CRQBench: A Benchmark of Code Reasoning Questions

Anonymous EMNLP submission

Abstract

Large Language Models have demonstrated exceptional proficiency on coding tasks, but it is challenging to precisely evaluate their code reasoning ability. Existing benchmarks are insufficient as they are unrealistic and conflate semantic reasoning ability with performance on software engineering tasks. We introduce CRQBench, a benchmark of 100 C++ code reasoning questions and answers derived from contextualized code review comments. To curate CRQBench, we use an LLM assistant alongside human inspection, reducing manual effort. We conduct an evaluation of GPT-4 on CRQBench and find that it produces correct responses grounded in the given context for 65 of the 100 questions.

1 Introduction

Large Language Models (LLMs) have demonstrated effectiveness in coding tasks and appear to understand deep semantic properties of code (Chen et al., 2021; Chowdhery et al., 2022; Touvron et al., 2023). However, evaluations across various tasks (Jimenez et al., 2023; Zhong and Wang, 2023) show less promising results, suggesting that models may have a limited syntactic understanding of programs. To evaluate a model's semantic reasoning ability in isolation, a benchmark specifically tailored for code reasoning question answering is needed.

The predominant benchmarks for evaluating LLMs trained on code are HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). They measure a model's ability to synthesize programs from docstrings. These text-to-code benchmarks are synthetic, handwritten, and involve generating a standalone function. Other, more realistic, benchmarks (Jimenez et al., 2023; Zhong and Wang, 2023) are designed to evaluate code reasoning indirectly through a software engineering task, and as a result conflate the model's ability to perform reasoning with the ability to perform the downstream task. In this work, we set out to curate a real-world, contextualized, benchmark for evaluating semantic reasoning ability in isolation.

Ideally, a benchmark for evaluating semantic reasoning ability should reflect real-world programming scenarios. Code review comments present an appealing target for this as they are non-synthetic and tied to a surrounding code context. Through a study of contextualized code review comments at a Corporation[1], we find that a subset embody semantically deep questions about code, but a majority are superficial (related to refactoring or style). Furthermore, we find that comments are rarely concise and unambiguous questions. Although code review comments provide a source of authentic semantic queries, it is non-trivial to extract clean questions.

We present CRQBench: a benchmark of real-world, contextualized, code reasoning questions. To reduce human curator effort, we propose a cooperative LLM and human-in-the-loop approach which leverages in-context learning (Brown et al., 2020) to filter and rephrase code reasoning questions from code review comments. We reproduce our Corporate results for open source release using Github pull request comments in the CodeReviewer dataset (Li et al., 2022).

In summary, our work presents a benchmark of 100 C++ (code reasoning questions, answer, code context) tuples derived from pull request comments in the CodeReviewer dataset. In addition, we present our curation technique as

[1] Anonymized for double-blind review

a re-usable methodology and evaluate its effectiveness in reducing manual effort in benchmark curation. Lastly, we evaluate GPT-4 (OpenAI et al., 2024) on CRQBench and find that it produces correct responses grounded in the given context for 65 of the 100 questions.

2 Motivating Examples

In this section, we illustrate the presence of code reasoning questions in code review comments, while highlighting the challenges in extracting them. Reviewers' identities are anonymized.

Observation 1: Most code review comments are not related to code reasoning. Through a manual analysis, we find that a majority of Github (65%) and Corporate (80%) code review comments are not related to code reasoning. We consider a comment to be related to code reasoning if in order to ask, answer, or address, it requires reasoning over reachability, data flow, control flow, or program variable and state. Instead, code review comments are often shallow edit suggestions related to style, structure, documentation, or syntactic reasoning. Consider Figures 1 (and 6 in appendix), in which the reviewers make shallow comments regarding style and *syntactic* reasoning respectively. During our analysis, we also found comments that are discussions of the intended behavior or specification (Figure 7 in appendix). We quantify the density of these comment categories in Table 1.

