Towards Autonomous Testing Agents via Conversational Large Language Models

Robert Feldt
Chalmers University of Technology
robert.feldt@chalmers.se

Sungmin Kang
KAIST
sungmin.kang@kaist.ac.kr

Juyeon Yoon
KAIST
juyeon.yoon@kaist.ac.kr

Shin Yoo
KAIST
shin.yoo@kaist.ac.kr

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00148

***Abstract*—Software testing is an important part of the development cycle, yet it requires specialized expertise and substantial developer effort to adequately test software. Recent discoveries of the capabilities of large language models (LLMs) suggest that they can be used as automated testing assistants, and thus provide helpful information and even drive the testing process. To highlight the potential of this technology, we present a taxonomy of LLM-based testing agents based on their level of autonomy, and describe how a greater level of autonomy can benefit developers in practice. An example use of LLMs as a testing assistant is provided to demonstrate how a conversational framework for testing can help developers. This also highlights how the often criticized "hallucination" of LLMs can be beneficial for testing. We identify other tangible benefits that LLM-driven testing agents can bestow, and also discuss potential limitations.**

***Index Terms*—software testing, machine learning, large language model, artificial intelligence, test automation**

I. Introduction

Software testing, an integral part of the development cycle, enables quality assurance and bug detection prior to deployment, for example via continuous integration practices [1]. However, automated software testing can be challenging, and necessitates a high level of technical acumen. There is significant expertise required to appropriately test software, as evidenced by the existence of test engineers/architects. Meanwhile, as Arcuri [2] notes, existing automated software test generation tools may also require significant expertise in the tool, in addition to being difficult to apply in industry.Furthermore, software testing and the writing of software tests can be repetitive, as Hass et al. [3] note. A more positive attribute of test cases is that their syntax is often significantly simpler than production software [4].

These unique characteristics of test code naturally bring about a distinction between testing experts and domain experts, which existing literature on developer expertise [5] supports by identifying distinct types of expertise: "understanding the vision of the project" and "knowledge about tools". Under this framework, an ideal setup would be one in which a testing expert and a domain expert collaborate to write tests for a project. The domain expert may lay out the specifications of a project, while the testing expert may convert those specifications into concrete tests, based on the testing expert's experience. A great strength of this process is that as a result of such a dialogue, initially unexpected, yet nuanced issues with the specification may arise, which provide opportunities to clarify the desired behavior. Indeed, handling such unexpected behavior is one of the virtues of software testing [6], [7].

In this paper, we argue that Large Language Models (LLMs), which have been trained with a large quantity of code including software test data [8] may eventually be capable of providing such testing knowledge, and that humans may act as domain experts and specify or clarify to the LLM what the intended behavior is. Specifically, we argue that LLMs are sufficiently well-trained with software tests to 'fill in' lower-level details of the intention of the developer. They also exhibit some 'knowledge' about testing methodologies,and can adapt them to new situations [9]. Going further, LLMs appear sufficiently capable in dialogue to converse about the results with a prospective software tester so that they could engage in a 'Socratic' manner: that is, they could provide counterexamples to help the developer to think their specification through, and thus uncover unexpected issues with the desired behavior, in this process clarifying what would be ideal. Equipped with appropriate 'middleware' which provides tools that the LLM could interact with, our eventual vision is that we can grant the LLM 'autonomy', in which it would set up a plan and 'use' the tools at its disposal to achieve the high-level objective set by the developer while abstracting away lower-level details.

To illustrate this idea, we organize the paper as follows. In Section II, we present literature on LLMs and how they can be used to emulate cognitive models for human behavior, thus providing a way of implementing our vision of testing LLMs that interact with tools and have agency while interacting with humans. In Section III, we provide a taxonomy of LLM-based software testing systems based on whether the LLMs are used in an *interactive* way, and the degree of '*autonomy*', i.e. formulating and executing its own plans. In Section IV we present an example interaction with the GPT-4 model [10], demonstrating that even without significant autonomy, developers gain an opportunity to ponder fine-grained semantics of their code via dialogue. The benefits of (autonomous) conversational testing agents are given in Section V, and we argue that greater autonomy confers greater benefits. Potential limitations are given in Section VI, and conclude in Section VII.

II. Background

With the recent advancements in large language models, the possibility of having a personal agent capable of assisting with general tasks is greater than ever. One popular ongoing

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00148

project is AutoGPT [11], which aims to implement a fully autonomous system that works towards achieving user-defined goals. Beyond the basic capabilities of GPT-4 model, the proposed framework supports a high level of autonomy by giving access to external tools such as search engines, complementary neural network models, and file systems. Moreover, AutoGPT retains both short-term and long-term memory management to cope with complex tasks that exceed the context length limitation of currently available language models.

