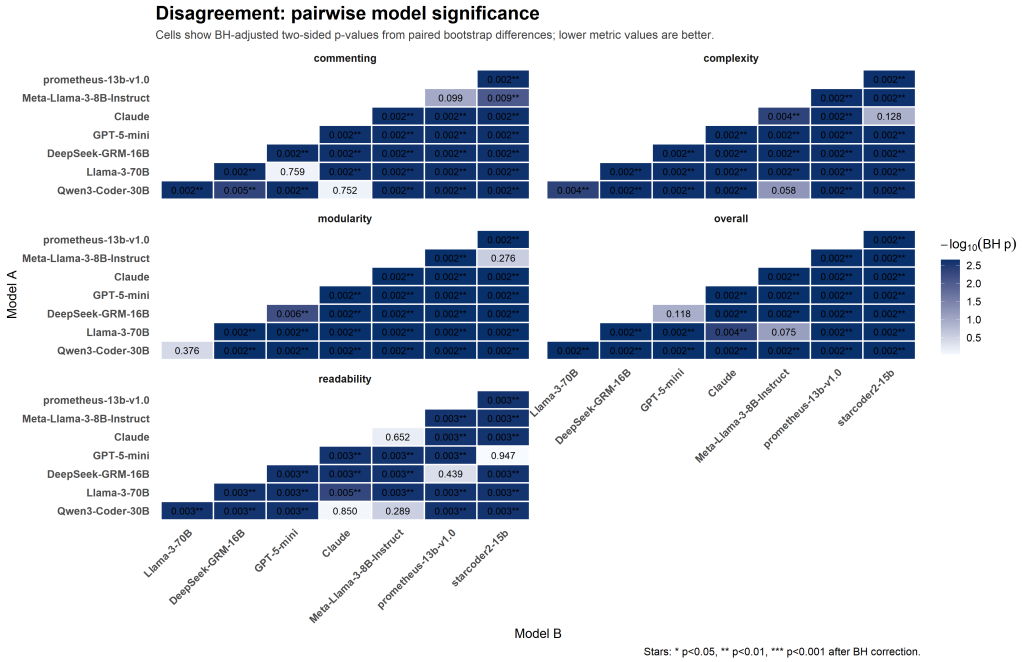

Figure 13: Pairwise models' significance scores for Disagreement metric.

tradeoff.

Problem: This Python code solves the algorithmic problem of LCS that involves computing the longest common subsequence of two strings with dynamic programming and reconstructing the subsequence.

ID 3. Text_A: More computationally complex code

Text_B: Less computationally complex code

Annotator_Score_1: 0

Annotator_Score_2: 2

Annotator_Score_3: 2

GPT5_Score: 0

Ann_Comments_1:

Ann_Comments_2: I would choose Option B because it implements Euclid's algorithm in a much simpler and more readable recursive form while maintaining optimal time complexity. Option A is more verbose and harder to maintain, even though it computes additional coefficients (extended GCD). In daily work, clarity and simplicity are usually preferable when only the GCD value is required.

Ann_Comments_3:

GPT5_Comments: I picked 0 (strongly prefer A) because Code A implements the plain Euclidean gcd with a single modulo recursion and minimal overhead, giving the fastest, simplest solution for just the GCD. Code B is an extended-GCD routine that maintains many extra variables and performs additional arithmetic to compute Bezout coefficients, so it does more work (and is more error-prone and harder to read) when only the gcd is required.

Problem: This Python code solves the algorithmic problem of GCD that involves computing the greatest common divisor of two integers using Euclid's algorithm and returning the result.