	Corp.	Github
CRQ	20%	35%
Shallow Edit Suggestion	60%	35%
Func Behavior Discussion	20%	30%

Table 1: Code Review Comments By Type.

Observation 2: Code review comments are often not phrased as questions. Through our manual analysis, we find that even when the comment is related to code reasoning, it is very rarely phrased as a concise and unambiguous code reasoning question. Consider Figure 2, in which the comment is phrased as an edit suggestion (removing the call to `std::move`) rather than the underlying code reasoning question: *Does calling `std::move` on the return value `s.releasePeerSet()` impact the program's behavior?* Furthermore, the comment contains extraneous information, referencing another reviewer. In Figure 3, the comment is posed as a question, but it is overly verbose. It consists of two sentences, one of which is an extraneous edit suggestion related to functional behavior. The first sentence, although related to code reasoning, is ambiguous and not contextualized in the reviewed code. It does not explicitly state which program variables *"something else"* encompasses. A concise, unambiguous rephrasing could be: *Can `error_code` hold a value other than `ECONNREFUSED` or `ECONNRESET`?*

We also observe that code reasoning questions can be categorized into two types of queries that encompass all CRQs: `VALUE` and `EQUIV` queries. A `VALUE` query (Figure 3) asks about the value or possible value of a variable or expression at a program point. An `EQUIV` (equivalence) query (Figure 2) asks if two segments of code have differences in behavior. `EQUIV` queries typically underlay an edit suggestion. We find that in both Github and Corporate code review comments, 75% of code reasoning questions are `EQUIV` queries while 25% are `VALUE` queries.

Observation 3: Answers to rephrased questions are not readily available. During our manual analysis, we inspected the developer's responses to comments. Responses came in the form of a natural language reply and/or a code edit. Answers in the form of a developer reply suffer from the same ambiguities and verbosity as the reviewer comments. Answers in the form of an edit require careful manual inspection to connect the change to the underlying code reasoning question. Sometimes the comment is ignored and not addressed.

3 Technique

Figure 4 illustrates our overall technique, which leverages a Corporate code aware LLM in combination with human validation.

3.1 Classifying Comments

As discussed in **Observation 1**, a minority of code review comments are related to code reasoning. To reduce manual inspection, we create an LLM based Code Reasoning Classifier (Figure 12 in appendix) which takes the

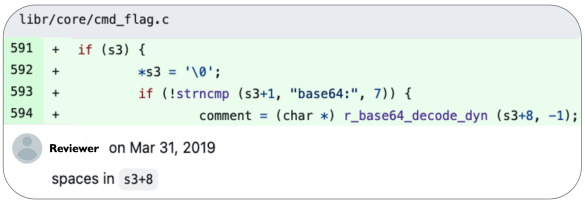

Figure 1: Shallow Edit Suggestion.[2]

Figure 2: Raw Code Review Comment.[3]

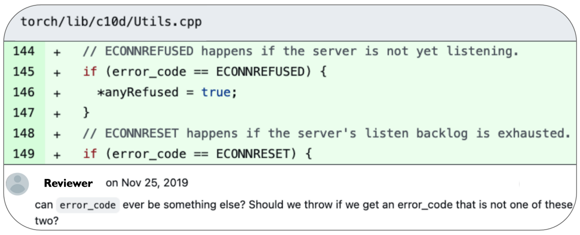

Figure 3: Raw Code Review Comment.[4]

raw reviewer comment and corresponding line of code and decides if it is related to code reasoning.

We evaluate the performance of our Code Reasoning Classifier prompt on 100 randomly selected, manually labeled comments as shown in Table 3. We also experiment with a keyword matching approach using a hand derived list of undesirable keywords (Figure 20 in appendix) [5], but find it incurs significantly more false positives than our LLM classification. In summary, our classifier correctly identified 11 out of 20 Corporate and 22 out of 35 GitHub code review comments as related to code reasoning, while misidentifying only 6 and 9 comments respectively.

