Enhancing Security Control Production With Generative AI

Chen Ling, Mina Ghashami, Vianne Gao, Ali Torkamani, Ruslan Vaulin, Nivedita Mangam, Bhavya Jain, Farhan Diwan, Malini SS, Mingrui Cheng, Shreya Tarur Kumar, Felix Candelario
Amazon AWS
New York, United States
{emorycl,ghashami}@amazon.com

Abstract

Security controls are mechanisms or policies designed for cloud based services to reduce risk, protect information, and ensure compliance with security regulations. The development of security controls is traditionally a labor-intensive and time-consuming process. This paper explores the use of Generative AI to accelerate the generation of security controls. We specifically focus on generating Gherkin codes which are the domain-specific language used to define the behavior of security controls in a structured and understandable format. By leveraging large language models and in-context learning, we propose a structured framework that reduces the time required for developing security controls from 2-3 days to less than one minute. Our approach integrates detailed task descriptions, step-by-step instructions, and retrieval-augmented generation to enhance the accuracy and efficiency of the generated Gherkin code. Initial evaluations on AWS cloud services demonstrate promising results, indicating that GenAI can effectively streamline the security control development process, thus providing a robust and dynamic safeguard for cloud-based infrastructures.

1 Introduction

In today's rapidly evolving digital landscape, safeguarding the security and integrity of cloud-based infrastructures has become a critical priority. The intricate nature and vast scale of modern cloud environments, coupled with the escalating sophistication of cyber threats, necessitate the deployment of robust and dynamic security measures. At the heart of these defenses are security controls, which are specific safeguards or countermeasures designed to detect, prevent, or mitigate risks to information systems. The development of security controls through traditional methods involves a series of labor-intensive and time-consuming steps. Security engineers must perform detailed research to stay updated on the latest threats, vulnerabilities, and best practices. They engage in comprehensive threat modeling to identify potential risks, evaluate their likelihood and impact, and devise effective mitigation strategies. The final phase involves crafting, testing, and deploying custom code and configurations for security controls. This process is both resource and time consuming, causing delays in implementing essential security measures and leaving systems vulnerable.

In light of these challenges, there is growing interest in leveraging generative AI to streamline and enhance the development of security controls. Generative AI has the potential to automate many of the labor-intensive aspects of this process, thereby significantly reducing the time and effort required to establish effective security measures. By accelerating the creation of security controls, organizations can more swiftly adapt to emerging threats, ensuring the ongoing protection and integrity of their cloud-based infrastructures.

1.1 Development of security controls

The development of security controls involves multiple intricate stages, each requiring careful planning and execution to ensure the robustness and efficacy of the controls. The process begins with security engineers identifying the specific service and resource, along with the appropriate type of control required for the pair. This initial stage is crucial, as it sets the foundation for the subsequent steps by determining the scope and focus of the security measures.

Once the service, resource, and control type are identified, the next stage involves writing Gherkin scripts. Gherkin, a domain-specific language, is employed to define the behavior of security controls in a clear and structured manner. Gherkins use plain language to describe the expected outcomes, making them accessible to both technical and non-technical stakeholders. Writing Gherkin scripts requires a deep understanding of the service and resource, as well as the specific security requirements they must meet.

After the Gherkin scripts are written, they undergo a rigorous review process. This step is critical to ensure the quality and accuracy of the scripts. Security engineers meticulously review each Gherkin script to verify that it accurately defines the intended security control and that it will function correctly when implemented. Once the Gherkin scripts have been thoroughly reviewed and validated, the next stage is the development of the actual code to execute the control. This involves translating the Gherkin-defined behaviors into executable code that can be deployed within the cloud environment. Finally, the code is deployed, and the security controls are put into operation. This deployment stage includes thorough testing to ensure that the controls function as intended and effectively mitigate the identified risks. Continuous monitoring and maintenance are also necessary to adapt to new threats and evolving requirements.