ID 4. Text_A: More computationally complex code

Text_B: Less computationally complex code

Annotator_Score_1: 3

Annotator_Score_2: 2

Annotator_Score_3: 1

GPT5_Score: 0

Ann_Comments_1: I would slightly prefer Option B. The dynamic programming approach efficiently computes the maximum

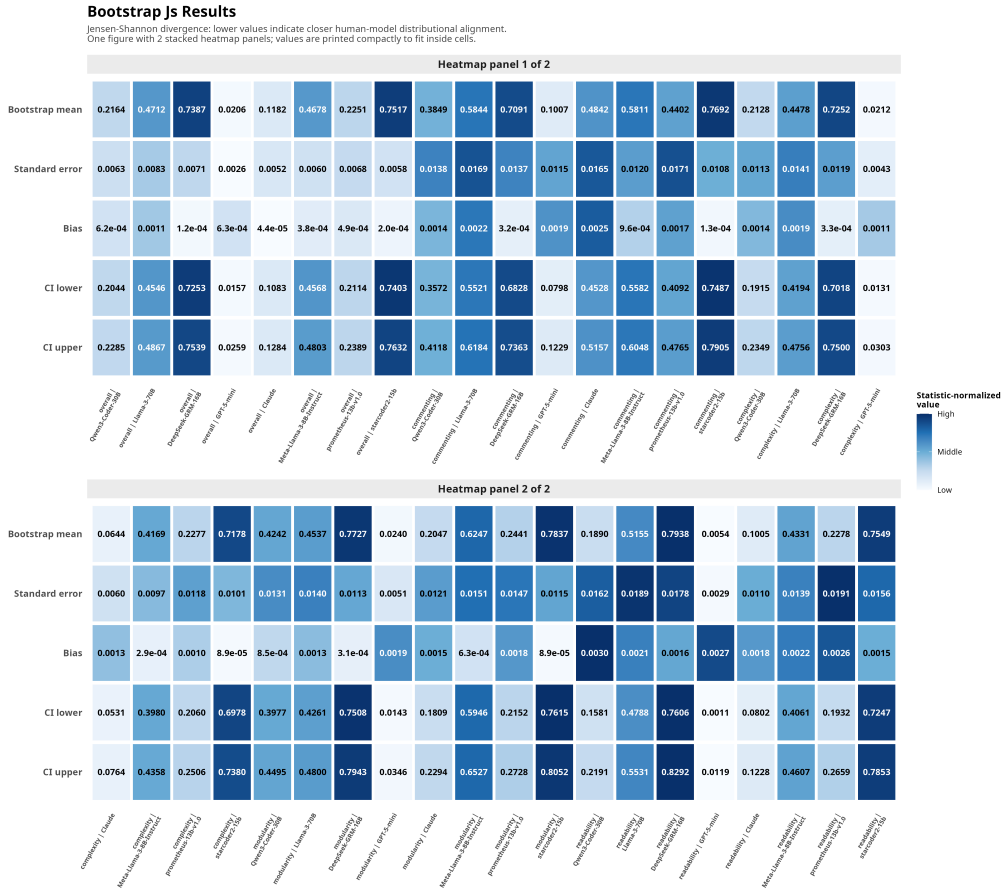

Figure 14: Human-LLM Bootstrapping Statistics for Jensen-Shannon divergence.

value without generating all possible combinations, which makes it much faster for larger numbers of items. While Option A is conceptually simple, it quickly becomes impractical as the number of items grows, so Option B is more suitable for daily work where performance matters.

Ann_Comments_2:

Ann_Comments_3:

GPT5_Comments: I chose 0 (strongly prefer A) because Code A directly returns both the optimal value and the explicit item selection (best_combination), making it a complete, easy-to-verify solution, whereas the provided DP (Code B) only returns the maximum value and omits reconstruction of which items achieve it. Also, although brute-force is exponential, it is simple, uses built-in combinatorics (good for small n), and avoids the potentially large O(n·capacity) time/space cost and implementation pitfalls of the DP when capacity is huge or item reconstruction is required.

Problem: This Python code solves the algorithmic problem of Knapsack that involves selecting items to maximize value under a weight/capacity constraint using dynamic programming and returning both value and chosen items.

