


Towards LLM-Assisted System Testing for Microservices

Mustafa Almutawa* 
KAUST

Qusai Ghabrah* 
KAUST

Marco Canini 
KAUST

Abstract—As modern applications are being designed in a distributed, Microservices Architecture (MSA), it becomes increasingly difficult to debug and test those systems. Typically, it is the role of software testing engineers or Quality Assurance (QA) engineers to write software tests to ensure the reliability of applications, but such a task can be labor-intensive and time-consuming. In this paper, we explore the potential of Large Language Models (LLMs) in assisting software engineers in generating test cases for software systems, with a particular focus on performing end-to-end (black-box) system testing on web-based MSA applications. We present our experience building *Kashef*, a software testing tool that utilizes the advanced capabilities of current LLMs in code generation and reasoning, and builds on top of the concept of communicative agents.

Index Terms—Software Testing, Testing Automation, Large Language Models (LLMs), Communicative Agents.

I. INTRODUCTION

Today, there is an increasing adoption for the Microservices Architecture (MSA), especially in large-scale, distributed applications. Notable large corporations adopting this architectural pattern include Netflix, Amazon, and X (formerly, Twitter), among many others [1]. MSA is characterized by its decoupling of system functionalities into fine-grained, independent services that communicate through messaging and remote procedure calls (RPCs) [2]. Applications designed in MSA offer a range of benefits for developers and organizations, including flexibility, scalability, and increased resiliency to failures [3]. In addition, by decomposing large systems into smaller, standalone components, MSA allows for improved collaboration between different teams, resulting in a faster and more efficient software development process [4].

The benefits of MSA, however, often come at the cost of increased system complexity [1]. This is particularly due to the potentially large number of services involved, the varying technology stacks employed across different services, and the complex dependencies between services. For example, in 2015, Netflix was reported to have over 700 microservices, running in a distributed cloud infrastructure [5]. As a result of this increased complexity, testing MSA-based applications can be a serious challenge to developers [6].

Nevertheless, software testing is important to ensuring the quality of software and its fulfillment of technical and business requirements [7]. To guarantee the correctness and quality of software, engineers perform varying types of testing during the software development life cycle (SDLC). This includes

unit testing, integration testing, system testing, and acceptance testing, to name a few [8]. Done fully manually, the software testing task can be labor-intensive and time-consuming.

In order to improve and accelerate the process of software testing, researchers and practitioners have looked into automation. Automated testing refers to the use of tools and frameworks to automate the generation and execution of test cases. Over the past several years, there has been an increase in adoption of automated testing [8], with various tools being developed along the way [9], [10].

Our work comes as a continuation of the efforts applied in developing automated testing tools. We take a different approach from previous work in that we explore the capabilities of advanced large language models (LLMs) and investigate how they can improve software testing.

LLMs have been gaining huge popularity, especially after the release of the now-popular GPT models from OpenAI [11]. In the years that followed, LLMs specifically trained to generate code have emerged. Examples of such LLMs include Facebook’s CodeLlama [12], Anthropic’s Claude [13], and OpenAI’s Codex (which powers Github’s Copilot) [14]. Given the impressive abilities of modern LLMs in code generation, we presume that they can assist engineers in software testing tasks. Several research endeavors [15]–[17] have already explored the potential of LLMs in software testing. [18] offers a literature review of this relatively nascent field of software testing using LLMs.

In our inspection of the existing literature on LLM-based software testing, we have found that the majority of work is focused on unit testing and fuzzing [18], with limited work focusing on using LLMs for end-to-end system testing. Recognizing this gap in the literature, we are interested in examining the utility of LLMs in the broader objective of system-level testing. To this end, we set out to build an LLM-powered tool—named *Kashef*—that can assist developers in generating system tests. Our work is primarily concerned with testing web microservices applications, since they are common and typically difficult to test.

To build *Kashef*, we utilized existing LLMs, machine learning frameworks, and testing libraries. We also adopt the concept of multi-agent collaborative workflow [19]–[21] in our design for *Kashef*. The ultimate goal for *Kashef* is to be a one-stop solution for:

- 1) determining the functionalities of the system and the test cases needed,

*Equal contribution.