	Corp		Github	
	LLM	KW	LLM	KW
Precision	.64	.31	.71	.39
Recall	.52	.81	.63	.1
F1 Score	.57	.45	.67	.56

Table 2: Code Reasoning Classification performance of LLM and Keyword matching approaches.

3.2 Rephrasing Comments as CRQs

As discussed in **Observation 2**, comments are rarely phrased as concise questions

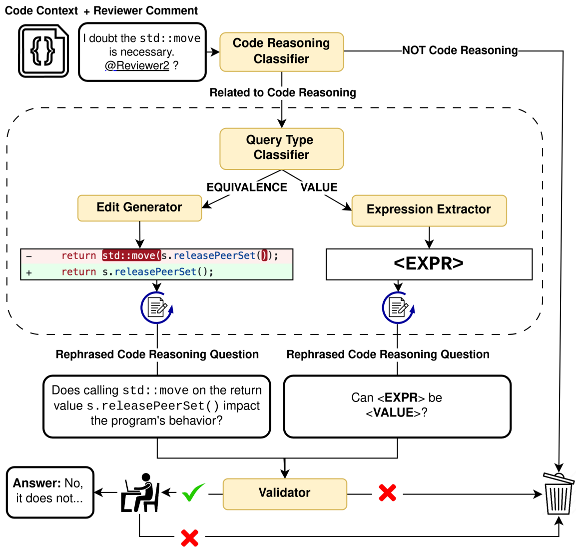

Figure 4: Benchmark Curation Methodology. Yellow boxes represent LLM prompts.

grounded over program elements. To avoid manual rephrasing, we again leverage the Corporate LLM. This portion of our technique is shown in the dotted box in Figure 4 and is invoked on samples that have been accepted by the Code Reasoning Classifier. Our technique invokes different rephrasing techniques for each query type. The Query Type Classifier (Figure 13 in appendix) classifies a comment as either an `EQUIV` query or `VALUE` query, triggering the appropriate rephrasing technique based on the classification.

When the Query Type Classifier decides the comment is an `EQUIV` query, we use chain of thought (Wei et al., 2022) reasoning to effectively rephrase. Since `EQUIV` queries are typically underlying edit suggestions, we employ an LLM based Edit Generator (Figure 14 in appendix) to perform the reviewer's suggested edit as a link in a chain of thought. The edit is leveraged to rephrase the reviewer comment using a few shot prompt (Figure 16 in appendix).

When the Query Type Classifier decides the comment is a `VALUE` query, we similarly use a two step inference process similar to a chain of thought. As a first step, our Expression Extractor uses a few-shot prompt (Figure 15 in appendix) leveraging the code context and reviewer comment to extract the relevant program expression: `<EXPR>`. The relevant `<EXPR>` is used as a link to rephrase the reviewer comment as a code reasoning question over the given expression using a few-shot prompt (Figure 17 in appendix).

Lastly, the rephrased question is given to the Validator for a self-consistency (Wang et al., 2023) check to reduce the occurrence of poorly rephrased code reasoning questions that are not faithful to the original line comment. The Validator prompt (Figure 18 in appendix) asks the LLM to decide if, given the original code, the reviewer comment has the same meaning as the rephrased comment. If the LLM Validator confirms the consistency, the rephrased question is selected as a confident candidate, and given to a human inspector to validate.

We evaluate our technique's effectiveness in rephrasing code review comments into concise and unambiguous code reasoning questions. Our rephrasing approach (entire dotted box component in Figure 4) is evaluated on both Corporate (150 samples) and Github (160 samples) code review comments that were flagged as related to code reasoning by our Code Reasoning Classifier. The samples were manually inspected and labelled as correct if they were concise, unambiguous, and faithful to the original reviewer comment. We achieved a precision of 66 on Corporate code review comments and .63 on Github pull request comments.