ID 5. Text_A: More computationally complex code

Text_B: Less computationally complex code

Annotator_Score_1: 1

Annotator_Score_2: 3

Annotator_Score_3: 1

GPT5_Score: 0

Ann_Comments_1:

Ann_Comments_2: I would slightly prefer Option A. It uses a recursive merge sort approach, which has a predictable O(n log n)

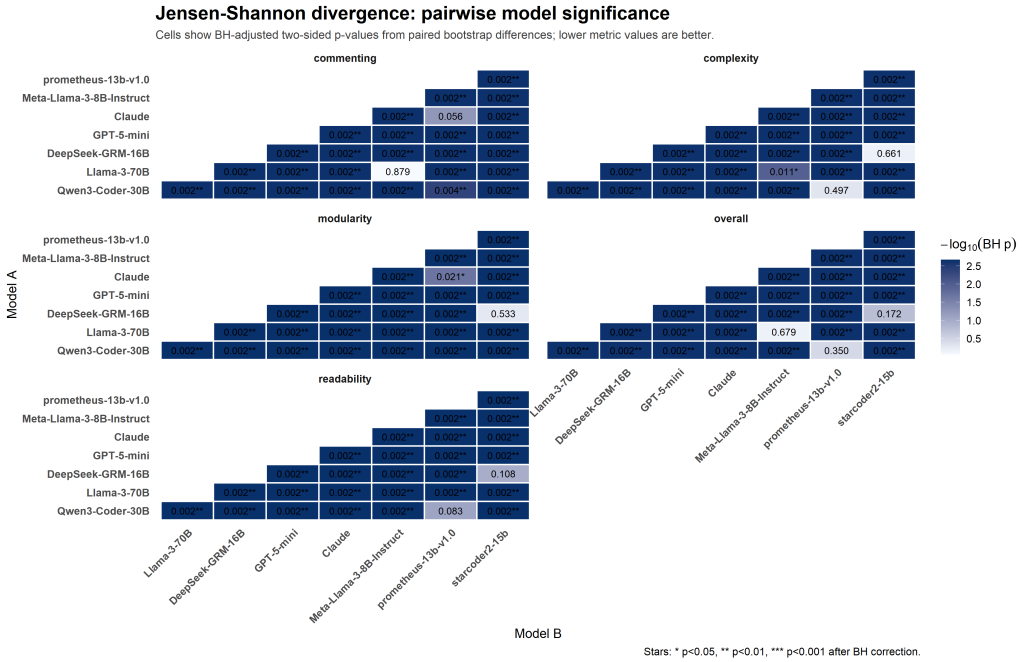

Figure 15: Pairwise models' significance scores for Jensen-Shannon divergence.

time complexity, making it more efficient for larger lists compared to bubble sort. Although the recursion and merging logic is a bit more complex, it's generally more suitable for handling bigger datasets in daily work.

Ann_Comments_3:

GPT5_Comments: I picked 0 (strongly prefer A) because this bubble_sort is simple, in-place, and has low per-operation overhead with an early-exit optimization (best case O(n)), making it predictable and lightweight for typical small-to-moderate inputs. The provided merge_sort implementation is actually inefficient in practice — it repeatedly slices lists, uses pop(0) (O(n) per pop), and concatenates lists, which induces extra copies and can degrade to much worse than the ideal O(n log n) behavior and use more memory, so its real-world time/space cost can exceed the bubble sort shown.

Problem: This Python code solves the algorithmic problem of Sorting Algorithm by ordering elements according to a key and stability rules, discussing time/space complexity and worst-case behavior.

