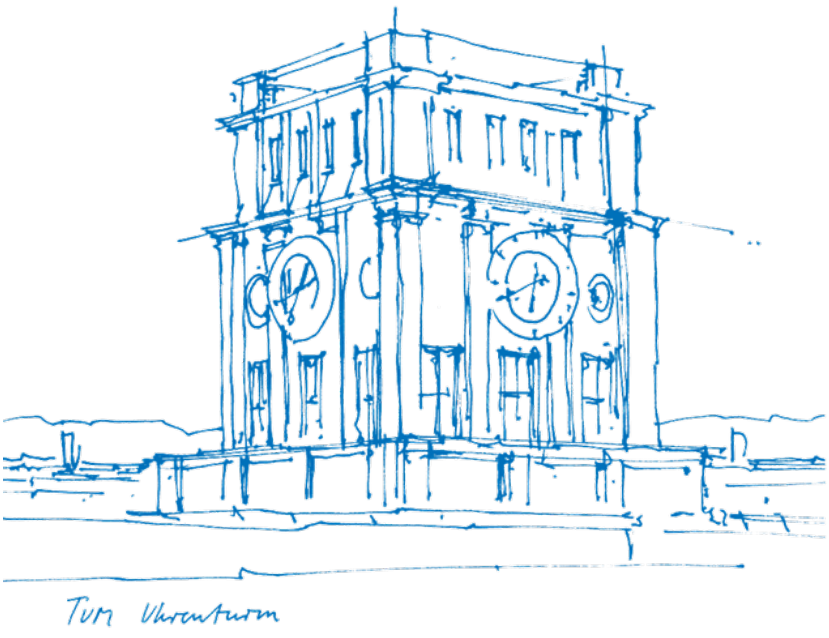
Chair of Information Systems and Business Process Management (i17)
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Information Systems

Furkan Gürbüz

Fine-Tuning Large Language Model with Custom Dataset for Ansible Code Generation



Chair of Information Systems and Business Process Management (i17)
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Information Systems

Furkan Gürbüz

Fine-Tuning Large Language Model with Custom Dataset for Ansible Code Generation

Feinabstimmung eines großen Sprachmodells mit benutzerdefiniertem Datensatz zur Code-Generierung in Ansible

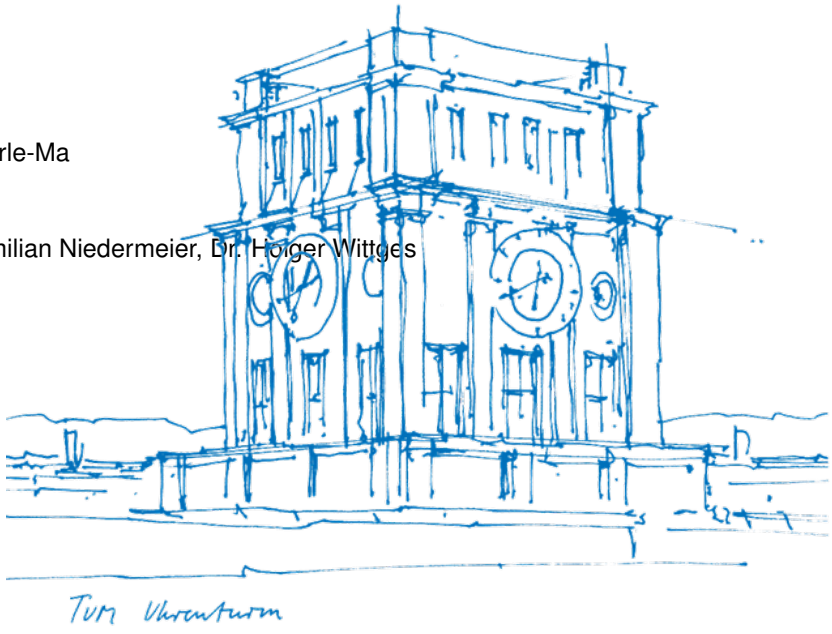
Thesis for the Attainment of the Degree
Master of Science

at the TUM School of Computation, Information and Technology,
Department of Computer Science,
Chair of Information Systems and Business Process Management (i17)

Examiner
Prof. Dr. Stefanie Rinderle-Ma

Supervised by
Thomas Teubner, Maximilian Niedermeier, Dr. Holger Wittges

Submitted on
01.06.2025



Declaration of Academic Integrity

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This thesis was not previously presented to another examination board and has not been published.

