Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study Preprint

Preprint.
Final version to appear in
AISoLA 2023, Lecture Notes in Computer Science vol. 14380,
Springer, 2023.

Lenz Belzner[1], Thomas Gabor[2], Martin Wirsing[2]

[1] TH Ingolstadt, Ingolstadt, lenz.belzner@thi.de
[2] Ludwig-Maximilians-Universität München, München,
{gabor,wirsing}@ifi.lmu.de

Abstract. Large language models such as OpenAI's GPT and Google's Bard offer new opportunities for supporting software engineering processes. Large language model assisted software engineering promises to support developers in a conversational way with expert knowledge over the whole software lifecycle. Current applications range from requirements extraction, ambiguity resolution, code and test case generation, code review and translation to verification and repair of software vulnerabilities. In this paper we present our position on the potential benefits and challenges associated with the adoption of language models in software engineering. In particular, we focus on the possible applications of large language models for requirements engineering, system design, code and test generation, code quality reviews, and software process management. We also give a short review of the state-of-the-art of large language model support for software construction and illustrate our position by a case study on the object-oriented development of a simple "search and rescue" scenario.

Keywords: large language model, GPT, Bard, LLM-assisted, software engineering, state-of-the-art, challenges, requirements, design, validation, verification.

1 Introduction

Software engineering (SE) has traditionally been a highly manual, human-driven process that involves a range of activities from requirements gathering and analysis, to system design, implementation, and testing. Recent advances in artificial

intelligence (AI) have led to the development of powerful natural language processing models such as OpenAI's GPT [36,6,34] or Google's Bard [42], known as large language models (LLMs). For an overview on current LLMs see [32,49].

LLMs have opened a wide array of applications in various domains, including applications in software engineering processes. In contrast to classical software development support, textual interaction with an LLM-based development environment is not command-based but intent-based [33] and conversational: a developer engages in a dialogue with the system in which she asks questions or describes the desired outcome, but does not state how that outcome is constructed or calculated.

Vision Our vision is that the use of LLMs will lead to a paradigm shift in software development. In "LLM-assisted software engineering" the LLM acts together with other supporting bots and tools to help the human developers in all phases of the software lifecycle. The LLM plays the role of development expert whereas developers act as domain experts. Humans specify and clarify to the LLM the intended requirements, judge and correct the software design proposals and the code produced by the LLM-bot and other bots the LLM cooperates with. In the validation and verification phase the LLM serves as testing and verification expert. It autonomously generates tests, invokes appropriate testing and verification goals and tools, and converses with the human for uncovering unexpected issues with requirements and design specifications. Moreover, an LLM may also play the role of a software process expert and plan the forthcoming development activities in agile development process.

However, to realize this vision many challenges still need to be overcome ranging from integrating LLMs into the larger domain and technical context as well as evaluating and testing the quality of the output generated by LLMs to many further issues in practical use. If all these challenges can be resolved, we see (at least) three scenarios of how our vision for integrating LLMs into software engineering might play out: (1) integration of today's LLMs into the standard tooling for routine software development, (2) integration into the standard software engineering processes so that LLMs take over some expert roles and thus replace some human experts, and (3) integration into each stage of software development so that the role of human software developers changes entirely to a manager of AI-induced software development processes.

Contribution This paper presents our position on the potential benefits and challenges of LLM-assisted software engineering, with a particular focus on requirements engineering, system design, code and test generation, and code quality reviews. We give a short overview of the current (July 2023) state-of-the-art of large language model support for software construction and illustrate our position by exemplary dialogues with ChatGPT and Bard on object-oriented development of a simple "search and rescue" scenario.

Our work provides an initial exploration into the application of LLMs throughout the full software development lifecycle, encompassing requirements engineering, system design, development, and quality assurance. We also provide a case

study with illustrative examples of LLM application in selected phases of the software development lifecycle. By setting the scope that comprises the early development phases, code and artifact generation, and the integration into software engineering processes, we aim to stimulate a broader discussion and uncover potential avenues for the application of LLMs in software engineering. We hope this exploration can serve as a starting point for more comprehensive investigations in the future.

Outline The paper is organized as follows: Section 2 describes our view of the prospects of software engineering applications of LLMs and the current state-of-the-art. Section 3 reports on our case study dialogues with ChatGPT and Bard for developing requirements, design and test cases of a "search and rescue" scenario. Section 4 discusses the challenges in adopting LLMs for software engineering, Section 5 depicts scenarios that could evolve if some or all of the challenges can be surpassed, and Section 6 concludes the paper.

2 LLMs in Software Engineering: Prospects and State-of-the-Art

The idea of embodying human expertise in a software system goes back to the roots of AI, to knowledge-based systems [20] and in particular expert systems [7]. Expert systems had applications in various domains, ranging from mathematics to healthcare and also software engineering [5]. A further step is conversational software engineering, i.e. engineering with the help of software-based systems which are capable of processing natural language data to simulate a smart conversational process with humans [31]. The first ideas for conversational software engineering go back to [38], which proposes an architecture for enabling assistant agents to reply to questions asked by naive users about the structure and functioning of graphical interfaces. [31] gives an overview on the field of conversational agents and their impact in the field of software engineering, but LLMs are not among the surveyed techniques nor are software development activities among the surveyed applications.

LLM-assisted software engineering goes a step further and promises to support developers in a conversational way with expert knowledge along the whole software lifecycle. Today, there are first experiments with LLMs helping software developers with common software engineering tasks such as ambiguity resolution in software requirements, method name suggestion, test case prioritization, code review, and log summarization [41]. In [43], a catalogue of prompt patterns is proposed that guide software developers in performing common software engineering activities using LLMs, such as disambiguating specification, creating APIs, proposing software architectures, or simulating web application APIs.