Figure 1 shows all development stages discussed above. The entire loop—from identifying and analyzing API documentation to reviewing the generated Gherkins—can span up to couple of weeks for a single service and resource pair. This protracted timeline underscores the need for innovative solutions, such as Generative AI, to automate and expedite the creation of security controls, thereby significantly reducing both the time and effort involved in their development and deployment.

1.2 Challenges

While Generative AI holds great promise in automating the generation of Gherkin codes, there are two practical challenges that must be addressed to fully realize its potential.

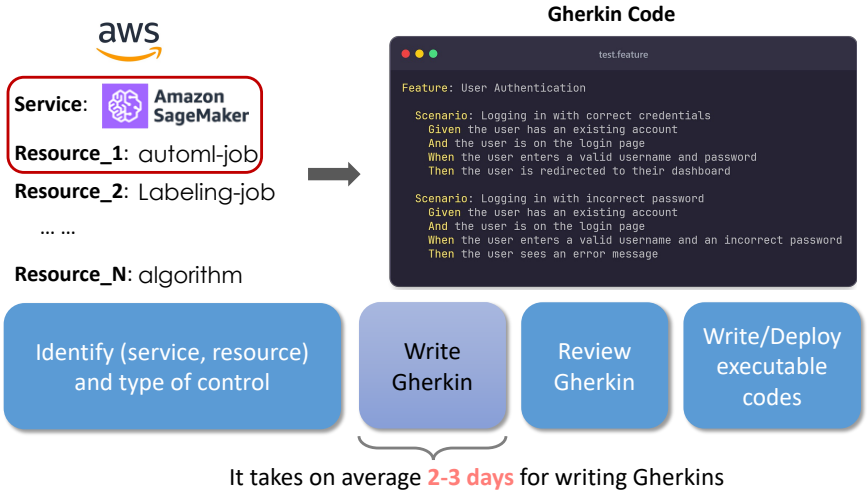

Figure 1: An example of the security control development process, illustrating that the development of writing Gherkin can take on average 2-3 days.

Challenge 1: Accurate Interpretation of Complex Service Documentation. Cloud services are highly diverse, each with its own set of configurations, actions, and security considerations. Given the limited annotated data, the large language model needs to understand the intricacies of these services and translating them into precise and actionable Gherkin specifications without conducting finetuning.

Challenge 2: Ensuring Quality and Reliability of Generated Gherkin Files. Security controls must adhere to standards to ensure they provide effective protection without introducing new vulnerabilities. This necessitates a comprehensive evaluation framework that can systematically assess the generated Gherkins for completeness and compliance with security best practices.

1.3 Contributions

In this work, we propose a novel paradigm to generate Gherkins for facilitating the development of security control. To tackle Challenge 1, our process starts with a comprehensive task description for the LLM, breaking it down into detailed steps to ensure clarity and precision. Next, we provide the model with examples of existing security controls and their associated Gherkins. Finally, we present the LLM with a final query to guide it in generating the desired security control efficiently. By combining a thorough task description, illustrative examples, and a clear query, our approach enables the LLM to produce accurate and effective security controls and Gherkins, thereby reducing the time and effort required from domain experts. To address Challenge 2, we collaborate with domain experts to generate a detailed rubric to evaluate generated Gherkins from multiple dimensions, ranging from 1) whether the generated scenarios are feasible; and 2) whether the description can correctly reflect the security control specified by the scenarios.

2 Related Works

LLM for Structured Output. As Gherkin codes are a type of highly structured output, recent advancements in LLMs [2, 10] for structured output have demonstrated significant progress in several key areas. LLMs are being enhanced through instruction tuning and innovative approaches such as "reflection-tuning," which improves the quality of training data by self-evaluation and enhancement, resulting in better output alignment [8]. Additionally, models like GPT-4 [1] and LLAMA models [13] have shown improvements in generating complex structured data by utilizing structure-aware fine-tuning and FormatCoT (Chain-of-Thought) techniques, significantly reducing formatting errors [12]. Furthermore, novel frameworks such as CRITIC enable LLMs to self-correct by interacting with external tools, enhancing the accuracy and reliability of their outputs [5]. These advancements highlight the potential of LLMs to handle complex, structured output tasks more effectively, paving the way for their application in diverse and sophisticated scenarios.

