GitChameleon - Explanation of Revisions for ACL ARR

Previous Submission ID: 682 (link: https://openreview.net/forum?id=EGq6tAkXc1)

We thank the Area Chair and the reviewers for their time and detailed feedback on our manuscript. While we believe some of the core criticisms stemmed from a misunderstanding of our benchmark's primary goal, the feedback was invaluable in helping us clarify our contribution. We have substantially revised the manuscript to address these points, and we believe the paper is significantly stronger as a result.

Overview of Major Revisions

The primary goal of this revision was to clarify that our benchmark, GitChameleon, evaluates **Version-Conditioned Generation (VCG)**—a model's ability to generate code for a specific, pinned library version—and to differentiate this from the task of **Code Evolution**. The major changes are as follows:

1. **Clarification of Core Contribution:** We have reframed the manuscript to explicitly define and discuss VCG. This includes adding a new **Figure 2** to visually contrast VCG (in-distribution) with Code Evolution (out-of-distribution) and using this terminology consistently throughout the text.
2. **Expanded Related Work:** To address concerns about missing literature, we have significantly expanded the **Extended Related Work** discussion in **Appendix D**. This appendix includes **Table 14**, which provides a point-by-point comparison of GitChameleon against other version-aware benchmarks, clearly positioning our work.
3. **Strengthened Experimental Rigor:**
    - We have updated our self-debugging experiments to remove any possibility of data leakage. As detailed in the revised **Appendix F**, the error trace provided to the model is now the top-level trace, and the unit test assertion is no longer exposed. All relevant results (**Table 1, Figures 7, 8, 10, 11**) have been updated accordingly.
    - To ensure the robustness of our findings, we have added **standard error bars** to our main result tables.
4. **Additional Experiment:**
    - We added a new experiment using the **Self-Explained Keywords (SEK) Prompting method (Table 10)** to provide a more comprehensive evaluation.
5. **Deeper Analysis:** We added **Appendix H** to analyze the logical complexity of our benchmark problems. This analysis, supported by **Figure 14**, demonstrates that our tasks primarily test version-specific knowledge retention rather than complex logical reasoning.
6. **Refined Limitations:** We have updated the **Limitations section (Section 5)** to more accurately reflect the scope of our work, clarifying that it is focused on VCG for Python and does not evaluate version-to-version translation or code evolution.

Point-by-Point Response to Reviewer Feedback

Response to Meta-Review (Area Chair yGom)

Weakness 1: "This benchmark is difficult to evaluate models' code evolution abilities, since it is not up-to-date."

- **Response:** We thank the Area Chair for this comment, which was central to our revision. We have clarified throughout the paper that our goal is not to evaluate "code evolution" but rather "Version-Conditioned Generation (VCG)". This distinction is now explicitly made in the introduction, defined in detail in its own subsection ("Code Evolution vs. Version Conditioned Generation"), and visually represented in the new **Figure 2**. We make it clear that testing on "not up-to-date" versions is an intentional design choice to evaluate a model's ability to handle legacy requirements.

Weakness 2: "The current version of this benchmark cannot avoid data contamination in the training data, since the sources are all public."

- **Response:** We directly address this concern in **Section 2.3**. We argue that this is not a weakness but a core feature of the benchmark's design. We state: "The challenge is therefore not one of data contamination, but of **control and disambiguation**: when a model has been exposed to multiple library versions, can it correctly generate code for the specific version required by the prompt?" This reframes the problem as a test of precision and control, which is the primary contribution of our work.

Response to Reviewer dqT7

Weakness 1: "Limited Benchmark Size."

- **Response:** We have clarified the difficulty and manual effort involved in constructing the benchmark in **Section 2: Benchmark**, noting the "roughly 350 human hours" required. This context helps justify the current size of the dataset while highlighting the quality and complexity of each hand-crafted problem.

Weakness 2: "Risk of test-set leakage or bias baked into the unit tests."

- Response: We have addressed this in two ways. First, we clarify the test construction process in Appendix A.4. Second, we argue that this is a non-issue, as rhe fundamental challenge in GitChameleon is not about passing a test with a particular structure, but about knowing and correctly applying a specific, historical API. If a model—even GPT-4.1—does not know the correct function name or argument signature for library v1.5, it will fail the execution-based test regardless of how "friendly" the test's implementation style is. The primary hurdle is the version-knowledge and control problem, which is much more difficult than any secondary stylistic bias in the test suite.

Weakness 3: "Benchmark hence largely measures memorisation of seen APIs rather than true generalisation to future changes."

- **Response:** This is correct, and it is the intended purpose of our benchmark. We have clarified this framing throughout the paper, particularly in **Section 1 (Introduction)** and **Section 2.3 (Statistics)**. We now clearly state that the benchmark is designed to test in-distribution knowledge retention and control, not out-of-distribution generalization to future changes. We further analyze this aspect in the new **Appendix H**, which shows that the problems are low on logical complexity and high on knowledge retention.

Response to Reviewer 4T5p

Weakness 1: "How to make sure the benchmark is up-to-date? ... make this benchmark not that practical."

- **Response:** As in our response to the meta-review, we have extensively revised the paper to clarify that the goal is not to be "up-to-date" but to test a model's ability to handle *any* specified version, which is a highly practical scenario in real-world development with pinned dependencies. This is addressed in **Section 1** and **Figure 2**.

Weakness 2: "How the version is controlled or required in the problems."

- **Response:** We have clarified this in **Section 3.1**, which now states: "To ensure version compliance, we use a dual control mechanism: the target version is explicitly included in the model's prompt, and the validation environment is configured with that exact library version". We had also updated **Figure 1** to include the library version in the prompt.

Weakness 3: "The reason of using greedy decoding instead of sampling is not clear."

- **Response:** While we maintain greedy decoding as our primary setting for reproducibility (a standard practice), we have added a new experiment using temperature sampling (**Table 9 in Appendix B**). The results show that performance is largely similar, supporting our use of a deterministic baseline while providing a more complete picture for the reviewer.

Response to Reviewer KtRy

Weakness 1: "Lacking critical related work."

- **Response:** We thank the reviewer for this crucial feedback. We have thoroughly addressed this by an extensive revision of **Appendix D**. Appendix D includes **Table 14**, which explicitly compares GitChameleon 2.0 to the benchmarks mentioned by the

reviewer (CodeUpdateEval, JavaVersionGenBench, RustEvo2, etc.) across key dimensions, clearly articulating our unique contribution.

Weakness 2: "Positioning dataset as version aware code generation + execution based eval."

- **Response:** Our revised **Appendix D** now clearly positions the work. We differentiate GitChameleon 2.0 by highlighting its unique focus on **instruction-based generation for static versions**, in contrast to other benchmarks that focus on code repair, updates, or completion.

Weakness 3: "Questions about Selection protocol and data leakage" / "RAG using API documentation."

- **Response:** We have clarified the RAG experimental design in **Section 3.1.4**. Furthermore, we address concerns about what the benchmark truly tests by adding **Appendix H**. This new section analyzes the logical complexity of the problems, showing that they are designed to test version-specific knowledge over complex reasoning, which is a form of "leakage" or knowledge recall that we are intentionally trying to measure.