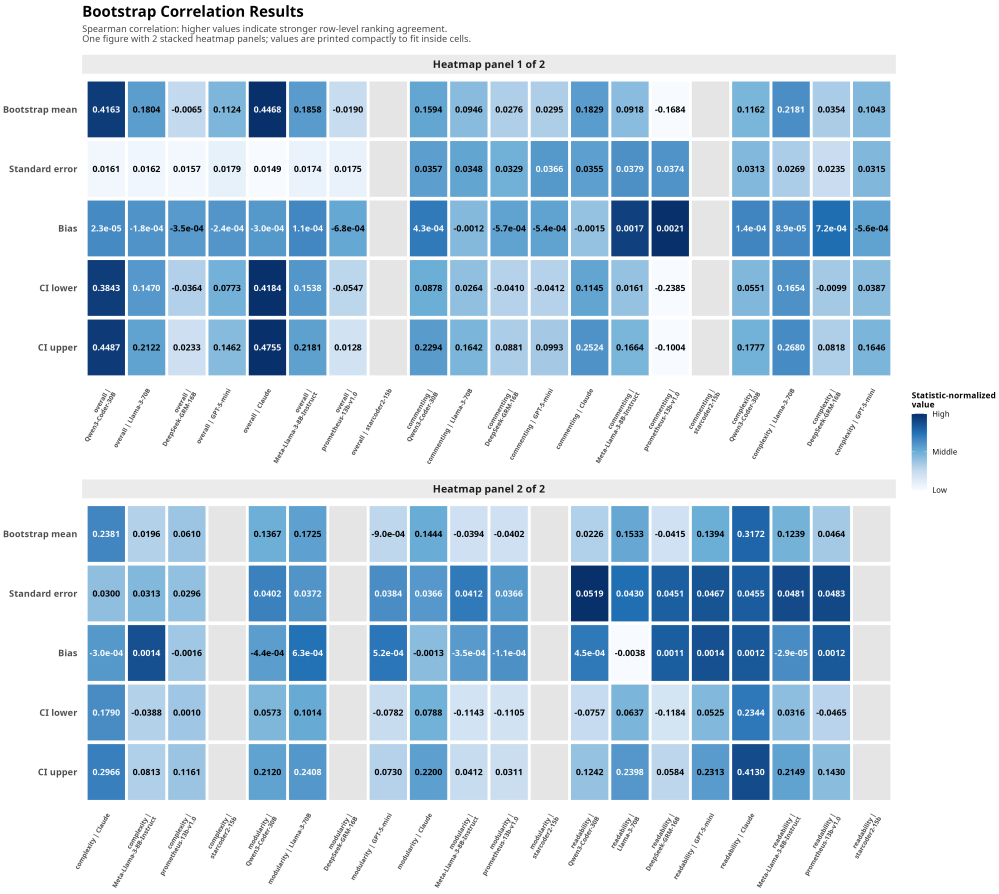

Figure 16: Human-LLM Bootstrapping Statistics for Spearman correlation.

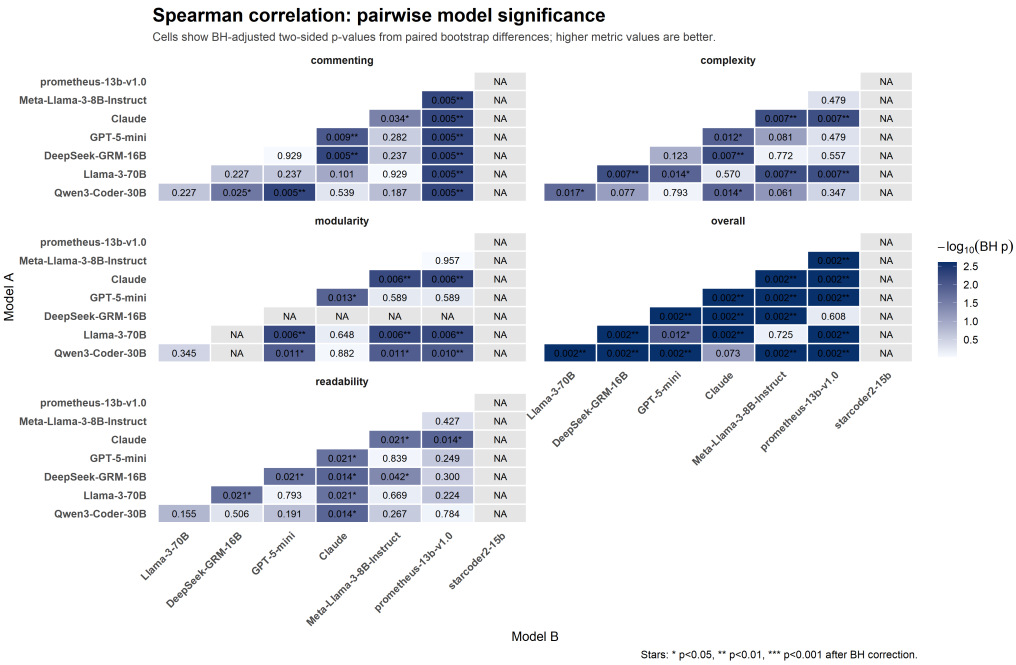

Figure 17: Pairwise models' significance scores for Spearman correlation.