The operation of AutoGPT can be interpreted from the perspective of existing cognitive architecture frameworks. In fact, modelling the human mind has been a longstanding research interest, driven by both objectives of explaining human behaviors and devising artificial intelligent agents. Several influential architectures such as ACT-R [12], and SOAR [13] have been developed so far, and their core components contain associative memory structures linked with perception and actuation ("motor") [14]. This bears resemblance with AutoGPT's architecture: i.e., incorporating external tools to perceive new information (e.g., via search engine results) or perform an action (e.g., writing a Python script) may be viewed as building the perception and actuation modules into the architecture. On the other hand, LLMs can strengthen classical cognitive architectures by deriving plausible actions using the relevant memory and current state as prompting context.

Park et al. [15] provided an interesting case study, where a memory-enabled framework that embeds LLMs with a unique reflection strategy on stored memories is used to simulate multi-agent social interactions. By prompting a LLM, their agent architecture continuously induces higher-level interpretation on what the agent had perceived. This enables the agent to maintain long-term coherence of its own behaviour, and in that process, plausible emergent social behavior is simulated. The recent paper by Wang et al. [16] also shows an LLM-based architecture that can explore a 3D world, acquire diverse skills, and make novel discoveries without human guidance.

Such advances in developing cognitive architectures on top of LLMs also open up numerous possibilities for software testing automation, such as managing continuous testing history as memories and planning the general testing strategy and then trying to fulfill sub-goals of the plan. In addition, an autonomous testing agent could evolve the test suite on its own by allowing the architecture to execute existing testing tools and access the results. In the following section, we provide a series of steps towards implementing such a testing agent.

III. Vision - SocraTest

Based on existing research on LLMs, our vision is to build SocraTest, a framework for conversational testing agents that are potentially autonomous and supported by existing automated testing techniques via a dedicated *middleware*, that would invoke appropriate tools based on LLM output, so that LLMs can operate in an autonomous manner. We posit that such an agent can not only become an intelligent testing partner to a human software engineer, but also be able to handle typical testing related tasks autonomously.

A taxonomy of LLM use when performing software testing is presented in Table I, with higher rows representing higher degrees of autonomy from the LLM perspective. Specifically, the Driver column shows who drives the operation, i.e., who initiates the task, collects information, and decides the next step. For example, code completion provided by GitHub Copilot is automatically initiated by the front-end, i.e., the editor. Techniques based on contextual prompting, such as Libro [17], are still considered to be driven by the technique itself, in that a human is the starting point but not part of the workflow. Conversational testing, an example of which is shown in Section IV, involves a human in the interactive loop: the user drives the next action via the dialogue with the LLM.

We can also categorize LLM usages based on their information sources: more advanced use cases increasingly involve a wider range of information sources and more complicated interactions. In the most basic usage of LLMs, i.e. autocompletion and in-filling, the only information source is the code context, which is already written by the human user. In contrast, Contextual Prompting provides further contextual information, e.g. in the form of examples, and depends on the few-shot learning capabilities of LLMs to perform the given task. While this approach has successfully enabled much more complicated tasks such as bug reproduction [17], its format is still a single query-and-response, without any interaction between the developer and the tool.

We argue that a tool capable of dialogue, corresponding to Conversational Testing and upward in the taxonomy, can extend the scope of both the role of the driver and the information sources and enable unique benefits (as in Section IV). At the lowest level of autonomy (Conversational Testing), as a conversational partner, LLMs partially drive the process, but only respond to human requests without autonomy. One level up, we can introduce a low level of autonomy by providing codified instructions for the LLM to follow (Conversational Testing with Tools): for example, we can set structural testing as a goal and allow LLMs to initiate the use of appropriate tools, e.g. EvoSuite [18] and Jacoco [19], to generate tests and measure coverage. Finally, at the highest level of autonomy (corresponding to Conversational Testing Agents), LLMs are augmented with memory and planning capabilities so that humans only need to provide high-level directions, while LLMs initiate and complete whole tasks of a testing process.