Few-shot/Zero-shot LLM Prompting. LLMs for zero/few-shot prompting has shown remarkable advancements in enhancing their reasoning and inference capabilities without requiring task-specific training examples. Notably, the introduction of strategies like Plan-and-Solve Prompting and Zero-shot Chain-of-Thought (CoT) prompting has significantly improved LLM performance on complex reasoning tasks by guiding the models to break down problems into smaller, manageable steps [7, 14, 15]. Furthermore, innovations such as UPRISE (Universal Prompt Retrieval for Improving Zero-Shot Evaluation) have improved the generalization and task adaptability of LLMs by retrieving and using relevant prompts across different models and tasks [3]. Additionally, methods like SelfCheck enable LLMs to verify and correct their step-by-step reasoning autonomously, thus enhancing their accuracy and reliability in zero-shot settings [11].

Retrieval-augmented Generation (RAG). Recent advances in RAG have significantly enhanced the capabilities of LLMs. These systems combine retrieval mechanisms with text generation models to improve performance across various tasks, including dialogue generation, machine translation, and question answering [9]. Notable developments include methods like Forward-Looking Active Retrieval Augmented Generation (FLARE), which iteratively retrieves relevant information to enhance text generation accuracy [6]. Furthermore, approaches such as self-memory frameworks

leverage iterative retrieval and generation to create dynamic memory pools that improve content generation [4]. These advancements highlight the ongoing evolution and potential of RAG methods in addressing complex information needs and reducing hallucinations in generated outputs.

3 Gherkin Generation with GenAI

Approach Overview. In this work, we adopt an innovative approach using in-context learning with retrieval-augmented generation to streamline the process. Our approach begins by providing the LLM with a detailed task description, breaking it down step-by-step to ensure clarity and precision. We then supply the model with existing examples of security controls, including their associated Gherkins. Finally, we present the LLM with the final query, guiding it to generate the desired security control efficiently.

3.1 Control Types

In this work, we focus on a set of critical control types identified as essential by subject matter experts (SMEs). These controls are designed to address key security and compliance requirements for cloud environments. The control types include:

- *Encryption of Data at Rest*: Ensuring that all stored data is encrypted to protect it from unauthorized access.
- *Encryption of Data in Transit*: Securing data during transmission to prevent interception and tampering.
- *Tagging*: Implementing consistent and meaningful tags to manage and organize resources effectively.
- *Resources Running on a Supported Version*: Ensuring that resources are running on supported versions to mitigate vulnerabilities associated with outdated software.
- *Backup Enabled*: Guaranteeing that data is regularly backed up to enable recovery in case of data loss or corruption.
- *Multi-AZ Deployment*: Distributing resources across multiple Availability Zones to enhance fault tolerance and availability.
- *Inbound IP Connection Control*: Restricting inbound IP connections to resources to prevent unauthorized access.
- *Resource Accessibility*: Ensuring that resources cannot be accessed by anyone without proper authorization.
- *Audit Logging Enabled*: Enabling audit logging and specifying the destination for logs to facilitate monitoring and compliance.

These control types have been selected based on their importance in securing cloud environments and ensuring compliance with industry standards and best practices. For a detailed description of each control type, please refer to the appendix.

3.2 Detailed Task Description

As seen in the left part of Figure 2, the task description provides the LLM with an initial understanding of the overall objective of the task, which involves designing a detailed and structured prompt to guide the LLM through the generation process. The steps are as follows: 1) *Task Definition*: Clearly define the task to the LLM. In this case, the task is to generate Gherkin code for a specific control type (e.g., logging or version support). Emphasize the role of the LLM as an expert security engineer responsible for generating these controls. 2) *Context Provision*. Provide comprehensive context to the LLM about the importance of the security control. This context helps the LLM understand the rationale behind the security controls it needs to generate. 3) *Expected Output*. Define the structure of the output explicitly. The output should be a JSON formatted message with no additional text. The JSON should contain specific placeholders that the LLM needs to fill with relevant information.