3.3 Evaluation

We evaluate our methodology in terms of manual human curation required. In a purely manual approach, a human curator would need to inspect and classify 500 Corporate (or 285 Github) code review comments and manually rephrase 100 questions. Using our proposed methodology, a human curator would need to inspect only 150 Corporate (or 160 Github) code review comments without the need for any manual rewriting. Figure 21 (in appendix) and Figure 5, respectively, illustrate this comparison. The pencil indicates a manual rephrasing while the magnifying glass indicates inspection using our proposed technique. In summary, our cooperative LLM + human validation approach reduces the number of samples required to inspect by 1.8x on Github pull request comments and 3.3x on

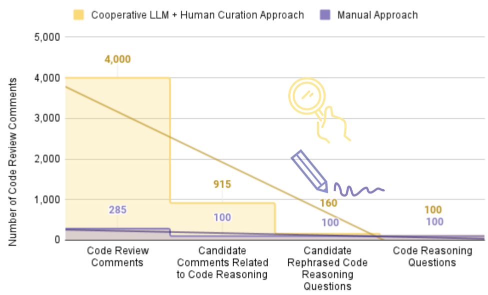

Figure 5: Manual vs Cooperative LLM Curation on Github Pull Request Comments.

Corporate code review comments.

4 GPT-4 Performance

We evaluate GPT-4 on our benchmark by prompting it with the surrounding function context (Figure 11 in Appendix). We evaluate the outputs manually considering a response to be accurate if it is both correct and contextually relevant. Correctness refers to the technical accuracy of the generated natural language response. Contextual relevance refers to the degree that the response is grounded in the given code context. We find the GPT-4 provides an accurate response on 65 of the 100 queries and are almost always (94%) grounded in the given code context.

	Acc	Total	%
	65	100	65%
`VALUE`	33	54	61%
`EQUIV`	32	46	70%

Table 3: Performance of GPT-4 on CRQBench.

Lastly, we conducted an error analysis to categorize the 35 incorrect responses. The majority of errors (25 instances) were due to the model lacking necessary context, such as usages of the given function, definitions of a used function or macro, or usages of a variable. Five errors were attributed to gaps in C++ knowledge, and the remaining five were due to incorrect evaluation of logic. Examples of each error scenario are shown in the Appendix (Figures 8 - 10).

We also experimented with evaluating the 7 billion parameter open source model Falcon (Almazrouei et al., 2023), but found it to have a much lower accuracy ($\tilde{2}5\%$) as it is a significantly smaller model.

5 Limitations

5.1 Extracting Answers to CRQs

To extract answers, we use an entirely manual based approach. A human curated an answer through a best effort approach by inspecting the cloned repository at the commit being reviewed. The answer is derived by reasoning over the code context, edit made (or not), and developer textual responses in the comment thread. In essence, our benchmark gathers the response which was implicitly provided by the developer, rather than an answer verified by a symbolic program analysis approach. We default to manual curation of answers due to the challenges presented in Section 2.

5.2 Size of Target Environment

Although the number of samples to inspect or rephrase is greatly diminished with our approach, the total number of comments needed to arrive at 100 code reasoning questions is much larger. Our cooperative approach requires greater than 10x more code review comments to derive 100 CRQs. This is due to false negatives in Code Reasoning Classifier and the Validator.

[2] https://github.com/radareorg/radare2/pull/13555#discussion_r270676564

[3] https://github.com/XRPLF/rippled/pull/1904#discussion_r88226072

[4] https://github.com/pytorch/pytorch/pull/30354#discussion_r350485189

[5] We also experimented with a hand derived desirable keyword list (Figure 19 in appendix). We report results for the undesirable keyword approach as it achieved a higher F1 score.

References

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The Falcon Series of Open Language Models. arXiv:2311.16867

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. arXiv preprint arXiv:2108.07732 (2021).