The paper [37] presents an LLM-assisted prototype system, called the Programmer's Assistant, that supports conversational interactions of developer and system. The Programmer's Assistant is based on a code-fluent large language model and helps the developer by answering general programming questions,

by generating context-relevant code, by enabling the model to exhibit emergent behaviors, and by enabling users to ask follow-up questions that depend upon their conversational and code contexts. The AutoScrum system [40] supports a highly iterative Scrum process between LLM and humans where the LLM generates small chunks of text or code at a time, which are reviewed by the user and then committed into the database. The GPT2SP system [17] helps in estimating story points for implementing agile product backlogs.

In the following we will discuss how LLM-assisted software engineering can support the classical phases of a software development life cycle (for AI-based systems see, e.g., [19,18,44]), i.e., requirements engineering, design, validation and verification. A case study with illustrative examples for some of the approaches outlined here is given in Section 3.

2.1 Requirements Engineering

Requirements Engineering (RE) is a critical process in the development of software systems, involving the definition, documentation, and maintenance of software requirements. This is a multidimensional task that requires robust information retrieval, effective communication with diverse stakeholders, and the production of detailed textual descriptions. It is an endeavor that can prove challenging due to the complexity and breadth of the tasks involved. LLMs can significantly facilitate this process, providing support in several areas of RE such as requirements elicitation, specification extraction and refinement, and generating solution concepts and system architectures.

Extraction and Elicitation First experiments report that LLMs can be very helpful in specification extraction from documents. According to a recent study [46], LLMs that are coached by few-shot learning achieve better extraction results than state-of-the-art techniques such as Jdoctor [4] and DocTer [45].

In addition to specification extraction, LLMs can also be used to support the refinement of system specifications and other artifacts such as epics, stories, and tasks in agile development. For instance, an LLM can be prompted to ask questions to identify and fill gaps in system specifications. This can be particularly beneficial during the refinement processes in agile environments. Leveraging the domain expertise exhibited by LLMs, the refinement process can be tailored to different perspectives and roles such as development, testing, and security. For instance, our case study (see Section 3) is a simple example for elaborating and refining requirements to obtain a system design and test cases.

Moreover, the translation and summarization capabilities of LLMs can be used to generate a system concept. By appropriately prompting the LLM, it can organize the generated summary or system concept along certain axes, such as system goals, stakeholders, functional and non-functional requirements, and so on. Interestingly, LLMs can also be prompted to explicitly express any missing information for certain aspects, thereby providing a comprehensive picture of the system's requirements.

In terms of user interaction, LLMs can generate and elaborate on user flows in an application. This can be especially beneficial in the absence of an interactive system during the RE process. LLMs have demonstrated their capacity to derive plausible user stories from system descriptions, thereby providing a tangible rendering of the user experience.

Use Case Design In use case design LLMs exhibit considerable utility. For example, given a comprehensive system specification, LLMs can ascertain the various use cases present. This involves extraction and description of the specified interactions between the users and the system, as well as the system's response to these interactions.

LLMs can construct illustrative use case and sequence diagrams from the identified use cases. Use case diagrams provide a graphical representation of the system's intended interactions with its environment and users, making the system's functionality more understandable. Sequence diagrams detail the sequential order of interactions between different system components corresponding to a specific use case, adding another layer of clarity to the system's operations.

The use of LLMs in requirements engineering not only speeds up the process but also reduces costs, making them an efficient alternative to manual labor. This, coupled with the ability of LLMs to gather information regarding a particular target artifact, offers an intriguing and promising avenue for future exploration.

2.2 System Design

In the realm of system design, Large Language Models (LLMs) are emerging as beneficial tools. Their capabilities range from proposing design alternatives to identifying trade-offs. Given a description of a system's functionalities, LLMs can suggest varied design solutions, and generate diagrams for visual communication. This enables designers to survey design alternatives more efficiently, thereby expediting decision-making.

Architecture Large Language Models (LLMs) could play a crucial role in the formulation of architectural views, which are vital in providing a comprehensive overview of a system's structure. These views take into consideration both functional and non-functional requirements of a system, aiding in visualizing the system's organization and functionality. LLMs can also help in data modeling and API design.

In creating architectural views, LLMs can explore and present a multitude of possible architectural configurations. Each option is accompanied by an analysis of its strengths and weaknesses, providing a balanced evaluation of each potential architectural design. This ability to critically evaluate each option makes the decision-making process more efficient and data-driven. In [47], it is shown how this can be used for hardware-software Co-Design of Accelerators for deep neural networks. The experimental results indicate a substantial speedup compared to

a state-of-the-art method. [43] provide a prompt pattern that helps developers in requesting different architectural possibilities from the LLM.

Beyond suggesting and evaluating design options, LLMs can also proactively identify potential risks associated with each architectural option. Recognizing these risks early in the design phase can prevent costly errors down the line. Also, LLMs can potentially offer possible mitigation strategies to minimize these risks, thereby adding another layer of robustness to the system design process.

A possible advantage of utilizing LLMs in this way is the alignment it creates between system design choices and the system's requirements. This harmonization ensures that the chosen architectural design adequately meets the system's functional and non-functional requirements, leading to a more effective and optimized system. If there are trade-offs involved, LLMs can help to document these and support decision making in the architectural design process.

The capability of LLMs to align requirements and design is also particularly valuable in architectural reviews. It may enable a streamlined and efficient review process, with LLMs effectively serving as tools for cross-verifying the alignment between system requirements and design choices.

Data Modeling LLMs can identify and define core entities within a system. Core entities in a system refer to the main components or elements that make up that system, and understanding these entities and their interactions is crucial for effective system design and development.

When using LLMs for identifying and defining core entities, we are essentially harnessing their capability to understand and interpret natural language descriptions of a system. Given a detailed description of the system's features, functions, and behaviors, an LLM could potentially highlight what the central entities are, based on frequency of mentions, context, and importance in achieving system functionalities.