- 2) generating correct code for testing the system, and
- 3) executing the tests and providing feedback.

Currently, our implementation of Kashef only achieves the latter two objectives (i.e., generating and executing test cases). However, while the full implementation of Kashef is still a work in progress, we believe our experience and findings in building Kashef can inspire and help future work by the research community. As such, we highlight in this short paper the effort required to build an LLM-assisted testing tool, the challenges and limitations encountered along the way, and the preliminary results we obtained by using Kashef. The source code for Kashef is publicly available at <https://github.com/Kashef-KAUST/Kashef>.

II. BACKGROUND AND RELATED WORK

A. LLM-Assisted Software Engineering

The process of Software Engineering (SWE) involves many tasks, ranging from requirements collection to implementation and testing. With the emergence of LLMs, these processes have changed dramatically. We can witness this firsthand with tools such as ChatGPT, GitHub’s Copilot, and CodiumAI, as well as the various efforts spanning across different SWE tasks. As an example, in requirements collection, Ronanki et al. [22] found that ChatGPT was able to generate consistent and understandable requirements compared to those formulated by experts. When it comes to software design, Stojanovic et al. [23] demonstrated how ChatGPT was able to use three system descriptions to identify microservices and their dependencies for each system. Lastly, Denny et al. [24] evaluated the performance of Copilot and found that it was able to solve 60% programming problems in addition to being a promising way of learning programming. However, there are some challenges associated with using LLMs in SWE, including LLM integration into current SWE tools, using LLMs to automate SWE processes, and integrating LLMs into the various stages of SWE. There is also the challenge of evaluating and testing the outputs of LLMs in SWE context without the need of human judgment [25].

B. LLM-Assisted Testing

There has been work that specifically targets the usage of LLMs in software testing. Engineers at Meta created TestGen-LLM [15], which is an LLM tool that improves existing tests and verifies the generated tests against multiple criteria, including its measured improvement to the existing tests. It was evaluated on several Instagram features with a 25% increase in coverage and 73% of its testing code applied in production deployment at Meta. According to the paper, at the time of its writing, it is the only “industrial scale deployment of LLM-generated code” with assured test improvements. Another example includes the work of Jensen et al. [26] that explored the usage of LLMs in identifying software vulnerabilities and assessing the functionality of code. Their work evaluated 9 LLMs, including `text-davinci-003` as well as models from the GPT and Llama families. Their results show that LLMs can

achieve up to 95.6% and 88.2% accuracy for flagging security vulnerabilities and functionality validation, respectively.

One notable effort that is slightly similar to our work is presented by Daniel et al. [16]. They developed a GUI-based software testing tool that utilizes GPT-4. In contrast to monkey testing, which is a software testing technique that tests an application randomly through random actions, their approach leverages LLMs to understand current HTML snapshots and identify potential elements to interact with, keeping history of the changes these interactions cause to the state of the Document Object Model (DOM). Their evaluation of the branch coverage shows that they are able to achieve notably higher branch coverage compared to monkey testing.

However, Kashef differs from this work on a couple of points. First, while their work tests the application in-place (i.e., just simulating human interactions), ours aims to generate testing modules that will be a part of the testing suite of the application. Second, their approach is essentially randomized testing supported by LLM reasoning. That is, the LLM is used to reason about the next random element to interact with. Our focus instead is on verifying the functionality of the software requirements (system testing as opposed to fuzzing/monkey testing). Lastly, our end goal is for the LLM to identify the functional requirements of the system and generate relevant test cases, while their work is essentially an extension of monkey testing that utilizes LLMs to improve branch coverage.

Additionally, upon inspecting the code for their tool, we found a couple of obstacles that could obstruct its usage in a more general context. In particular, there is an assumption that interactable elements consist only of buttons and must have IDs to be referenced, which is not necessarily the case in real-world examples.