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. CoRR abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. CoRR abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:cs.CL/2204.02311

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:cs.CL/2310.06770

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. arXiv:cs.SE/2203.09095

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:cs.CL/2302.13971

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:cs.CL/2203.11171

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Advances in Neural Information Processing Systems, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

Li Zhong and Zilong Wang. 2023. Can ChatGPT replace StackOverflow? A Study on Robustness and Reliability of Large Language Model Code Generation. arXiv:cs.CL/2308.10335

A Appendix

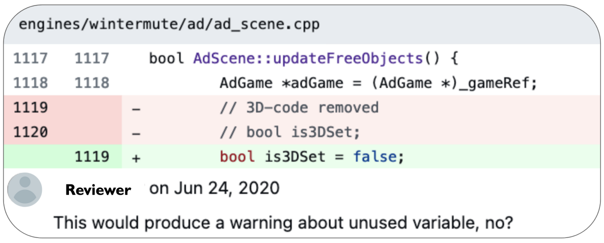

Figure 6: Shallow Edit Suggestion.[5]

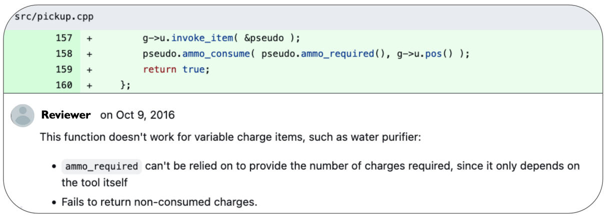

Figure 7: Functional Behavior Discussion.[6]

[5]https://github.com/scummvm/scummvm/pull/2347#discussion_r445103384

[6]https://github.com/CleverRaven/Cataclysm-DDA/pull/18691#discussion_r82533193

[7]The prompt used in our technique uses Corporate internal code review comments. To share the prompt, comments are substituted with open source code reviews from Github. We attempt to find substitutions which are similar to our Corporate code based prompt.

GPT-4

Prompt: In the following code, Is there a difference between using `!WI_IsFlagSet` and `WI_IsFlagClear`?

```
...
case WM_WINDOWPOSCHANGING:
{
    // GH#10274 - if the quake window gets moved to another monitor via aero
    // snap (win+shift+arrows), then re-adjust the size for the new monitor.
    if (IsQuakeWindow())
    {
        // Retrieve the suggested dimensions and make a rect and size.
        LPWINDOWPOS lpwpos = (LPWINDOWPOS)lparam;

        // We only need to apply restrictions if the position is changing.
        // The SWP_ flags are confusing to read. This is
        // "if we're not NOT moving the window"
        if (!WI_IsFlagSet(lpwpos->flags, SWP_NOMOVE))
        {
            // Figure out the suggested dimensions and position.
                RECT rcSuggested;
...
```

Response: Yes. `WI_IsFlagSet` and `WI_IsFlagClear` are (presumably, given they are not defined in the given code) utility methods for bit flag checking in the code.

`WI_IsFlagSet` checks whether a particular bit (or bits) is set in a value. If the bit is set, it means that the corresponding feature or property is enabled or true.

`WI_IsFlagClear`, on the other hand, checks whether a particular bit (or bits) is not set in a value. If the bit is not set, it means that the corresponding feature or property is disabled or false.

So the difference between them is that they check for opposite things:

- `WI_IsFlagSet` -> Checks if the bit(s) is/are set (1).
- `WI_IsFlagClear` -> Checks if the bit(s) is/are not set (0).

Which to use depends on what you are trying to check in your code.

Figure 8: Incorrect GPT Response: Logic Error. The model recognized that the flags check for opposite conditions but failed to deduce that the inverse of one flag is equivalent to the other.

GPT-4

Prompt: Can `uchar_from_utf8(&ch, &*iter, normalized_src.end() - iter)` be less than or equal to zero?

```
void imgtool::simple_charconverter::from_utf8(std::ostream &dest, const std::string &src) const
{
  // normalize the incoming unicode
  std::string normalized_src = normalize_unicode(src, m_norm);

  auto iter = normalized_src.begin();
  while(iter != normalized_src.end())
  {
    // get the next character
    char32_t ch;
    iter += uchar_from_utf8(&ch, &*iter, normalized_src.end() - iter);
...
```

Response: No. The function `uchar_from_utf8()` is not defined or included in the provided code context, so it's a bit speculative to determine its exact behavior. However, normally, such a function would return the number of bytes processed while converting a UTF-8 sequence into a unicode code point (character).