Garching, 01.06.2025 Furkan Gürbüz

Abstract

This thesis explores the development and evaluation of a fine-tuned large language model (LLM) tailored for Ansible code generation in SAP environments. The goal is to enhance the automation capabilities within the SAP UCC by generating accurate and context-aware Ansible playbooks. To achieve this, a custom dataset was created using the Ansible Content Parser, focusing on relevant and high-quality YAML code extracted from open-source repositories. Following data preparation and cleaning, the fine-tuning was performed using the Phi-4 model in combination with the Unsloth and LoRA frameworks to ensure efficiency and scalability. The implementation was evaluated across multiple metrics, including ROUGE, METEOR, CHRF, and a custom ansible-lint-based score. The results demonstrate the effectiveness of the approach in producing syntactically correct and semantically meaningful code segments. Furthermore, the thesis includes iterative fine-tuning cycles to enhance model performance and concludes with a discussion on the implications for future research and enterprise applications.

Keywords: *large language model, fine-tuning, Ansible, code generation, infrastructure automation, IaC, YAML, SAP.*

Contents

1 Introduction **7**

1.1 Motivation . . . 7

1.2 Research Questions . . . 8

1.3 Contribution . . . 9

1.4 Methodology . . . 10

1.5 Evaluation . . . 11

1.6 Structure . . . 12

2 Related Work **14**

3 Solution Design **17**

3.1 Distinction of small and large language models . . . 17

3.2 Architectural and Functional Innovations of Phi-4 . . . 19

3.3 Low Rank Adaptation . . . 22

3.4 Evaluation and Performance Metrics . . . 23

ROUGE Score . . . 23

METEOR Score . . . 25

chrF Score . . . 26

Custom Evaluation Metric: Ansible Lint Score . . . 27

4 Implementation **30**

4.1 Dataset creation with ansible content parser . . . 30

Selection of relevant repositories . . . 32

4.2 Data preparation and cleaning . . . 34

Eliminate duplicates . . . 35

Handling missing values . . . 35

Quality assurance . . . 35

Dataset Splitting and Preprocessing . . . 36

Tokenization and Label Preparation . . . 37

4.3 Technology stack and important libraries . . . 38

Unsloth 38
PyTorch Framework 39
CUDA GPU Acceleration 40
Evaluate and NumPy: Calculation libraries 41
4.4 Hardware Resources 42
4.5 Fine-Tuning Process 44
Model Initialization and Configuration 45
Integration of Unsloth and LoRA 46
Training Procedure 48
Evaluation Metrics 51
Checkpointing and Model Saving 53

5 Results **55**

5.1 First Iteration 55
Runtime Training Results 55
Evaluation Metrics 56
5.2 Second Iteration 61
Runtime Training Results 61
Evaluation Metrics 65

6 Discussion **68**

6.1 Comparison of the Results 68
6.2 Future outlook 70

7 Conclusion **71**

A Appendix **73**

A.1 Jupyter Notebook Code 73

List of Tables

1 Performance of Phi-4 on a set of standard benchmarks. 18
2 Training and Evaluation Metrics - First Iteration 56
3 Training and Evaluation Metrics – Second Iteration 65

List of Figures

1 DSR process cycles 12

2 Training time comparision 39

3 Training procedure curve - First Iteration 57

4 Training procedure curve - Second Iteration 62

Introduction

Motivation

In the era of digital transformation, the ability to automate IT operations has become a fundamental driver of organizational efficiency and scalability. Modern enterprises and research institutions are increasingly adopting rapid development cycles that demand frequent updates and enhancements (**gupta2019role**). This continuous evolution requires efficient system management, making the automation of deployments and configurations a critical aspect of DevOps practices (**bass2015devops**).

Furthermore, Enterprise Resource Planning (ERP) systems are crucial to achieve efficiency and scalability by integrating and streamlining essential business functions (**poston2000impact**). Among these, SAP stands out as a leading ERP solution, continuously evolving its capabilities since its inception in Germany (**klaus2000erp**). Educational institutions, such as the SAP University Competence Center at the Technical University of Munich, play a crucial role in providing training and equipping future professionals with the knowledge and skills required to operate ERP systems proficiently (**2024ucc**).