LLMs can generate relational data models from context descriptions. These models can be expressed in domain-specific languages like mermaid.js[3] for visual representation. LLMs can also accommodate different levels of abstraction in data models, such as those seen in ELT or ETL pipelines.

API Design Based on use cases that are identified and described in requirements engineering, LLMs can further assist in the design process by defining endpoints for a corresponding REST API. In a web application, endpoints are the URIs (Uniform Resource Identifiers) where specific resources reside and can be manipulated via HTTP methods such as GET, POST, PUT, and DELETE. An LLM can derive these endpoints from the use cases, ensuring that the functionality encapsulated in each endpoint aligns with a specific user-system interaction.

By extracting use cases and generating REST endpoints from system specifications, LLMs streamline the design process, reducing time and effort required and minimizing the potential for human error. LLMs can also help the designers to create ansd simulate APIs (see, e.g., the corresponding patterns in [43]). This

[3] `https://github.com/mermaid-js/mermaid`

allows designers to devote more attention to other critical aspects of the system design, such as performance optimization and user experience.

2.3 Code Generation

Code generation and testing are key components of the implementation phase in software engineering. LLMs show remarkable capabilities in code generation, given a distinctive task specification for a small enough coding task [12]. We think that the combination of these coding capabilities with systematic problem decomposition assisted by LLMs could greatly expand the application domain from only coding singular tasks to providing code artifacts for large, complex, coordinated software systems.

LLMs have shown promising results in generating code snippets and test cases, tailored to specific requirements and design specifications. E.g., by combining code snippet generation of ChatGPT with tools for Language-Driven Engineering and modelchecking, [8] achieve a no-code development of a point-and-click adventure web game. Other interesting LLM-applications are generating code guided by test-driven user-intent formalization [24], or translating code with the help of LLM-based transpilers [21]. By automating such tasks, software engineers can focus more on solving complex problems while trusting the LLMs for routine and repetitive tasks.

These approaches can be combined with artifacts from the previous phases. For example, it is possible to create unit tests for REST endpoints and their respective description and sequence diagrams. Following for example a test-driven development approach, this test can the be passed to a subsequent call to the LLM asking for an implementation based on the specification that aims to pass the unit test.

2.4 Quality Assurance, Testing and Verification

As software projects grow in size and complexity, code reviews become crucial to maintaining the quality and integrity of the codebase. LLMs can be employed to conduct automatic code quality assessments, identifying potential problems, suggesting improvements, and providing recommendations based on best practices. There are several approaches for enhancing code review using LLMs, e.g. for providing useful suggestions on human-written code [30] or for supporting code review of pull requests of GitHub workflows [16]. This can lead to a more efficient code review process and improve overall code quality. However, currently ChatGPT can recognize some security code flaws but – not being trained for such applications – it recognizes many false positives so that its output has to be examined with great care [1].

The paper [14] provides an overview of conversational and potentially autonomous testing agents and proposes a framework for conversational testing agents that are potentially autonomous and supported by existing automated testing techniques. Such an LLM-system could not only become an intelligent testing partner to a human software engineer, but also be able to handle typical

testing related tasks autonomously. Similarly, in [39] an adaptive test generation technique is presented that uses an off-the-shelf LLM for automatically generating unit tests. Still, tests generated by ChatGPT often suffer from correctness issues [48]. As a remedy, in [48] an iterative test refiner is proposed for reducing compilation errors and incorrect assertions of ChatGPT-generated tests.

Furthermore, as current programming benchmarks are limited in both quantity and quality, Liu et al. [27] present a ChatGPT-based evaluation framework, called EvalPlus, for automatically generating and diversifying test inputs. The authors show that EvalPlus can considerably improve the popular HUMANEVAL benchmark [12].

Combining LLMs with automated verification tools is a promising avenue. In [11], a symbolic bounded model checker cooperates with an LLM for debugging and verifying code. If the model checker detects a violation of a memory safety property, the code is fed to the LLM for repair. The corrected code is passed again to the model checker for a verification check. As mentioned above, in [8] model checking ensures the correctness of an LLM-generated web application.

3 Case Study

For illustrating the software development abilities of LLMs we considered a simple search-and-rescue scenario (see [3]) and focus on the early phases, i.e., requirements engineering and design as well as testing.

In the scenario, a robot agent searches for victims in a probabilistic environment over a discrete graph of locations. Locations can be in fires that can ignite and cease. The goal of the agent is to rescue the victims and to bring them to safe locations. Note that a version of our example is accessible via the web (e.g. [3]), and therefore potentially part of ChatGPT's training data.

By interacting with ChatGPT and also with Bard we conducted two systematic developments consisting of a (textual) requirements specification and an object-oriented design specification in form of a class diagram and of pseudocode of the agent behavior. With Bard, we also developed the basic unit tests and system tests of the scenario. In the following we present excerpts of these developments and comment the results.

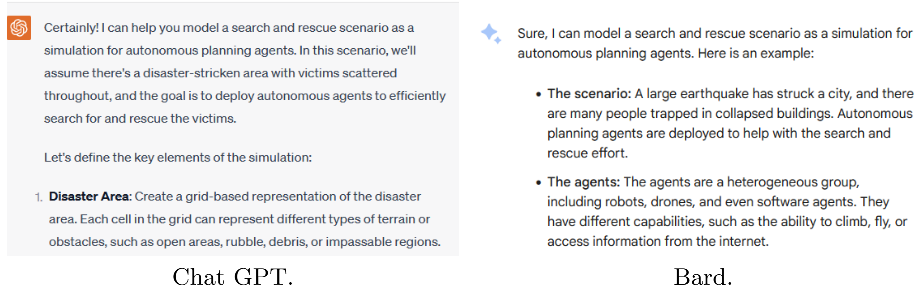
The conversation with ChatGPT-3.5, July 20, 2023 version was held on July 25, 2023 and the one with Bard on July 16, 2023. See the footnotes for links to the conversation with ChatGPT [4] and the one with Bard [5].