By convention, functions like these generally return a non-negative integer indicating the number of bytes processed. However, it's not impossible for such a function to return a negative value or zero in case of error conditions like an invalid UTF-8 sequence or encountering the end of the input prematurely.

Therefore, whether `uchar_from_utf8(&ch, &*iter, normalized_src.end() - iter)` can be less than or equal to zero mostly depends on the implementation details of `uchar_from_utf8()`.

In general, good coding practice would suggest handling such possible error conditions. However, assuming well-formed UTF-8 data and no I/O errors, it would typically always return a positive number.

Without the code or documentation of `uchar_from_utf8()`, it's challenging to give a definitive answer.

Figure 9: Incorrect GPT Response: Missing Context. The model was unable to determine if the expression could be less than or equal to zero as it did not have access to the function definition.

GPT-4

Prompt: In the following code, Is there a difference between using `std::make_pair` with `emplace_back` as opposed to not using `std::make_pair`?

```
...
if (timeout.count() != 0) {
    // Compute the expiration time for this message based on the timeout
    auto expirationTime = computeRpcMessageExpiryTime(timeout);

    // Add the Future to the right vector in the timeoutMap_
    {
      std::unique_lock<std::mutex> lock(timeoutMapMutex_);
      auto& timeoutFuturesVector = timeoutMap_[expirationTime];
      timeoutFuturesVector.emplace_back(
          std::make_pair(futureResponseMessage, timeout));
    }
    timeoutThreadCV_.notify_one();
}
...
```

Response: Yes, there is a difference between using `std::make_pair` with `emplace_back` versus not using `std::make_pair`.

In the context of this code, `timeoutFuturesVector.emplace_back(std::make_pair(futureResponseMessage, timeout));` is adding a `std::pair` containing the `futureResponseMessage` and `timeout` to the `timeoutFuturesVector`.

`std::make_pair` is a utility function that constructs a `pair` object with its first element set to x and its second element set to y.

`emplace_back` is used to construct and insert elements directly into a vector without the need for an intermediate pair object.

Here are the two cases for clarity:

1. **With `std::make_pair`**:
In this case, `std::make_pair` will construct a pair object from the passed-in arguments and then `emplace_back` just inserts this pair object into the vector.

2. **Without `std::make_pair`: **
If you don't use `std::make_pair` and directly pass arguments to `emplace_back`, it will try to construct an object of the vector's element type from the arguments.

That being said, in this case, without `std::make_pair`, you may get a compile error if the vector's element type can't be constructed directly from the arguments you pass to `emplace_back`. So if your vector is of `std::pair` type (like in your case it seems to be), then using `std::make_pair` is the right way.

Figure 10: Incorrect GPT Response: Incorrect C++ knowledge. The model incorrectly responded Yes. In particular, the model was incorrect that without using `std::make_pair`, a compile error may occur.

```
Given the following code, {QUESTION}
{SURROUNDING_FUNCTION}
```

Figure 11: Prompt to GPT-4 for CRQBench evaluation.

```
You are a senior, expert C/C++ programmer with a lot of experience analyzing C/C
    ++ code.
Given a C/C++ line comment:

{COMMENT}

associated with this line of code:

```
{CODE_SEGMENT}
```

please determine which of the following categories best classifies that
line comment:

1. unknown: unknown
2. reasoning: a code reasoning question such as requiring control-flow or value
    propagation
3. explanation: a question asking what the code does
4. structure: request to reorganize or refactor code, such as extracting classes
     or methods
5. planning: prioritization, planning, or choosing what to work on next
6. style: a code style, or readability question including things like renaming
    symbols
7. format: a comment or question about code formatting

and provide an explanation of why that comment should be in that specific
    category.
If you are uncertain about the category, respond with "0".

Generate your output in YAML format like this:

Explanation: <why you chose a specific category>
Line comment category: <category>

Response:
```yaml
```

Figure 12: Code Reasoning Classifier Prompt.
```

```
You are an expert software engineer, asked to determine whether or not a
    reviewer comment is about substituting code.

Here are some examples.

Example 1:

LINE_COMMENT:
‘‘‘
also check if the tf example is empty?
‘‘‘

1. No.

Example 2:

LINE_COMMENT:
‘‘‘
This stores a pointer to a temporary object. This pointer becomes invalid right
    after this statement. I suggest you initialise it with a ‘nullptr‘ and check
     for that.
’’’

1. No.

Example 3:

LINE_COMMENT:
‘‘‘
Please use stolen(fduri) instead of hardcoding 5
’’’

1. Yes.

Example 4:

LINE_COMMENT:
‘‘‘
The copy of a single int (previous type of fg, bg) is fine here, however copying
     the vector is not good. Use a const reference instead.
’’’

1. Yes.

Example 5:

LINE_COMMENT:
‘‘‘
{COMMENT}
’’’

1.
```

Figure 13: `Query Type Classifier` Prompt.[7]

```
You are an expert C++ coder and know how to review C++ code at a Corporation,
    and how to
respond to reviewer comments.
In the code below, a reviewer has left a comment marked by LINE_COMMENT.

Code:
```
{CODE_SEGMENT_WITH_COMMENT}
```

The comment is asking to consider the following original code and modified code:
```

Figure 14: `Edit Generator` Prompt.

```
You are an expert C++ coder and know how to review C++ code at a Corporation,
    and how to respond to reviewer comments. In the code below, a reviewer has
    left a comment marked by LINE_COMMENT.

Please answer the following question.
1. Can you please list the relevant program expression?

Here are some examples.
Example 1:

Code:
```
int comfort = 0;
comfort_response_t comfort_response;
comfort_response.aid = &item("null");
[*] [LINE_COMMENT] This stores a pointer to a temporary object. This pointer
    becomes invalid right after this statement. I suggest you initialise it with
     a nullptr and check for that.
bool plantsleep = has_trait(trait_CHLOROMORPH);
bool fungaloid_cosplay = has_trait(trait_M_SKIN3);
bool websleep = has_trait(trait_WEB_WALKER);
```

LINE_COMMENT: This stores a pointer to a temporary object. This pointer becomes
    invalid right after this statement. I suggest you initialise it with a
    nullptr and check for that.

1. comfort_response.aid

Example 2:

Code:
```
Value *Arg = Call->getArgOperand(0);
Value *LoweredArg = getLoweredByValOperand(Arg, Builder);
for (Value *A : Call->arg_operands()) {
    DXASSERT(A == Arg, "oops");
[*] [LINE_COMMENT] how could A be different unless there's a fundamental problem
     with operand iteration or something?
}
HLModule::MarkPreciseAttributeOnValWithFunctionCall(LoweredArg, Builder, *
    m_pModule);
addToDeadInsts(Call);
'''

LINE_COMMENT: how could A be different unless there's a fundamental problem with
     operand iteration or something?

1. A

Example 3:

Code:
```
{CODE_SEGMENT_WITH_COMMENT}
'''

LINE_COMMENT: {COMMENT}
```

Figure 15: Expression Extractor Prompt.[7]

```
Can you please rephrase a reviewer LINE_COMMENT that suggests the following code
    change? The rephrased comment should include program elements. Here are a
   few examples.

Example 1:
Before:
```
void gyroDataAnalyse(const gyroDev_t *gyroDev, biquadFilter_t *notchFilterDyn) {
    static FAST_RAM float fftAcc[XYZ_AXIS_COUNT] = {0, 0, 0};
    static FAST_RAM uint32_t fftAccCount = 0;
```
After:
```
void gyroDataAnalyse(const gyroDev_t *gyroDev, biquadFilter_t *notchFilterDyn) {
    static FAST_RAM float fftAcc[XYZ_AXIS_COUNT];
    static FAST_RAM uint32_t fftAccCount = 0;
```
[*] [LINE_COMMENT] Zero-initialized by default as well.
Rephrased: Is there a difference between initializing fftAcc to {0, 0, 0} and
   not initializing fftAcc?

Example 2:
Before:
```
if (isIntegralType(iter.dtype(), false)) {
    AT_DISPATCH_INTEGRAL_TYPES(iter.dtype(), "fmod_cpu", [&]() {
        cpu_kernel(iter, [](scalar_t a, scalar_t b) -> scalar_t {
            return std::fmod(a, b);
        });
    });
 }
```
After:
```
 if (isIntegralType(iter.dtype(), false)) {
    AT_DISPATCH_INTEGRAL_TYPES(iter.dtype(), "fmod_cpu", [&]() {
        cpu_kernel(iter, [](scalar_t a, scalar_t b) -> scalar_t {
            return a % b;
        });
    });
 }
```
[*] [LINE_COMMENT] Should we use fmod or % here?
Rephrased: Is there a difference between computing the modulus of a and b using
   std::fmod vs using the binary operator %?

Example 3:
Before:
```
{BEFORE}
```
After:
```
{AFTER}
```
[*] [LINE_COMMENT] {COMMENT}
```

Figure 16: Equiv Rewriter Prompt.[7]

```
You are an expert C++ coder and know how to review C++ code at a Corporation,
    and how to respond to reviewer comments.
A reviewer has left a comment marked by LINE_COMMENT.

Can you please rephrase the LINE_COMMENT as a question over program expressions?
The Question should start with the word "Can".

Here are some examples.

Example 1:

Comment:
```
This stores a pointer to a temporary object. This pointer becomes invalid right
    after this statement.
I suggest you initialise it with a nullptr and check for that.
```

Program Expression: comfort_response.aid
Question: Is there a difference between initializing comfort_response.aid with &
    item( "null" ) as opposed to nullptr?

Example 2:

Comment:
```
how could A be different unless there's a fundamental problem with operand
    iteration or something?
```

Program Expression: A
Question: Can A ever be equal to Arg?

Example 3:

Comment:
```
{COMMENT}
```

Program Expression: {EXPR}
```

Figure 17: Value Rewriter Prompt.[7]

```
You are an expert C++ coder and know how to review C++ code at a Corporation,
    and how to respond to reviewer comments.
In the code below, a reviewer has left a comment marked by LINE_COMMENT.

Code:
```
{CODE}
```

Given the LINE_COMMENT, answer the following question with YES or NO:
Does ```{Q_C}``` have the same meaning as the LINE_COMMENT?
```

Figure 18: Validator Prompt.

```
Interesting Keywords:
"except"
"segfault"
" fault"
"precondition"
 "assumption"
"undefined behavior"
 " ub "
"null",
"reach"
"ever be true"
"ever be false"
"branch taken"
"branch not taken"
"deref"
"reference"
```

Figure 19: Desirable Keywords for Positive Matching.

```
Uninteresting Keywords:
"test"
\nit"
\follow up"
\log a higher level"
"log a lower level"
"logging"
\naming"
\readability"
\TODO"
\description"
\comment"
"typo"
"clang"
"style guide"
\period"
"restructure"
"restructuring"
"refactor"
"move"
"offline"
"space"
"spacing"
```

Figure 20: Undesirable Keywords for Negative Matching.

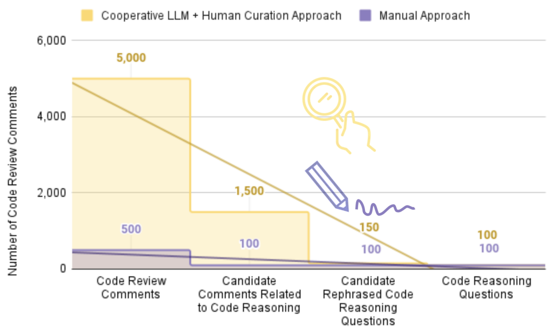

Figure 21: Manual vs Cooperative LLM Curation approaches on Corporate Code Review Comments.