To implement such autonomous testing agents using LLMs, a prerequisite is the implementation of middleware for conversational testing agents as a set of supporting features. Various existing testing tools and techniques should be included in the middleware so that they can be used by the LLM. The middleware can also augment LLMs with memory, similarly to experiments such as AutoGPT [11] or other autonomous cognitive models based on LLMs [15]. This middleware may use frameworks such as LangChain [20], which ease the connection between LLMs and external tools. In lieu of the fully realized vision, we present how even at a lower level of autonomy, i.e. at the conversational testing level, testing can become much easier from the developer's perspective.

TABLE I
TAXONOMY OF LLM USES IN SOFTWARE TESTING

Mode of Usage	Driver	Interactive	Available Information	Autonomy
Conversational Testing Agents	Human, Middleware, LLM	Yes	Extensive:, information from both user and the tools in middleware	High
Conversational Testing with Tools	Human, Middleware	Yes	High, additional outputs from algorithms & methods	Low
Conversational Testing	Human	Yes	Rich: a mixture of templates, contexts, examples, and explanations	No
Contextual Prompting	Front-end, Testing SW	No	Medium: templates with contexts & examples	No
Completion & Infilling	Front-end, Testing SW	No	Low: typically autocompletion of given code	No

IV. INSPIRATIONAL EXAMPLE TASKS

We have had a large number of software testing related conversational interactions with the GPT-4 model through the ChatGPT interface. We have found that the model can both describe different types of testing methods, merge and condense them to checklists to guide testers, as well as write executable test code to apply and exemplify the methods/checklists. We have also found the conversational mode essential both to clarify, as a developer or tester, the type of testing and support one needs and to request additional test code and the use of additional testing methods. For brevity, we here provide only a condensed example of a multi-step interaction we had with the model to do unit testing for the Julia language [21], with each user query marked with "Prompt N".

After describing the type of Julia code we developed we asked GPT-4 for concrete advice, methods and checklists for how we should write unit tests (Prompt 1). It provided a detailed and long checklist that gave general and broad advice. It was actionable but quite generic. We then asked it to focus on test input selection and to provide a more detailed method and checklist (Prompt 2). GPT-4 proposed that we should use "Equivalence Partitioning" and "Boundary Value Analysis" and went on to define them. It also proposed a checklist that combined the main steps of the two techniques. We then asked it to provide example Julia test code to test a function in Julia's Base library that takes 2-3 inputs (Prompt 3). The model selected the `Base.clamp(x, lo, hi)` function and briefly described it ("The clamp function restricts a value x to be within the range [lo, hi]. If x is less than lo, it returns lo. If x is greater than hi, it returns hi. Otherwise, it returns x."). It then provided Julia test code with 16 test cases, an excerpt of which is shown below. It grouped test cases in relation to its checklist and briefly documented each group to indicate the checklist item that "leads" to the group.

```
using Test

@testset "clamp function tests" begin
    # Valid input classes
    # Typical inputs
    @test clamp(5, 2, 7) == 5
    @test clamp(1, 0, 10) == 1
    @test clamp(6, 0, 10) == 6

    # Boundary cases
    @test clamp(0, 0, 10) == 0
    @test clamp(10, 0, 10) == 10
    @test clamp(5, 5, 5) == 5

    # Invalid input classes
    # lo > hi
    @test_throws ArgumentError clamp(5, 7, 3)
    <...9 test cases omitted for brevity>
end
```

Based on our limited experience of the `clamp` function we did not spot any problems with the test code; all test cases initially made sense. Upon execution, it turned out that 15 test cases pass but that an `ArgumentError` is **not** thrown by Julia when calling `clamp(5, 7, 3)`; Julia actually returns 3. We told the model about this (Prompt 4) by providing the output from running the test code as well as from making the call and showing the value returned by Julia. We asked it to explain why this happened. The model apologized and explained that Julia actually returns the `hi` value in cases where it is lower than the `lo` value. It went on to update the test code and corrected the `@test_throws ArgumentError ...` as shown in the following.

```
# Invalid input classes
# Julia return hi if lo > hi
@test clamp(5, 7, 3) == 3
```

We were not impressed by the fact that the model were now confidently explaining that the `clamp` function behaves in this way when it had earlier proposed this was not the case. However, the conversational mode of interaction was useful in nudging the model to give us more detailed and concrete information and in particular to provide relevant test code to exemplify its recommendations. It seems clear that this can have pedagogical and learning benefits as well as act as a reminder to apply important testing techniques in new contexts. The interactive, conversational mode also allowed us to further explain what we meant and requested and to ask the model to update and refine test code it had earlier provided.