C. Communicative Agents

One big component of our work is the use of communicative agents to drive the testing process. The concept of communicative agents refers to chat-based language models that converse and collaborate with each other in order to achieve a particular task without human intervention. Multiple works have explored this concept. Camel [19] is an open-source library contributed by researchers. It proposes a role-playing framework and uses inception prompting, allowing agents to collaborate on completing a task and ensuring it aligns with the given prompt. ChatDev [20] is another example of the concept of communicative agents. Their contribution is a virtual chat-powered SWE company that simulates the whole software development process. Kashef is designed to leverage communicative agents, each tasked with a specific role in the testing process, thereby mimicking the approach a software tester would take in a black-box testing environment.

III. KASHEF DESIGN OVERVIEW

Kashef is an LLM-powered software testing tool that automates the generation and execution of tests for web-based microservices applications. The primary goal of Kashef is to

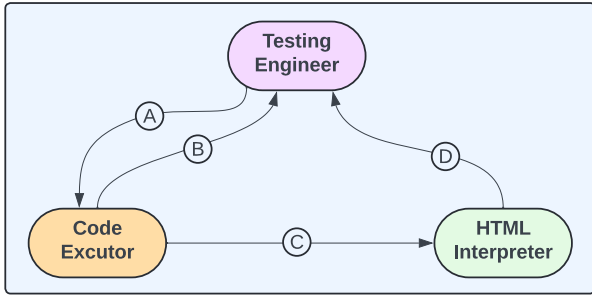


Fig. 1: Workflow of Kashef agents.

assist testing engineers in a) determining the test cases needed to test the system, b) generating test cases, and c) executing tests. As stated previously, our implementation currently only accomplishes (b) and (c). We plan on achieving (a) in future work.

Automating these tasks is challenging, especially if we are to view the MSA system under test as a black box and there is no given specification for the application interfaces. On the other hand, we assume, as it is common practice, that a deployed instance of the system executes in a controlled environment wherein various experiments can be run that poke at the system using a variety of scenarios. Thus, our primary aim is to identify interesting testing scenarios with as little human input as possible.

The main feature of Kashef is its utilization of the advanced capabilities of LLMs in reasoning and code generation. Our design for Kashef is based on the concept of communicative agents (§II-C). In this paradigm, different agents with different roles engage with each other (in a conversational manner) with the goal of completing a particular task, with no need for human involvement besides the initial task prompt. The multi-agent setup allows Kashef to generate tests in an automated and incremental manner, which is essential in our context of testing web applications since the testing task often involves performing multiple, sequential actions.

Before describing the specifics, it is worth to remark our rationale for using a multi-agent setup. We initially attempted to describe the test-case generation using step-by-step instructions and examples as part of a single prompt for the LLM. We generally observed that with this approach the LLM was unable to succeed and would fail to follow the instructions. Therefore, we set up the approach to offer simpler piecemeal tasks that can be solved by different LLM agents collaboratively.

The design of Kashef consists of three agents: Testing Engineer (TE), HTML Interpreter (HI), and Code Executor (CE). The TE and HI are LLM-based agents, while the CE is a tool to execute code. Serving as the entry-point in the tests generation workflow, the TE expects a simple prompt from the user specifying the action to perform on the application. For example, in the context of an e-commerce application, the prompt could be: “checkout a random product from the website www.example.com”. Figure 1 illustrates the workflow of the three agents. (A) Working in incremental and

collaborative fashion, the TE sends the generated code to the CE. (B) If the code execution fails, the CE simply returns the errors to the TE for code fixing. (C) Otherwise, if the code execution succeeds and HTML code is retrieved from the application under test, the CE sends the result to HI. (D) Upon receiving and analyzing the HTML code, the HI provides a detailed description of the key elements in the HTML page to the TE. This workflow continues until the task is completed or a certain iterations threshold is reached.

To implement Kashef, we used the machine learning frameworks LangGraph [27] (a module developed on top of LangChain) and AutoGen [21]. These frameworks enable the design of multi-agent workflows. Within each framework, we experimented with multiple LLM models, such as GPT-3.5, GPT-4, CodeLlama, and Llama2 to serve as the LLM for our agents. In addition to the machine learning frameworks, we also used Selenium, a testing framework that enables programmatic interaction with web browsers. Specifically, we instruct the TE of Kashef to use Selenium library when generating the testing code. The choice of Selenium comes as a result of its suitability for our testing approach as well as its widespread popularity.