3.3 Step-by-Step Instructions: CoT Reasoning

To ensure the LLM generates accurate and effective Gherkin code for security controls, we adopt a chain-of-thought approach. This approach breaks down the task into detailed, sequential steps, facilitating clear and logical reasoning. The instructions are divided into two main steps: step 1) Service, API Call, and Insecure Configuration Analysis, and step 2) Gherkin Code Generation.

Step 1: Service, API Call, and Insecure Configuration Analysis. First, identify the AWS resource in question, and then identify the relevant API call that provides information about this resource. Next, understand the security best practices related to the control type (e.g., version support and management or encryption of data-at-rest). Finally, formulate a list of checks to verify compliance with security best practices. Each check should be written in a format that the LLM can interpret. These checks will help identify whether a resource is compliant with the specified security standards.

Step 2: Generate Gherkin Code. Based on the analysis from Step 1, the next step is to generate Gherkin code. Gherkin is a language used to write structured, human-readable tests and specifications. This code will help provide a clear and executable structure for security controls. Start by defining the rule components. This includes the Rule Identifier, which is a unique identifier for the rule, and the Rule Name, which is a descriptive name that clearly indicates the rule's purpose. The Description should be a detailed explanation of what the rule checks for and why it is important. The Trigger specifies the condition that initiates the rule, which can be either "Periodic" or "Configuration Changes". If there are any specific parameters required for the rule, these should be defined in the Rule Parameters section. If there are no specific parameters, this section can be left empty.

3.4 In-Context Learning and Retrieval-Augmented Generation

After providing the detailed task description and the chain-of-thought reasoning instruction, our approach further combines the strengths of in-context learning and retrieval-augmented generation to create a robust framework for generating security controls. The process can be summarized as follows:

We design the in-context demonstrations by creating a detailed prompt that embeds examples of existing security controls to guide the LLM. We use public APIs[1] to gather background information about AWS security best practices and integrate this information into the prompt to provide the LLM with the necessary context. We then present the LLM with the final query, guiding it to generate the desired security control based on the provided context and examples. The LLM uses the embedded examples and retrieved information to produce accurate and effective Gherkin code. By leveraging these techniques, we can significantly reduce the time and effort

[1] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

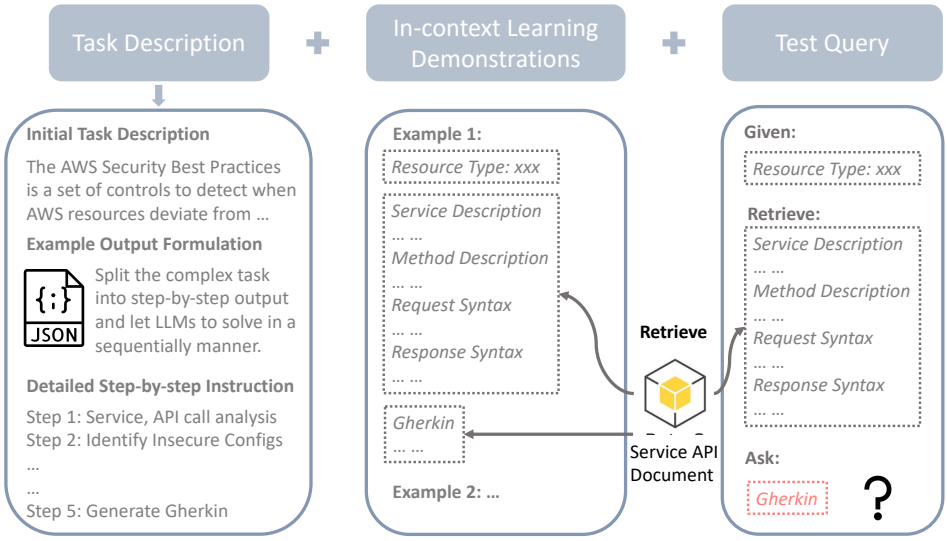

Figure 2: An illustration of the Gherkin Generation framework.