We also argue that the "erroroneous" test code provided

for the `lo > hi` case shows that LLMs like GPT-4 can be particularly useful for testing. While the "error" the model did in the earlier step can be seen as a type of hallucination [22], we argue that for testing this is less severe (test code will not be part of the final, deployed software system) and can even be a benefit. In this case we argue that even a human tester could have assumed that the clamp function would first ensure that the lo value is less than or equal to the hi value and that an exception would be thrown otherwise. We actually learnt something about Julia through this mistake and we argue that a tester and developer could also have learnt something and even decided that raising an exception would be the more sensible thing to implement. In this sense, for software testing, the so called "hallucination" that LLMs have been criticized for can, at least sometimes, be a benefit, as it can prompt deeper thought. This is in line with the argument of Feldt et al. [23] that "process" use of AI in software development is less risky.

V. Progress towards vision

While even low-autonomy conversational testing can help the developer verify software, techniques with higher autonomy can confer even greater benefits. We identify that there are at least three benefits to conversational testing via LLMs, which are increasingly "unlocked" with a higher level of autonomy. To start, as mentioned earlier, while LLM hallucination has been identified as a problem [24], it can actually be an asset when doing software testing, as in general we want to be able to generate tests that uncover the unexpected behavior of software [6], [7]. This characteristic benefits all levels of LLM use for testing, as 'hallucination' can happen at any level of content generation while using LLMs.

At a greater level of autonomy (Conversational Testing with Tools or higher), we argue that one of the greatest benefits LLMs can bring about is the fact that they can codify and implement non-formalized testing scripts that are still manually processed [3], [25] based on their natural language processing capabilities. For example, we can imagine a conversational testing agent interpreting and executing natural language testing guidelines written for humans, executing tools and seeking clarifications if needed via the middleware. As such non-formalized testing techniques or guidelines are intended for humans, they could be readily adopted as part of already-existing testing practices, which can improve developer acceptance of results [26], while also acting as explanations for any generated results [27]. At the greatest level of autonomy, LLMs would formulate and execute testing plans, while conversing with the developer on abstract terms. For example, in our example from the previous section, a human had to copy-and-paste the generated tests from an LLM and manually execute the tests; with the appropriate middleware, this process could be automated, and the developer would only need make higher-level decisions. As a result, this level of autonomy has the potential to significantly reduce the developer effort that goes into testing software. It could also lead to better utilisation of computer resources by continuously trying to fulfill testing goals even when the human/developer is away.

VI. Present-day limitations

A major limitation to the attempt to use current generation LLMs for software testing in the way of SocraTest is that on their own, LLMs lack any agency to use external tools. However, specific prompting techniques such as REACT [28] or PAL [29] have shown that external tools can be indirectly woven into the dialogue, providing LLMs the information produced by external tools so that it can continue further inference using them. Also, systems like HuggingGPT [30] and AutoGPT [11] show that even if an LLM is not provided with tool access on a lower level it can be done via direct prompting and explanation.

A further limitation is that the planning abilities of current LLMs are not well-defined, often considered among their less developed competencies [31]. While this might be mitigated by multi-step prompting techniques as in recent work [32], other hybrid systems might need to be explored that combines LLMs with more traditional AI planning tools and algorithms.

The significant costs of LLM training and operation constitute an indirect limitation, with e.g. their few-shot learning ability associated with their size [33]. The ensuing model growth leads to significant energy requirements and limits access to resource-rich organizations, hence impeding development of open tools. Despite this, performance does not rely solely on size, but also on training data volume [34]. Furthermore, techniques like model quantization and low-rank adaptation have shown promise in creating smaller, yet effective models [35], [36], which due to their more permissive licenses can also mitigate some concerns about LLM use when dealing with confidential data.

VII. Conclusions

This paper provides an overview of conversational and potentially autonomous testing agents by first presenting a taxonomy of such agents, describing how these agents could help developers (and increasingly so when granted with greater autonomy). A concrete example of a conversation with an LLM is provided as initial confirmation that conversational testing can be used to enhance the testing effectiveness of developers. Finally, limitations of these techniques is provided, providing context for our vision. As described in the paper, appropriate middleware is critical for realizing the autonomous testing agents that we envision; we plan on investigating which software engineering tools could aid the efficacy of conversational testing, and how they can be integrated harmoniously with LLMs to aid software testing in practice.