As of this writing, we were not able to obtain a fully working setup with LangGraph (which is still a preview library under heavy development). The results in the rest of this paper are based on the AutoGen implementation.

IV. EXPERIMENTS AND EVALUATION

To evaluate Kashef, we define four tasks (IV-A). Further, for the LLM agents of Kashef, we make use of four different LLMs: `lama-2-70b-chat-hf` (Llama 2), `CodeLlama-70b-Instruct-hf` (CodeLlama), `gpt-3.5-turbo-0125` (GPT-3.5), and `gpt-4-0125-preview` (GPT-4). For each LLM, we execute 10 runs for each task, resulting in a total of 40 runs for each LLM. We report average results across the runs.

A. Testing Tasks

We provide Kashef with four testing tasks, with varying levels of difficulty. A testing task consists of a website and a task prompt that specifies for Kashef what functionality needs to be tested in that website. We use this message as an input prompt to the TE agent. Our selection of websites for the tasks includes an e-commerce demo, a word counter, a pastebin, and a to-do list. We aim to test websites with different layouts and different functionalities. For the task prompts, we construct them to be general and provide as little details as possible so we can assess the LLMs ability to reason on how to approach the tasks. The prompts for these tasks are given below:

- 1) **Task 1:** “Checkout a random item from this website <https://cymbal-shops.retail.cymbal.dev>. Make sure to complete the entire checkout process and randomize the item selection. When you reach the shipping and payment page, use the pre-filled information.”
- 2) **Task 2:** “Open this word counter website <https://looabuzfed.com/>. Then, I want you to verify the word counting functionality and ensure it is working correctly.”

TABLE I: Evaluation metrics and definitions.

#	Metric	Definition
1	Success Rate	The tool completed the task fully and successfully
2	Subtasks Completed %	The percentage of the full task that is completed (given that tasks consist of specific subtasks)
3	Code Regenerations	The number of code regenerations during a successful run due to errors
4	LLM Invocations	The number of LLM invocations during a successful run
5	HTML Non-utilization	The number of times the LLM agent did not utilize the HTML code
6	Versioning Ignored	Whether or not the TE preserved previously generated code in new iterations of the code
7	False Positive	The indication by the tool of a successful run when the actual outcome should have been otherwise (e.g., failure to complete the task)
8	Context Length Exceeded	Whether or not the LLM’s context length was exceeded
9	Random Product Selection	Whether or not random product selection was achieved (only for task 1)

- 3) **Task 3:** “I have a website for to-do list. Open the website and create a to-do by clicking on ‘create projoodle’. When having to input information, just provide random information. For date inputs, format the date like MM/DD/YY.”
- 4) **Task 4:** “I want you to verify the functionality of this pastebin website <https://privatebin.net>. Specifically, you have to ensure the post data is persistent by comparing the text you posted with the text found in the generated pastebin link.”

B. Metrics

To evaluate the performance of Kashef on the tasks defined earlier, we devise a set of evaluation metrics. We outline in Table I these metrics and their corresponding definitions. Note that the metrics are defined for a single run of Kashef.

We elaborate here on a few metrics that are not self explanatory. The reason behind metric 2 is the observation that for some tasks, Kashef is able to achieve a significant portion of the task but fails at a certain step. Therefore, we want to measure the subtasks completion to gain a detailed insight into the per-task performance. Metric 3 is useful because it measures the accuracy of Kashef and the consequent cost (since code regenerations involve LLM invocations), while metric 4 is helpful because it can approximate the cost of running Kashef.

In certain runs, the LLM does not utilize the HTML code properly, resulting in incorrect web interactions and Selenium exceptions like `NoSuchElementException`. Due to this reason, we devise metric 5. Further, in runs involving complex tasks with a long sequence of steps, the TE tend to discard previously generated code that accomplished earlier steps of the task, and generates new code that tackles the remaining steps with the assumption that previous interactions are saved in the browser session. This leads to errors because each code

execution starts from a fresh browser session. We refer to this issue as *code versioning* and measure it in metric 6.