In addition to educating future professionals, the SAP UCC also offers hosting of SAP solutions with robust backup and recovery services. The center utilizes an IT Automation software to install and configure SAP systems (**2024ucc**). The software plays a critical role in automating administrative tasks, such as configuration management and application deployment, through the implementation of playbooks. (**2024ansible**). Manually developing these playbooks is inherently complex and time consuming, requiring a nuanced understanding of the implementation syntax and the specific requirements of SAP systems (**geerling2015ansible**). This complexity poses a substantial challenge for organizations that want to quickly scale operations or fully embrace automation.

Consequently, there is a substantial demand for code generation and support in implementing Ansible playbooks. As it stands, the academic chair does not possess adequate tools for code generation support in Ansible, and existing models do not meet the requirements satisfactorily. By elevating automation capabilities, IT professionals will be well-equipped to optimize operations. This initiative aligns with trends with focus on enhancing operational efficiency, reducing costs, and accelerating deployment timelines.

Research Questions

This thesis will cover multiple steps involved in fine-tuning a large language model. Firstly, we will create a custom dataset required for the fine-tuning process, ensuring consistency and tailoring the model to our specific use case. Subsequently, we will fine-tune the large language model, a process that requires significant computational resources and coding. Finally, we will conduct a testing and evaluation phase to assess the fine-tuned model.

After implementing the fine-tuned Large Language Model, the deployment process of the SAP UCC will be enhanced due to faster implementation of Ansible playbooks facilitated with the optimized model.

Research Question 1: What methodology can be used to gather and prepare a custom dataset for fine-tuning large language models?

Methodology:

In the context of this research question, the focus is on the creation of a custom dataset, which will be used for fine-tuning the large language model. We employ a Question Answering dataset format, utilizing specific tools to gather our dataset. Following this, we will clean and preprocess the data to eliminate noise and ensure consistency in data quality and format. This step is crucial to optimizing the data set for model training. The data needs to be partitioned into different types of set. This will ensure robustness and reliability of the trained model for our use case.

Expected results:

We have created a dataset that will be cleaned and prepared for fine-tuning the large language model.

Research Question 2: What steps are involved in implementing the fine-tuning process for the large language model?

Methodology:

In the context of this research question, the focus is on the fine-tuning process with the created dataset from RQ1. This research question will focus on the implementation and execution of fine-tuning strategies.

Expected results:

We have tuned the large language model, that is suitable for our use case.

Research Question 3: To what extend does the fine-tuned large language model meet the requirements in terms of performance, accuracy, and applicability?

Methodology:

Concerning this research question the focus is on testing the model performance. Consequently, we test our model's performance with professionals by ensuring the usability, accuracy and functionality.

Expected results:

We have tested our model and give insights on the performance of our fine tuned large language model in the context of code generation.

Contribution

The goal of this thesis is to optimize the implementation of Ansible playbooks by using a fine-tuned large-language model. By generating Ansible code specifically tailored for SAP systems, this thesis aims to reduce the time and expertise required for playbook development, making the process more efficient and accessible for both developers and system administrators. This will enable them to focus on higher-level tasks and strategic decision-making. This approach will lead to more consistent, error-free configurations and contribute to the streamlined management of complex IT systems.

In order to achieve this objective, the thesis will involve fine-tuning a large language model specifically for Ansible code generation. The process will begin with the creation of a custom dataset containing Ansible code specifically designed to meet the unique requirements of SAP system configurations. Once the dataset is created, it will undergo preparation and pre-processing to ensure consistency and remove noise. The data set will then be used to train the model, enabling it to generate context-specific high-quality Ansible playbooks (**howard2018universal**). Using this custom data set, the model will learn to produce accurate and efficient Ansible YAML code.

Afterwards, the fine-tuning process follows several key steps, beginning with the selection of a pre-trained model. The model parameters will be fine-tuned by training it on the custom dataset, adjusting the model to generate code that is syntactically correct and functionally effective. Fine-tuning iterations will involve gradually adjusting the hyperparameters and retraining the model to optimize its performance. Upon completion of the fine-tuning process, the model's performance will be tested using a separate validation set to assess its accuracy and effectiveness.