3.1 Requirements

First, we ask ChatGPT and Bard whether they can model a search and rescue scenario as a simulation for autonomous planning agents. Both give appropriate

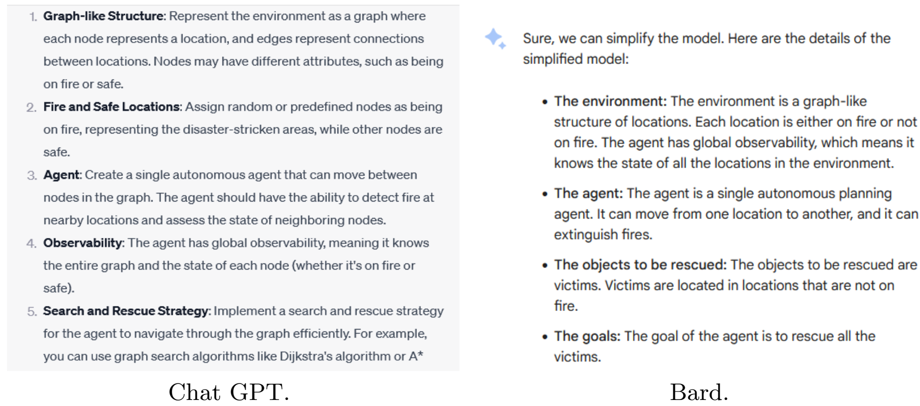
[4] ChatGPT `https://chat.openai.com/share/a93d844d-e542-4997-a7d5-0d254e007c08`

[5] Bard `https://g.co/bard/share/c51838296a3c`



Chat GPT. Bard.

Fig. 1. Search-and-Rescue Requirements (excerpts).



Chat GPT. Bard.

Fig. 2. Simplified Search-and-Rescue Requirements (excerpts).

answers: Bard proposes an abstract earthquake scenario and some of the challenges for a simulation such as the complexity of the environment; ChatGPT provides a more operational description of the key elements of the simulation (see Figure 1).

Then we ask both LLMs to instantiate the abstract model to just one agent with global observability and an environment consisting of a graph-like structure of locations where locations may be on fire. Again both LLMs answer with appropriate textual requirements descriptions, the one of ChatGPT being more concrete and proposing also two possible graph search algorithms (see Figure 2).

3.2 Design

For developing a design specification, first we ask the LLMs to provide a UML diagram of the scenario domain. Both LLMs present plausible class diagrams and corresponding pseudocode of the important classes and their relationships. Asked about operations, they also add appropriate methods. Concerning the additional constraint that the agent should extinguish the fire when it moves to a location in fire, ChatGPT adds a corresponding clause to the body of the

```
class Agent:
    def __init__():
        self.currentLocation = None  # Reference to a L
        self.rescuedVictims = []     # List of Victim o

    def move_to(self, location):
        self.currentLocation = location
        if location.isOnFire:
            self.extinguish_fire()
```

Chat GPT Move operation.

```
class Agent {
  canMove: Boolean
  canExtinguishFire: Boolean
  location: Location

  def move(toLocation: Location): Unit
  def extinguishFire(location: Location): Unit
  def rescueVictim(victim: Victim): Unit
}

class Simulation {
  locations: List[Location]
  agent: Agent
  victims: List[Victim]
}
```

Bard Move operation.

Fig. 3. Search-and-Rescue Class Diagram (excerpts).

```
def behavior_loop(self):
    while not self.is_terminal(self.current_state):
        # Step 1: Select the best action using MCTS
        action = self.select_action(self.current_state

        # Step 2: Execute the chosen action and observ
        next_state = self.simulate_action(self.current

        # Step 3: Update the agent's current state bas
        self.current_state = next_state

        # Step 4: Evaluate if there are any victims to
        for victim in self.current_state.victims:
            if victim.priority >= 0.5:  # Some priorit
                self.rescue_victim(victim)

        # Step 5: Extinguish fire if the agent is at a
        if self.current_state.isOnFire:
            self.extinguish_fire()
```

Chat GPT.

```
def agent_behavior(agent, environment):
  # Monitoring
  data = environment.monitor()

  # Analysis
  goals = agent.analyze(data)

  # Planning
  tree = MCTS(agent, data, goals)
  action = best_action(tree)

  # Execution
  environment.execute(action)

  # Logging
  agent.log(data, goals, action)

  return action
```

Bard.

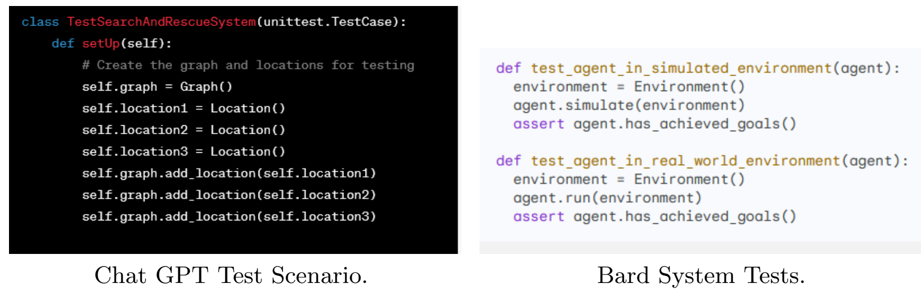
Fig. 4. Search-and-Rescue Agent Behavior (excerpts).

"move" operation (see Figure 3 left) whereas Bard remains abstract (see Figure 3 right) and mentions the constraint only in the documentation.

Then we want to construct an online planning algorithm for the agent to perform effective rescuing. Both LLMs give informed discussions about possible algorithms. ChatGPT proposes A* search and mentions also "planning algorithms like D* Lite, Incremental A*, or variants of Monte Carlo Tree Search (MCTS)." Bard proposes A* search and Dijkstra's shortest path algorithm as well as probabilistic and Bayesian planning; it also mentions MCTS for probabilistic scenarios. Concerning MCTS, both LLMs provide appropriate pseudocode for MCTS and also for the behavior of the agent using MCTS for online planning. ChatGPT is again more concrete and considers the constraint concerning the fires (see the last two lines of Figure 4 left) whereas Bard remains abstract (see Figure 4 right).