Lastly, for the e-commerce task (task 1), we reason that random selection of products is essential to effective testing of the website. Therefore, we measure the number of times Kashef achieves random selection of products using metric 9.

C. Results

Out of the four LLMs we experimented with, GPT-4 yielded the highest performance. For clarity, we present results based on each LLM used.

GPT-4: Table II presents the test results for GPT-4. The metrics are averaged across the 10 execution runs of each task. For task 1, Kashef using GPT-4 successfully completes the task in 7 out of 10 runs (70% success rate). However, when measuring the percentage of subtasks completed, we find that on average, 90% of the full task is completed. Further, we find that on average, each successful run involves about 0.4 code regenerations and requires around 6.8 LLM invocations (accounting for both the TE and the HI). Out of the 10 runs, Kashef did not utilize the HTML code twice only and this occurred within a single run. During the 10 runs, Kashef did not ignore versioning, did not report false positives, and did not exceed the model’s context length. Kashef achieved random selection of products in all execution runs.

For task 4, creating a task on a to-do list website, Kashef did not succeed in any of the runs. We note that the reason for failing in this task was largely due to disregard for code versioning. In several runs, the TE does not retain previously generated code in new versions of the code. Because browsers sessions are cleared after every code execution, a subsequent code version that does not include earlier versions naturally fails to execute.

Due to space limit, we refer the reader to Table II for a summary of the results for all the tasks. However, we mention below a few observations that we think are noteworthy:

- In tasks 1 and 4, some of the failed runs are due to the HI refusing to analyze the HTML code. One of the responses from the HI is: “I’m unable to execute or interact with web pages directly...” The other responses of the HI are very similar. We are not sure of the reason behind this behavior, but we speculate that this might have happened due to model temperature, the HI misunderstanding its role, and/or the HI misinterpreting script elements as code that it should run.
- In one of the runs for task 2, the TE writes the testing script twice. In the first test script, the agent makes an arithmetic error in counting the words of its generated text, therefore concluding that the outputted word count from the website is incorrect. It then decides to test the functionality of the website again and succeeds in the second test script. This confirms observations made elsewhere [28] about how LLMs can struggle with certain simple tasks, and suggests that we should instruct the agents to use tools whenever possible during verification attempts.

TABLE II: Test results for GPT-4 model.

Task	Success Rate	Subtasks Completed %	Avg. Code Regeneration	Avg. LLM Invocations	Avg. HTML Non-utilization	Versioning Ignored	Avg. False Positives	Context Length Exceeded	Random Product Selection
1	70.0%	90.0%	0.4	6.8	0.2	0.0%	0.0%	0.0%	100.0%
2	100.0%	100.0%	0	5.1	0	0.0%	0.0%	0.0%	N/A
3	0.0%	46%	N/A	N/A	0.3	60.0%	0.0%	0.0%	N/A
4	80.0%	87.50%	0.3	5.3	0.1	0.0%	0.0%	0.0%	N/A

TABLE III: Test results for GPT-3.5 model.

Task	Success Rate	Subtasks Completed %	Avg. Code Regeneration	Avg. LLM Invocations	Avg. HTML Non-utilization	Versioning Ignored	Avg. False Positives	Context Length Exceeded	Random Product Selection
1	0.0%	33.33%	N/A	N/A	1.0	10.0%	20.0%	10.0%	80.0%
2	80.0%	83.33%	0.7	5.3	0	0.0%	20.0%	0.0%	N/A
3	0.0%	13.0%	N/A	N/A	0	0.0%	0.0%	0.0%	N/A
4	0.0%	22.5%	N/A	N/A	0	0.0%	10.0%	0.0%	N/A

GPT-3.5: Table III presents the results for GPT-3.5. The metrics are averaged across the 10 execution runs of each task. For task 1, Kashef using GPT-3.5 is unable to execute any of the trials successfully (0% success rate). However, when measuring the percentage of subtasks completed, we find that on average, 33.33% of the full task is completed. Since all trials failed to complete, we do not note the average LLM invocations and average code generation. The tool does not utilize the HTML code a total of 10 times (distributed among 4 out of the 10 trials). Out of the 10 trials, 1 ignored code versioning, 2 were false positives, and 1 exceeded the context length limit. The tool achieved random selection of products in 8 out of the 10 trials.