The expected outcomes include not only time and cost savings, but also the ability to scale automation across more complex IT environments. Through the integration of advanced machine learning techniques, this research will contribute to the ongoing evolution of IT automation practices.

Methodology

Design Science Research (DSR) is a well-established methodology in the field of information systems researchIts main focus is to create and evaluate practical solutions—called artifacts—that address real-world problems. First introduced by Hevner et al., DSR highlights the importance of not only building innovative solutions but also thoroughly testing them to bridge the gap between theory and practice (**hevner2010design**). The key Components of Design Science Research are Stakeholders and Artifacts. Stakeholders within DSR are divided into different roles:

- **Researchers:** The individuals who conduct the research and implement the artifacts.
- **Practitioners and Users:** The end-users and organizations that will benefit from using the artifacts.
- **Reviewers and Evaluators:** The experts who evaluate the functionality and impact of the artifacts.
- **Sponsors and Funding Bodies:** The organizations or institutions providing financial support for the research initiative.

Artifacts in DSR - as defined by Hevner et al. - include constructs, models, methods, and instantiations, all serving as outputs of the research process that are both innovative and rigorously evaluated (**hevner2010design**). Hevner et al. outline a systematic six-step process to conduct DSR (**hevner2010design**), which ensures both scientific rigor and practical relevance:

1. **Problem Identification and Motivation:** Clearly describe the research problem and justify its importance. Provide a detailed analysis to establish the relevance and urgency of the problem (**hevner2010design**.
2. **Define Objectives of a Solution:** Establish the criteria and benchmarks for an effective solution. Clearly define what constitutes success for the artifact being developed and how it differs or improves existing solutions (**hevner2010design**.

3. **Design and Development:** Develop the artifact using theoretical insights and design principles. Specify the components, functionalities and intended applications of the artifact (**hevner2010design**.
4. **Demonstration:** Apply the artifact in a controlled or real-world setting to prove its practical utility. Provide empirical evidence that the artifact can effectively address the identified problem (**hevner2010design**.
5. **Evaluation:** Assess the performance of the artifact against predefined objectives. Use qualitative and quantitative measures to assess the utility, efficacy, and quality of the artifact, gathering feedback from key stakeholders (**hevner2010design**.
6. **Communication:** Document the research process, findings, and implications. Disseminate results to both academic and practitioner communities to highlight contributions to theory and practical applications (**hevner2010design**).

In this thesis, Hevner's DSR principles will guide the research design. This includes structuring the problem definition and solution design process according to the DSR steps. Additionally, emphasizing the development and preliminary evaluation of a solution artifact, and involving stakeholders to validate the research outputs and ensure practical relevance. By integrating elements of Hevner's DSR, the research aims to maintain a structured, rigorous approach that enhances the significance, relevance, and applicability of the implemented solutions.

In our use case, stakeholders are defined as follows. The researchers consist of the thesis writer and the supervisor. The users include developers and system managers at the SAP UCC Chair. The reviewers and evaluators, a subgroup of the users, will be responsible for assessing the artifact. The funding body supporting this research is the Technical University of Munich.

The primary artifact of this research is the fine-tuned large language model designed to generate improved Ansible code. The artifact aims to enhance the quality, efficiency, and accuracy of code generation to support developers and system managers in the SAP UCC Chair.