3.3 Testing

Asked about unit and system tests, both LLMs proposed relevant tests for both test categories. Again Bard is more abstract and proposes unit tests corresponding to the abstract operations of the MAPE loop whereas ChatGPT defines more concrete tests using the unittest testing framework. Concerning the system tests, ChatGPT offers a simple concrete scenario (consisting of three locations and two victims) whereas Bard defines abstract tests for a simulated and a real-world environment (see Figure 5).



Chat GPT Test Scenario. Bard System Tests.

Fig. 5. Search-and-Rescue Tests (excerpts).

3.4 Discussion

We observed interesting and stimulating generated outputs for the software engineering tasks in our case study. The example is quite simplistic, so the scalability of our "naive" prompting approach remains to be evaluated. At times, we observed a difference between the generated artifacts (e.g. the relation multiplicities in a class diagram) and our expectations. We attribute this to ambiguities in our description of the requirements. We think that systematic and potentially automated evaluation of such qualitative divergences is a highly interesting and valuable direction for future research (see also Section 4.2 for challenges of evaluating generated output).

4 Challenges in Adopting LLMs for Software Engineering

In the following we outline several challenges that need to be addressed when adopting LLMs in software engineering.

4.1 Integration with Large Context

Integrating LLMs into a larger domain and its context, such as existing processes, background knowledge, technical environments, and tools, is challenging.

LLMs need to understand and capture the context-specific information to provide meaningful and actionable insights to software engineers, requiring new methods and processes for contextualization [35].

Divide and Conquer Utilizing a large language model necessitates an organized, progressive approach, particularly for tackling intricate tasks such as the design of a software architecture. The first step entails employing the model to construct a high-level plan. This involves identifying the key components of the architecture, comprehending how these parts interrelate, and outlining the basic procedures necessary to create the structure. It is crucial at this stage to recognize that the output will be a broad overview, serving as a roadmap for subsequent, more detailed design stages.

After crafting a high-level plan the model is directed to focus on each component individually. This involves generating detailed information for each part, treating them as separate tasks. This methodology enables a deeper exploration of the unique characteristics of each component.

Several open-source projects explore LLM-based automated decomposition for software engineering, e.g., AutoGPT[6], MetaGPT[7], GPT-Synthesizer[8], and gpt-engineer[9]. However, finding the "right" level of abstraction remains challenging. For further evaluation or even automating and improving abstraction level decisions, introducing quantifiable or ranking (comparative) quality measurements would be highly valuable. These metrics could assess the relevance, coherence, and depth of the generated content w.r.t. a given task, offering potential avenues for future research. See also the following Section 4.2 on evaluation and testing of generative output for further elaboration on this matter.

Scaling to Large Context Managing large domain contexts within the constraint of LLM's limited prompt sizes poses a significant challenge. Prompt engineering, leveraging known relations and abstractions in software engineering, can help circumvent this issue. By distilling complex concepts into concise, structured prompts, we can guide the model's responses more effectively. Additionally, vector database approaches that utilize embeddings of code and documentation artifacts can help manage these large contexts. These embeddings capture semantic information in a compact form, aiding in maintaining contextual continuity across prompts.

Combining LLMs with knowledge graphs is another promising avenue to scale to larger contexts [35]. Knowledge graphs provide structured and interconnected representations of domain knowledge, which can be leveraged to provide context to LLMs beyond their prompt limits. The graph's nodes and edges can be transformed into prompts or fed into the model as needed to support context-rich generation tasks. Creating heuristics of how to organize prompt information

[6] `https://github.com/Significant-Gravitas/Auto-GPT`
[7] `https://github.com/geekan/MetaGPT`
[8] `https://github.com/RoboCoachTechnologies/GPT-Synthesizer`
[9] `https://github.com/AntonOsika/gpt-engineer`

from knowledge graphs, or how to use LLM's agentic capabilities to derive these heuristics contextually on the fly seems a large area for future research. Also, incorporating knowledge graphs already when training or finetuning LLMs is an interesting but challenging direction.

Semantic Interfaces for Natural Language Artifacts When managing numerous components in a larger artifact, the task of keeping track of their interfaces and relations can become highly complex. The context window of a large language model (LLM) is limited, making it unfeasible to hold all the component details simultaneously. However, if each component is well-defined, it is not necessary to retain all details; maintaining a record of their interfaces in the LLM context would suffice. In addition to generating the complete component, it is thus pivotal to create a succinct interface for the LLM context.

This concept is well-established for formal artifacts such as code, where interfaces denote a defined point of interaction between two components. However, the notion of an interface becomes nebulous when dealing with natural language artifacts, which are inherently ambiguous and lack the structured nature of formal code. The challenge lies in defining these interfaces for natural language artifacts. LLMs, with their robust language understanding capabilities, could be a valuable tool for creating interfaces for these ambiguous, natural-language-based artifacts. However, the method of achieving this remains unclear and requires further exploration. This could be a promising avenue of research for more efficient utilization of LLMs in complex, multi-component tasks.

4.2 Evaluating and Testing Generative Outputs

Evaluating the output generated by LLMs is a crucial aspect of their deployment. Primarily, this involves measuring the factuality, relevance, coherence, and depth of the generated content. LLMs are known to suffer from so called hallucinations that create seemingly plausible but factually or logically wrong outputs [2,22]. Therefore, accurately evaluating facual correctness of generated outputs is highly important. However, it is unclear how to perform such an evaluation without human labor in the abscence of an already labeled ground truth dataset. Relevance assesses whether the output corresponds well with the input prompt, coherence measures the logical consistency and fluidity of the response, and depth gauges the level of detail and complexity. These evaluations can be conducted both qualitatively, through human reviewers, and quantitatively, using metrics such as BLEU, ROUGE (for references see [9]), and METEOR [25]. Furthermore, evaluation might also consider the model's ability to avoid generating harmful or biased content. However, these metrics should be used in combination with human judgment, as they may not capture all nuances of language and context. For a recent survey on LLM evaluation, see [10].