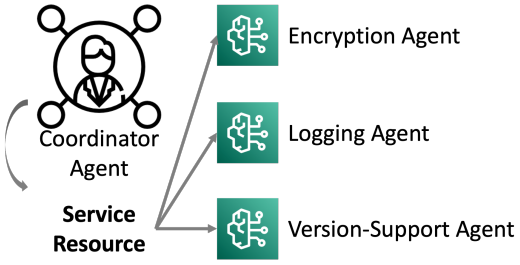

Figure 3: The illustration of agent-based system for automated determination of control types (detailed in Section 3.1) given the service and resource name.

required from security engineers to develop robust security controls. The LLM, equipped with detailed prompts and rich context, can efficiently generate Gherkin code that adheres to best practices in cloud security. This innovative approach not only streamlines the development process but also ensures the implementation of effective and reliable security measures in AWS environments.

3.5 Agent-based Security Control Type Identifier

As mentioned in Section 3.1, security engineers at AWS have identified and categorized nine top-priority security control types for which they are interested in generating Gherkin scripts. These control types have been chosen based on their critical importance in maintaining the security and compliance of cloud environments.

In addition to the primary prompt, we have implemented a feature that allows an LLM agent to determine which security control types are applicable to a given service and resource. This feature addresses a significant challenge faced by security engineers, who often spend considerable time identifying the appropriate control types for specific cloud services and resources. The LLM agent is equipped with detailed descriptions of each Security Control Type, enabling it to make informed decisions about applicability. For instance, the control type "Encryption of Data at Rest" is defined as: "*Data at rest refers to data stored in persistent, non-volatile storage for any duration. Encrypting data at rest helps protect its confidentiality, reducing the risk of unauthorized access. This control checks if the Resource is encrypted at rest. If the Resource is not encrypted at rest, the control will return NON_COMPLIANT. If the Resource is encrypted, it will return COMPLIANT.*"

By leveraging the LLM agent, we can automate the identification and application of relevant security controls, significantly reducing the manual effort required by security engineers. This system enhances the efficiency of the security control development process and ensures that the appropriate measures are in place to protect cloud services and resources.

4 Evaluation System

To ensure the effectiveness and accuracy of Gherkin scripts generated by generative AI, we employ a human-in-the-loop approach for evaluation. This approach leverages expert human judgment to assess the quality of generated Gherkins against a structured rubric. The rubric, shown in Table 1, provides a quantitative framework for evaluating Gherkins based on specific criteria, ensuring a consistent and objective assessment. We provide a more detailed description of each criteria as follows.

4.1 Scenario Evaluation (S)

(1) *(S1) The number of scenarios recorded is correct.* This criterion assesses whether the generated Gherkin includes the appropriate number of scenarios. Each scenario should represent a distinct and necessary test case for the security control.
(2) *(S2) The field specified in the scenario exists.* This checks if all fields referenced in the scenarios are valid and present in the context of the security control being defined. It ensures the relevance and applicability of the scenarios.
(3) *(S3) The resulting compliance status is possible.* This criterion evaluates whether the compliance status derived from the scenario is feasible. It ensures that the scenarios result in legitimate compliance outcomes.

Criteria	Evaluation
S1:	The number of scenarios recorded is correct.
S2:	The field specified in the scenario exists.
S3:	The resulting compliance status is possible.
S4:	The configuration of the resource specified by the scenario is possible.
S5:	The conclusion of the scenario is correct.
R1:	The rule name correctly describes the control specified by the collection of scenarios.
R2:	The description correctly describes the control specified by the collection of scenarios.

Table 1: Gherkin Rubric evaluates the validity of the generated Gherkin code from two dimensions: 1) whether the generated scenarios are feasible; and 2) whether the Rule Identifier and Description can correctly reflect the control specified by the scenarios.