For space constraints, we refer the reader to Table III for a summary of the results for the rest of the tasks. However, we mention here a few observations that we think are noteworthy:

- In all tasks, the TE seems to struggle with generating correct Selenium code. Many of the code regenerations are a result of webdriver errors. It is also unable to fix these errors regardless of the multiple times it tries to. Additionally, instead of having one code block in a given response, the TE provides multiple code blocks in one response with each being a version of the code block prior to it. However, it does that for the first response only and before getting more context from the HI. Other than that, the TE seems to completely ignore the specification of providing the HTML after each code increment. We believe this is the main reason why many trials have failed. Lastly, the way that the code generated by TE handles errors is by printing the stack trace instead of the error description. Thus, when the CE executes the code and it fails, it doesn't provide a useful description of the error that the TE can use to fix the error.
- Tasks 2 and 4 are very similar in terms of the website's simplicity and the actions needed to test them. However, there is a big difference in their success rates (80% vs 0%, respectively). This is due to the absence of providing the HTML content of the website to the HI. In the code generated for task 2, the HTML content is printed to the HI, while this is not the case for task 4. We are still unsure as to why this behavior happens, especially since the system prompts are the same and the task prompts are very similar.

Llama 2: For Llama2, there were several challenges that hindered the collection of meaningful test results. First, the context length of Llama2 was too small to handle lengthy HTML codes retrieved from web pages. For task 1, 2, and 4, the model's context length was already exceeded by the first HTML code fetched from the landing page of the website. Second, the Llama2-powered TE was largely incapable of building the code incrementally and maintaining code versioning. Indeed, in several of the runs, the TE simply produced the entire code at once. That is, as opposed to first opening the website and retrieving the HTML code in order to proceed with next steps, the TE generated a boilerplate that approximated the steps necessary for accomplishing the task. Third, the code generated by the TE was often not formatted in one block, but instead interleaved with explanatory text. As a result, the CE was unable to effectively execute the full code produced by the TE.

Due to the aforementioned challenges, we did not run all the testing trials (compared to GPT models) because doing so seemed unproductive. We postulate that either the model is incapable of achieving our defined tasks or that our prompts require modifications.

CodeLlama: When running trials on the CodeLlama model, we faced a problem where it refused to comply with the system and task prompts due to ethical concerns. We observed that this behavior is affected by the temperature of the model. If we increase the temperature, it would occasionally generate the code we requested. However, code generation was very infrequent, which made it infeasible to conduct tests and obtain meaningful test results.

V. CURRENT LIMITATIONS AND CHALLENGES

It is worth noting that what we have accomplished barely scratches the tip of the iceberg in terms of system testing. The challenges ahead lie in generating and executing test cases that can stress the system, identifying stragglers and breaking conditions. Kashef is currently limited for two reasons. First, Kashef adopts the black-box testing approach and interacts with applications through one entry point (i.e., the frontend service). However, it would be more effective to also trigger other services independently and simultaneously. Second, Kashef is not yet designed to generate and execute test cases in parallel, which is essential to producing a workload repre-

sentative of the production environment that can reasonably test the microservices application.

Moreover, while Kashef is currently able to conduct simple black-box testing tasks, there are several challenges to address in this context as well. First, the output of Kashef does not yet scale reliably as the task complexity increases, as shown in the evaluation. This seems largely due to code versioning issues, where the complete code is not maintained across versions but rather discarded for newer versions that achieve later subtasks of the full task. Second, the job of crafting effective prompts for the LLM agents is challenging due to the manual effort required in revising and improving the prompts.

After successfully achieving the black-box testing objective, the natural progression of Kashef is using gray box testing and reaching a level of sophistication where Kashef can generate more complex tests that can effectively and automatically stress the system under test, reporting (mis)behaviors of interest when they arise.