A novel challenge arises when the output from the LLM is "better" than the ground truth, which refers to the model generating a response that is more accurate, insightful, or comprehensive than the expected answer. In this case, traditional metrics that penalize divergence from the ground truth could unfairly

penalize the model. Therefore, flexible evaluation methods that can appreciate and reward such enhancements are necessary. One approach could be to utilize expert human reviewers who can appreciate such improvements in the context of the task at hand. Additionally, considering the use of "range-based" or "bucket" scores, which allow for a certain degree of deviation from the ground truth without penalty, could be another viable approach. These evaluations should be designed in a way that they encourage innovative and high-quality responses, rather than just adherence to a predefined answer. However, this area remains largely unexplored and warrants further research.

4.3 Challenges in Practical Use

Evaluating Value Creation and Utility In some instances, the time invested in scrutinizing, understanding, and modifying the output generated by an LLM could potentially outweigh the advantages of automation. It is therefore essential to identify the scenarios where LLMs can add true value and efficiency in terms of time and resources. In these scenarios, LLMs can augment human productivity by automating tasks such as code generation, bug detection, and documentation, among others.

Accountability for Generated Content Maintaining human accountability for the outputs generated by LLMs is of utmost importance, particularly when automatic evaluation of generated output remains an open research issue (as discussed in Section 4.2). These models, while powerful, are tools that aid in various tasks, and the ultimate responsibility for their application rests with the humans employing them. This accountability is pivotal in ensuring ethical, lawful, and appropriate use of LLMs. Particularly in fields like software engineering, where potential consequences of errors can be significant, human oversight of model outputs is crucial. This situation accentuates the importance of robust review capabilities within systems deploying LLMs. As long as the automatic evaluation of generated outputs is not entirely reliable or comprehensive, the human review and revision of these outputs not only ensure quality but also embed a layer of human judgement and responsibility, adding a vital layer of safety and accountability.

Legal Uncertainty and Copyright Issues Legal issues regarding the copyright of content generated by large language models (LLMs) represent a complex and as yet unresolved area of discussion. Traditional copyright laws are designed around human authorship, and how these apply to machine-generated content remains a matter of debate. Questions about who owns the rights to the content generated by these models – the developers of the model, the users who prompt the generation, or perhaps no one at all – are currently under examination. Furthermore, there is the question of whether LLM outputs, if they inadvertently reproduce or mimic copyrighted content, constitute a violation of existing copyright laws.

Privacy Concerns Privacy and personally identifiable information (PII) considerations are paramount when using large language models (LLMs) in a professional context, such as in requirements engineering, system design, code generation, and quality assurance in software engineering. Since these models learn from the data they are trained on, there is a risk they may inadvertently memorize and reproduce sensitive information. In a professional setting, this could lead to the disclosure of confidential business information, proprietary algorithms, or personal data of stakeholders, potentially violating privacy laws and ethical guidelines. As LLMs increasingly find use in diverse fields, it becomes crucial to implement robust mechanisms to prevent the leakage of sensitive information. These might include data anonymization techniques, strict access controls, and regular audits. Also, clear guidelines about what types of data should and should not be input into the model are essential. Given the seriousness of these concerns, the ongoing development and refinement of privacy-preserving techniques in AI models represent an area of significant importance in the field of AI ethics and governance [23,26].

Hosting open-source LLMs in private environments can help alleviate these concerns. This allows organizations to keep their data in a secure network, reducing the risk of data breaches. Strict access controls and security protocols can further enhance data security.

Model Efficiency Increasing the resource efficiency of large language models (LLMs) is a vital consideration, especially as the scale and complexity of these models continue to grow. One way to achieve this is through model compression techniques, which aim to reduce the size of the model without significantly impacting its performance. Methods such as quantization, pruning, and knowledge distillation are commonly employed. Quantization reduces the precision of the numbers used in the model's computations, effectively shrinking its size [13]. Pruning involves removing less important connections in the model's neural network, leading to a sparser but still effective model [15]. Knowledge distillation, on the other hand, is a process where a smaller model (student) is trained to mimic the behavior of a larger model (teacher), thereby achieving similar performance with less computational resources. These techniques can hopefully lead to more efficient LLMs that maintain high performance while reducing both the computational power required and the currently associated resource requirements and carbon footprint [29].

5 What Could Happen if the Challenges are Resolved?

So far we have been concerned with the software engineering applications of LLMs *at hand*. As the discipline is shifting, we see major opportunities as well as severe challenges. Naturally, no one can predict future developments that this new technology might bring about, but we can imagine various scenarios of how the integration of LLMs into software engineering might play out; we might mostly sort these scenarios by the (necessary) involvement of LLMs in all

standard tasks of software engineering and likewise by the impact they might have on the discipline.

The simplest assumption we might make is just based on the capabilities we have seen LLMs exert today. Through integration of today's LLMs into the standard tooling for routine software development, we achieve Scenario 1.

Scenario 1 (A Better Bat) *LLMs become integrated into standard tools for software engineering. As such, they augment software IDEs with features like integrated code suggestion, adaptation of code samples from the web, or the generation of documentation from code (or vice versa). As sketched throughout this paper, they might also augment tools concerned with requirement analysis (by allowing to automatically generate summaries or check for inconsistencies in given descriptions) and thus aid and accelerate the whole software engineering life cycle. However, the standard processes involved in software engineering remain intact (albeit somewhat shifted w.r.t. the distribution of human effort) and well-known models for software engineering still apply. The long-term impact of LLMs is somewhat comparable to that of more powerful debuggers or better IDEs in the past.*

As long as LLMs are not entirely disregarded as a technology, the achievement of Scenario 1 appears almost inevitable.