(4) *(S4) The configuration of the resource specified by the scenario is possible.* This ensures that the configuration actions described in the scenarios can actually be implemented within the given cloud environment. It checks for the practicality of the scenarios.
(5) *(S5) The conclusion of the scenario is correct.* This criterion checks if the scenario logically concludes with the correct outcome based on the preceding steps. It verifies the logical flow and correctness of the scenario's outcome.

4.2 Rule Evaluation (R)

(1) (R1) The rule name correctly describes the control specified by the collection of scenarios: This assesses the accuracy and appropriateness of the rule name. The rule name should succinctly and accurately reflect the control described by the scenarios.
(2) (R2) The description correctly describes the control specified by the collection of scenarios: This criterion evaluates the clarity and correctness of the rule description, and whether it provides a clear understanding of the control and its purpose.

4.3 Final Score

The overall score for a Gherkin script is calculated using the following formula:

$$score = (S1 + S2 + S3 + S4 + S5) \times (R1 + R2)/2 \quad (1)$$

This formula integrates the evaluations from both the scenario and rule criteria, and the final score is within the range of $[0, 5]$. The evaluation from both aspects ensures a comprehensive assessment of the Gherkin script's quality. The acceptance threshold is ≥ 2.5 for the generated Gherkins, which indicates security engineers would need light supervision/revision to make the generated Gherkins into production.

5 Experiments

The experiments are conducted in collaboration with domain experts in AWS based on the evaluation metric as identified in Section 4. We use data from all available AWS services and resources. For the use of LLM, we leverage Claude-3-Sonnet hosted on AWS Bedrock[2].

[2]https://aws.amazon.com/bedrock/

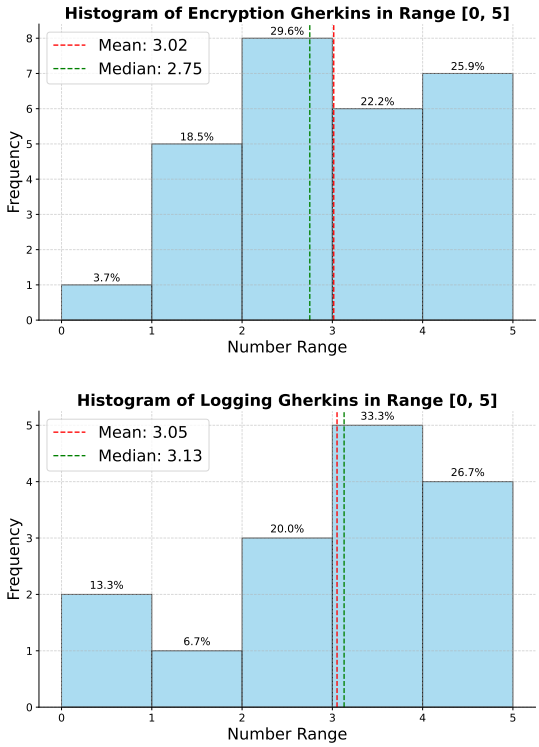

Figure 4: The average score of generated Gherkin for the Encryption of data-at-rest Type and Logging Type.

We first demonstrate two histograms of the evaluated Gherkins by domain experts. As can be seen from Figure 4, both histograms show that the majority of the generated Gherkins for Encryption and Logging types fall within the acceptable range of requiring slight to moderate revisions (score ≥ 2.5). The distribution patterns are similar, with most scores clustering around 3, suggesting that the generation process is relatively consistent in its output for both types. However, the consistency and concentration around the score of 3 indicate that further refinement in the generation process could help in reducing the amount of necessary revision.