VI. CONCLUSION AND FUTURE WORK

We presented our preliminary work on Kashef, an LLM-powered software testing tool that aims to automate the generation and execution of tests for web-based microservices applications. We discussed the design of Kashef, the evaluation metrics, and the results of our experiments. We found that Kashef was able to successfully complete some of the tasks, but faced challenges in others.

Additional agents: In actions that require a long sequence of steps, we observed that the TE stops building on previous code at one point, leading to execution errors (an issue we referred to as code versioning). Thus, we plan on introducing an agent responsible for ensuring that functional code from previous versions is preserved in new code iterations.

Prompt engineering and optimization: There are two strategies we are considering for improving the prompt. First, we are interested in exploring chain-of-thought and few-shot prompting techniques, which have been shown to provide performance improvements [18]. Second, we are interested in the concept of automatic prompt optimization [18]. In particular, we plan to explore DSPy [29], a framework for optimizing prompts for language models.

REFERENCES

- [1] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, 2018.
- [2] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," <https://martinfowler.com/articles/microservices.html>, Feb 2014.
- [3] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Information and Software Technology*, vol. 137, 2021.
- [4] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *SoutheastCon*, 2016.
- [5] Amazon Web Services, "AWS re:Invent 2015: A Day in the Life of a Netflix Engineer (DVO203)," <https://www.youtube.com/watch?v=mL3zT1iIKw>, 2015.
- [6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, 2021.
- [7] R. Sophocleous and G. M. Kapitsaki, "Examining the Current State of System Testing Methodologies in Quality Assurance," in *XP*, 2020.
- [8] M. Waseem, P. Liang, G. Márquez, and A. D. Salle, "Testing Microservices Architecture-Based Applications: A Systematic Mapping Study," in *APSEC*, 2020.
- [9] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo, "Microservices Integrated Performance and Reliability Testing," in *AST*, 2022.
- [10] L. Gazzola, M. Goldstein, L. Mariani, I. Segall, and L. Ussi, "Automatic Ex-Vivo Regression Testing of Microservices," in *AST*, 2020.
- [11] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, 2023.
- [12] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," 2024, arXiv:2308.12950.
- [13] Anthropic, "Claude," <https://www.anthropic.com/claude>, 2023.
- [14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021, arXiv:2107.03374.
- [15] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated Unit Test Improvement using Large Language Models at Meta," 2024, arXiv:2402.09171.
- [16] D. Zimmermann and A. Koziol, "GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver," in *ASEW*, 2023.
- [17] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," 2023, arXiv:2305.04764.
- [18] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," 2024, arXiv:2307.07221.
- [19] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society," in *NeurIPS*, 2023.
- [20] C. Qian, X. Cong, W. Liu, C. Yang, W. Chen, Y. Su, Y. Dang, J. Li, J. Xu, D. Li, Z. Liu, and M. Sun, "Communicative Agents for Software Development," 2023, arXiv:2307.07924.
- [21] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," 2023, arXiv:2308.08155.
- [22] K. Ronanki, C. Berger, and J. Horkoff, "Investigating ChatGPT's Potential to Assist in Requirements Elicitation Processes," in *SEAA*, 2023.
- [23] T. Stojanovic and S. D. Lazarević, "The Application of ChatGPT for Identification of Microservices," *E-business technologies conference proceedings*, vol. 3, no. 1, 2023.
- [24] P. Denny, V. Kumar, and N. Giacaman, "Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language," 2022, arXiv:2210.15157.
- [25] L. Belzner, T. Gabor, and M. Wirsing, "Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study," in *AISoLA*, 2023.
- [26] R. I. T. Jensen, V. Tawosi, and S. Almir, "Software Vulnerability and Functionality Assessment using LLMs," 2024, arXiv:2403.08429.
- [27] LangChain, "Langgraph," <https://langchain-ai.github.io/langgraph/>, 2024.
- [28] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," 2023, arXiv:2302.04761.
- [29] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines," in *ICLR*, 2024.