Scenario 2 (A Game Changer) *LLMs become integrated into the standard model for software engineering processes and profoundly alter the way software is developed. Standard techniques (and standard implementations of them) are established that can reform or entirely replace parts of the software engineering process. As such, variants of LLMs might take over some roles (providing an AI scrum master, e.g.) or tasks (writing and updating documentation, e.g.). Education for software developers now has to include correctly dealing with LLMs and shift focus towards the remaining "human tasks" in software development. However, the produced artifacts (the compiled software, documentation, etc.) still closely resemble those generated before and compatibility with earlier products and also with earlier processes is maintained: Human developers are easily able to alter (and, if necessary, emulate) any machine output and scale LLM integration according to their preference. Through more powerful development processes, new software products may emerge whose complexity was previously not feasible for development teams not using LLM-enhanced processes (quickly coding against a wide range of semantically similar but syntactically diverse APIs, e.g.). The long-term impact of LLMs is somewhat comparable to that of (increasingly) higher programming languages or development paradigms in the past.*

For this range of APIs we might imagine a smart home device that can be enabled to talk to refrigerators without the need for a standardized API for refrigerators, but by describing the desired interaction as well as a long list of vendor-specific API calls and thereby generating modules for hundreds or thousand different devices within a reasonable time frame. However, what we

end up with is still just a large code base that *could* have been written by a (albeit larger) human team.

Scenario 3 (An Entirely New Game) *LLMs fundamentally change the way we think about software and software development. LLMs become integrated in any stage of software development (and might even be the glue that keeps the process together or drives the process in the first place) and/or integrated into many artifacts that are produced. The pervasive accessibility of LLMs blurs the border between the artifact and the process that generates it, as capable LLMs can further develop (parts of) the software at any stage without human interaction. As such, LLMs become an inalienable part of any software product as even shipped software might self-adapt according to a specific user's preference via an LLM that can re-program (parts of) the original product or utilize LLMs for any means of inter-application communication (instead of protocols and APIs in today's sense). Software in today's sense is either reduced to work as a foundation of an LLM-based software ecosystem (similar to how Unix commands support today's software products) and/or its functionality is quickly re-implemented within that ecosystem. Akin to the vision of software gardening replacing software engineering [18], the role of human software developers changes entirely to a manager of AI-induced software development processes. These big changes in the software ecosystem are either justified by vastly increased productivity or entirely new capabilities only reached by this new kind of software.*

Naturally, what new capabilities software might reach in the future remains yet to be seen. However, we feel positive that more powerful development paradigms enable more powerful software products in the long run.[10]

As a possible endpoint to that line of thought, Liventsev et al. [28] already invoke the vision of "fully autonomous programming" (for now) by running an LLM-based loop for improving a sought-for program aided by prompt generation provided by a genetic algorithm.

6 Concluding Remarks

LLM-assisted software engineering holds a significant potential for revolutionizing software processes. However, there are challenges that need to be addressed, such as evaluating generative output, integration with business and technical contexts, and understanding the practical implications of automation. As research continues to advance in this area, we can expect considerable improvements in the right direction, providing new opportunities for the software engineering domain to embrace the power of large language models.

Addressing the challenges discussed, future work in LLM-assisted software engineering should delve deeper into using LLMs across various facets of the software process lifecycle that have not yet been thoroughly explored. Beyond

[10] Despite the validity of the Church–Turing thesis, more powerful tools enable more products in practice.

the application examples we outlined in this work, additional application areas include implementation of generated system designs, operations and monitoring, where LLMs could potentially contribute to automated incident detection, root cause analysis, and decision support for resolution strategies. Combining verification tools with LLMs and applying LLMs to system security are other promising directions. Additionally, there is the domain of data engineering where LLMs could assist in tasks like data cleaning, transformation, and metadata management. Furthermore, leveraging LLMs in analytics to generate insights from complex and diverse data sets, automating data interpretation, or enhancing decision-making processes also presents compelling research opportunities.

Acknowledgements We thank the anonymous reviewer for constructive criticisms and helpful suggestions.

References

1. C. Anley. Security code review with ChatGPT. NCC Group, February 9, 2023. `https://research.nccgroup.com/2023/02/09/security-code-review-with-chatgpt/`, accessed on June 20, 2023.
2. Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, et al. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.
3. L. Belzner, R. Hennicker, and M. Wirsing. Onplan: A framework for simulation-based online planning. In C. Braga and P. C. Ölveczky, editors, *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2015.
4. A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. In F. Tip and E. Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018.
5. B. I. Blum and R. F. Wachter. Expert system applications in software engineering. *Telematics and Informatics*, 3(4):237–262, 1986.
6. T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
7. B. G. Buchanan, R. Davis, R. G. Smith, and E. A. Feigenbaum. *Expert Systems: A Perspective from Computer Science*, page 84–104. Cambridge Handbooks in Psychology. Cambridge University Press, 2 edition, 2018.
8. D. Busch, G. Nolte, A. Bainczyk, and B. Steffen. ChatGPT in the loop. In this volume, 2023.
9. E. Y. Chang. Examining GPT-4: Capabilities, implications, and future directions.
10. Y. Chang, X. Wang, J. Wang, Y. Wu, K. Zhu, H. Chen, L. Yang, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie. A survey on evaluation of large language models. *CoRR*, abs/2307.03109, 2023.

11. Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification. *CoRR*, abs/2305.14752, 2023.
12. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
13. T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339, 2022.
14. R. Feldt, S. Kang, J. Yoon, and S. Yoo. Towards autonomous testing agents via conversational large language models. *CoRR*, abs/2306.05152, 2023. Accessed on June 29, 2023.
15. E. Frantar and D. Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. *CoRR*, abs/2301.00774, 2023.
16. M. Fu. A ChatGPT-powered code reviewer bot for open-source projects. Cloud Native Computing Foundation , June 6, 2023. `https://www.cncf.io/blog/2023/06/06/a-chatgpt-powered-code-reviewer-bot-for-open-source-projects/`, accessed on July 20, 2023.
17. M. Fu and C. Tantithamthavorn. GPT2SP: A transformer-based agile story point estimation approach. *IEEE Trans. Software Eng.*, 49(2):611–625, 2023.
18. T. Gabor. *Self-adaptive fitness in evolutionary processes.* PhD thesis, LMU, 2021.
19. T. Gabor, A. Sedlmeier, T. Phan, F. Ritz, M. Kiermeier, L. Belzner, B. Kempter, C. Klein, H. Sauer, R. N. Schmid, J. Wieghardt, M. Zeller, and C. Linnhoff-Popien. The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. *Int. J. Softw. Tools Technol. Transf.*, 22(4):457–476, 2020.
20. I. Goldstein and S. Papert. Artificial intelligence, language, and the study of knowledge. *Cogn. Sci.*, 1(1):84–123, 1977.
21. P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh. Attention, compilation, and solver-based symbolic analysis are all you need. *CoRR*, abs/2306.06755, 2023.
22. S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang. Who answers it better? an in-depth analysis of ChatGPT and Stack Overflow answers to software engineering questions. *CoRR*, abs/2308.02312, 2023.
23. S. Kim, S. Yun, H. Lee, M. Gubri, S. Yoon, and S. J. Oh. Propile: Probing privacy leakage in large language models, 2023.
24. S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao. Interactive code generation via test-driven user-intent formalization. *CoRR*, abs/2208.05950, 2022.
25. A. Lavie and A. Agarwal. METEOR: an automatic metric for MT evaluation with high levels of correlation with human judgments. In C. Callison-Burch, P. Koehn, C. S. Fordyce, and C. Monz, editors, *Proceedings of the Second Workshop on Statistical Machine Translation, WMT@ACL 2007, Prague, Czech Republic, June 23, 2007*, pages 228–231. Association for Computational Linguistics, 2007.
26. Y. Li, Z. Tan, and Y. Liu. Privacy-preserving prompt tuning for large language model services, 2023.
27. J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *CoRR*, abs/2305.01210, 2023.
28. V. Liventsev, A. Grishina, A. Härmä, and L. Moonen. Fully autonomous programming with large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'23)*, 2023.

29. A. S. Luccioni, S. Viguier, and A.-L. Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *CoRR*, abs/2211.02001, 2022.
30. R. McColl. On-demand code review with ChatGPT. NearForm blog, March 28, 2023. https://www.nearform.com/blog/on-demand-code-review-with-chatgpt/, accessed on June 20, 2023.
31. Q. Motger, X. Franch, and J. Marco. Software-based dialogue systems: Survey, taxonomy, and challenges. *ACM Comput. Surv.*, 55(5):91:1–91:42, 2023.
32. H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian. A comprehensive overview of large language models. *CoRR*, abs/2307.06435, 2023.
33. J. Nielsen. AI is first new UI paradigm in 60 years. Jakob Nielsen on UX, June 22, 2023. https://jakobnielsenphd.substack.com/p/ai-is-first-new-ui-paradigm-in-60, accessed on July 03, 2023.
34. L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.
35. S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu. Unifying large language models and knowledge graphs: A roadmap, 2023.
36. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. OpenAI, San Francisco, California, United States, 2019. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, accessed on July 05, 2023.
37. S. I. Ross, F. Martinez, S. Houde, M. J. Muller, and J. D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces, IUI 2023, Sydney, NSW, Australia, March 27-31, 2023*, pages 491–514. ACM, 2023.
38. J. Sansonnet, J. Martin, and K. Leguern. A software engineering approach combining rational and conversational agents for the design of assistance applications. In T. Panayiotopoulos, J. Gratch, R. Aylett, D. Ballin, P. Olivier, and T. Rist, editors, *Intelligent Virtual Agents, 5th International Working Conference, IVA 2005, Kos, Greece, September 12-14, 2005, Proceedings*, volume 3661 of *Lecture Notes in Computer Science*, pages 111–119. Springer, 2005.
39. M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. Adaptive test generation using a large language model. *CoRR*, abs/2302.06527, 2023.
40. M. Schröder. Autoscrum: Automating project planning using large language models. *CoRR*, abs/2306.03197, 2023.
41. G. Sridhara, R. H. G., and S. Mazumdar. ChatGPT: A study on its utility for ubiquitous software engineering tasks. *CoRR*, abs/2305.16837, 2023.
42. R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.
43. J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *CoRR*, abs/2303.07839, 2023.
44. M. Wirsing and L. Belzner. Towards systematically engineering autonomous systems using reinforcement learning and planning. In P. López-García, J. P. Gallagher, and R. Giacobazzi, editors, *Analysis, Verification and Transformation for*

Declarative Programming and Intelligent Systems, volume 13160 of *Lecture Notes in Computer Science*, pages 281–306. Springer, 2023.
45. D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey. Docter: documentation-guided fuzzing for testing deep learning API functions. In S. Ryu and Y. Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 176–188. ACM, 2022.
46. D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee. Impact of large language models on generating software specifications. *CoRR*, abs/2306.03324, 2023.
47. Z. Yan, Y. Qin, X. S. Hu, and Y. Shi. On the viability of using llms for SW/HW co-design: An example in designing cim DNN accelerators. *CoRR*, abs/2306.06923, 2023.
48. Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng. No more manual tests? Evaluating and improving ChatGPT for unit test generation. *CoRR*, abs/2305.04207, 2023. Accessed on June 29, 2023.
49. W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *CoRR*, abs/2303.18223, 2023.