We further break down the average score for two categories of security control (i.e., Encryption of data-at-rest and Logging). For each category, security engineers randomly picked ten generated

	Encryption of Data-at-rest	Logging
Scenario Score	4.19	4.07
Rule Score	0.72	0.75
Final Score	3.02	3.05

Table 2: The average score of generated Gherkin Evaluation

Gherkins and reviewed them. The scores are depicted in Table 2. Note that if the Final Score is greater than 4, then the generated Gherkin can be sent to development with little modifications. If the final score is above 2.5, the Gherkin would need moderate to slight amount of revision.

The table shows that the generated Gherkin scenarios for *Encryption of Data-at-Rest* and *Logging* are fairly accurate, with Scenario Scores of 4.19 and 4.07, respectively. However, the Rule Scores are low (0.72 and 0.75), indicating improvements are needed. Finally, the final score calculated by Eq. 1 of both control types are 3.42, and 3.32, respectively.

6 Conclusion

This study presents a novel framework for speeding up the generation of security controls using Generative AI, with a specific focus on producing Gherkin scripts. By incorporating large language models, detailed task descriptions, chain-of-thought reasoning, and retrieval-augmented generation, our approach can speed up the labor-intensive nature of traditional security control development from several days to less than one minute. Moreover, the evaluation results indicate that our method can meet the high standard of security controls evaluated by domain experts, which can be sent to deployment with minor revisions as indicated in Table 2. In summary, this framework significantly reduces the time and effort required from security engineers while maintaining the quality and reliability of the generated controls. This advancement highlights the potential of Generative AI to enhance the development and deployment of security measures in cloud environments, providing improved protection and adaptability against evolving cyber threats.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Guangji Bai, Zheng Chai, Chen Ling, Shiyu Wang, Jiaying Lu, Nan Zhang, Tingwei Shi, Ziyang Yu, Mengdan Zhu, Yifei Zhang, et al. 2024. Beyond efficiency: A systematic survey of resource-efficient large language models. *arXiv preprint arXiv:2401.00625* (2024).

[3] Daixuan Cheng, Shaohan Huang, Junyu Bi, Yuefeng Zhan, Jianfeng Liu, Yujing Wang, Hao Sun, Furu Wei, Denvy Deng, and Qi Zhang. 2023. Uprise: Universal prompt retrieval for improving zero-shot evaluation. *arXiv preprint arXiv:2303.08518* (2023).

[4] Xin Cheng, Di Luo, Xiuying Chen, Lemao Liu, Dongyan Zhao, and Rui Yan. 2024. Lift yourself up: Retrieval-augmented text generation with self-memory. *Advances in Neural Information Processing Systems* 36 (2024).

[5] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738* (2023).

[6] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Active Retrieval Augmented Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 7969–7992.

[7] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[8] Ming Li, Lichang Chen, Jiuhai Chen, Shwai He, Heng Huang, Jiuxiang Gu, and Tianyi Zhou. 2023. Reflection-tuning: Data recycling improves llm instruction-tuning. *arXiv preprint arXiv:2310.11716* (2023).

[9] Chen Ling, Xuchao Zhang, Xujiang Zhao, Yanchi Liu, Wei Cheng, Mika Oishi, Takao Osaki, Katsushi Matsuda, Haifeng Chen, and Liang Zhao. 2023. Open-ended commonsense reasoning with unrestricted answer candidates. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 8035–8047.

[10] Chen Ling, Xujiang Zhao, Jiaying Lu, Chengyuan Deng, Can Zheng, Junxiang Wang, Tanmoy Chowdhury, Yun Li, Hejie Cui, Tianjiao Zhao, et al. 2023. Domain specialization as the key to make large language models disruptive: A comprehensive survey. *arXiv preprint arXiv:2305.18703* 2305 (2023).

[11] Ning Miao, Yee Whye Teh, and Tom Rainforth. 2023. Selfcheck: Using llms to zero-shot check their own step-by-step reasoning. *arXiv preprint arXiv:2308.00436* (2023).

[12] Xiangru Tang, Yiming Zong, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2023. Struc-Bench: Are Large Language Models Really Good at Generating Complex Structured Data? *arXiv preprint arXiv:2309.08963* (2023).

[13] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[14] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.