Acknowledgment

Robert Feldt has been supported by the Swedish Scientific Council (No. 2020-05272, 'Automated boundary testing for QUality of AI/ML modelS') and by WASP ('Software Boundary Specification Mining (BoundMiner)'). Sungmin Kang, Juyeon Yoon, and Shin Yoo were supported by the Institute for Information & Communications Technology Promotion grant funded by the Korean government MSIT (No.2022-0-00995).

References

[1] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. DOI: 10.1109/ACCESS.2017.2685629.

[2] A. Arcuri, "An experience report on applying software testing academic results in industry: We need usable automated test generation," *Empirical Softw. Engg.*, vol. 23, no. 4, pp. 1959–1981, Aug. 2018, ISSN: 1382-3256.

[3] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 1281–1291.

[4] M. J. Rutherford and A. L. Wolf, "A case for test-code generation in model-driven systems," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, 2003, pp. 377–396.

[5] J. T. Liang, M. Arab, M. Ko, A. J. Ko, and T. D. LaToza, *A qualitative study on the implementation design decisions of developers*, 2023. arXiv: 2301.09789 [cs.SE].

[6] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST 2016, 2016, pp. 223–233.

[7] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[8] L. Gao, S. Biderman, S. Black, *et al.*, *The pile: An 800gb dataset of diverse text for language modeling*, 2020. arXiv: 2101.00027 [cs.CL].

[9] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," *arXiv preprint arXiv:2302.03287*, 2023.

[10] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[11] *AutoGPT: An Autonomous GPT-4 Experiment*, 2023. [Online]. Available: https://github.com/Significant-Gravitas/Auto-GPT.

[12] J. R. Anderson, *How can the human mind occur in the physical universe?* Oxford University Press, 2009.

[13] J. E. Laird, *The SOAR cognitive architecture*. MIT press, 2019.

[14] J. E. Laird, "An analysis and comparison of act-r and soar," *arXiv preprint arXiv:2201.09305*, 2022.

[15] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, *Generative agents: Interactive simulacra of human behavior*, 2023. arXiv: 2304.03442 [cs.HC].

[16] G. Wang, Y. Xie, Y. Jiang, *et al.*, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

[17] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," *arXiv preprint arXiv:2209.11515*, 2022.

[18] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, 2010, pp. 147–158. DOI: 10.1145/1831708.1831728.

[19] *JaCoCo Java Code Coverage Library*, 2013. [Online]. Available: https://github.com/jacoco/jacoco.

[20] H. Chase, *Langchain*, 2022. [Online]. Available: https://github.com/hwchase17/langchain.

[21] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[22] Z. Ji, N. Lee, R. Frieske, *et al.*, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[23] R. Feldt, F. G. de Oliveira Neto, and R. Torkar, "Ways of applying artificial intelligence in software engineering," in *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2018, pp. 35–41.

[24] Y. Bang, S. Cahyawijaya, N. Lee, *et al.*, *A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity*, 2023. arXiv: 2302.04023 [cs.CL].

[25] F. Dobslaw, F. G. de Oliveira Neto, and R. Feldt, "Boundary value exploration for software analysis," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2020, pp. 346–353.

[26] E. R. Winter, V. Nowack, D. Bowes, *et al.*, "Towards developer-centered automatic program repair: Findings from bloomberg," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1578–1588. DOI: 10.1145/3540250.3558953.

[27] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, *Explainable automated debugging via large language model-driven scientific debugging*, 2023. arXiv: 2304.02195 [cs.SE].

[28] S. Yao, J. Zhao, D. Yu, *et al.*, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[29] L. Gao, A. Madaan, S. Zhou, *et al.*, "Pal: Program-aided language models," *arXiv preprint arXiv:2211.10435*, 2022.

[30] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chat-

gpt and its friends in huggingface," *arXiv preprint arXiv:2303.17580*, 2023.

[31] S. Bubeck, V. Chandrasekaran, R. Eldan, *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[32] S. Yao, D. Yu, J. Zhao, *et al.*, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.

[33] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, *Scaling laws for neural language models*, 2020. arXiv: 2001.08361 [cs.LG].

[34] J. Hoffmann, S. Borgeaud, A. Mensch, *et al.*, *Training compute-optimal large language models*, 2022. arXiv: 2203.15556 [cs.CL].

[35] A. Polino, R. Pascanu, and D. Alistarh, *Model compression via distillation and quantization*, 2018. arXiv: 1802.05668 [cs.NE].

[36] E. J. Hu, Y. Shen, P. Wallis, *et al.*, *Lora: Low-rank adaptation of large language models*, 2021. arXiv: 2106.09685 [cs.CL].