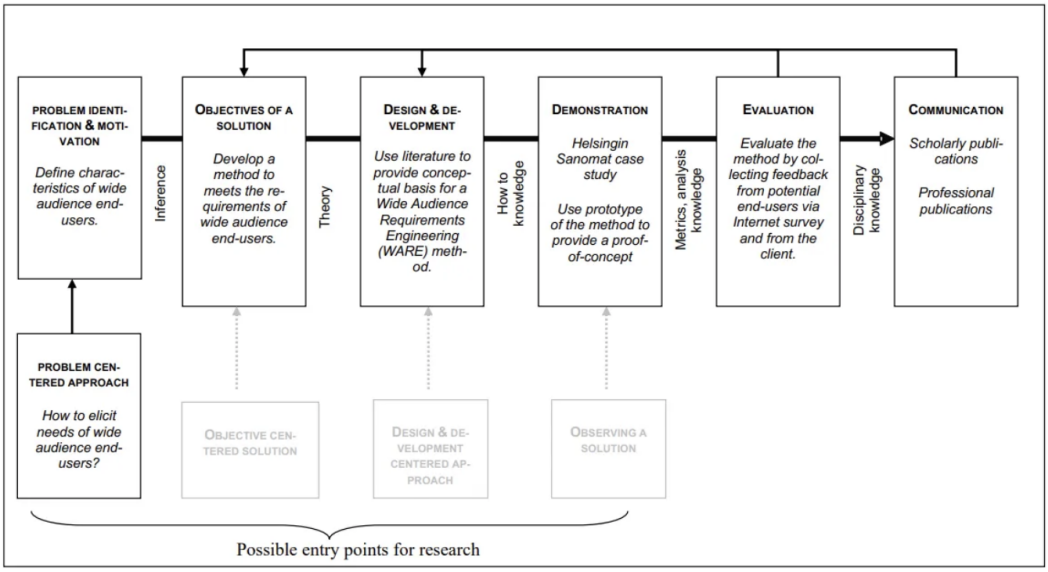
Evaluation

The evaluation will be conducted through multiple iterative cycles, based on the predefined DSR methodology as illustrated in Figure 1, where each iteration consists of two phases. One phase includes the model fine-tuning process with adjusted training configurations and training data. The

second phase consists of the evaluation of the performance by predefined metrics which focus on the model's accuracy in generating code (functional correctness) and syntactical correctness.

After each iteration, the results will be evaluated to see whether the adjustments applied will positively affect the outcome. Once a satisfied performance level is reached or a further iteration seems redundant.

Figure 1
DSR process cycles



Overview of the iterative phases in DSR process (**peffers2007design**).

Structure

This thesis will be structured into seven main chapters. First, an introduction into the conducting research will be given. Afterwards, we will discuss related work based on custom dataset creation and fine-tuning a large language model. Additionally, we will cover the solution design with focus on dataset creation and fine-tuning strategy. The implementation chapter will describe a detailed overview about tools, frameworks and programming languages needed for the development of the solution. Hereby, encountered challenges and problems will also be covered. The results chapter will explain the experiment to evaluate the tuned model and additionally justify the metrics that are used for testing. Furthermore, a discussion chapter will interpret the results and link them to the research questions stated in the introduction. Hereby, implications of the results will be discussed for future research. Lastly, a conclusion with a summary of the key findings and research contributions

will be presented. With this structure, the thesis aims to provide a clear and logical flow, guiding the reader from the initial research motivation and background, through the technical implementation and evaluation, to the final discussion and conclusion, ensuring that all relevant aspects of the research process are covered in a transparent and comprehensible way.

Related Work

Large language models (LLMs) have a significant impact on the field of computer science, introducing new possibilities across various domains. LLMs decreased the complexity in automation and improved workflows. In recent years, multiple studies show that LLMs create value in technical workflows by reducing the time spent on manual tasks (**brown2020language**; **radford2018improving**). The increased popularity of these models is because of their flexibility. LLMs can generate code, they can assist developers by automating repetitive tasks, documenting workflows, and providing context-aware explanations for complex code fragments, thereby supporting both novice and experienced developers.

Hereby, it is important to discuss the advantages and challenges within the integration of LLMs into workflows. On the one hand, research has shown that LLMs can accelerate development cycles. enhance code quality, and minimize common errors (**chen2021evaluating**). On the other hand, ensuring that the generated code is interpretable and trustworthy remains a crucial issue. Furthermore, data privacy concerns arise when proprietary or sensitive codebases are involved, and the need to fine-tune models for specialized domains introduces additional complexity (**bender2021dangers**).

Within the context of this research, LLMs are particularly promising for supporting the development and maintenance of Ansible playbooks in SAP environments. By fine-tuning a dedicated model, such as Phi-4, to the specific requirements and conventions of SAP infrastructure automation, the model can assist developers in multiple ways. For example, it can generate playbook templates, suggest improvements to existing code, automate routine configuration tasks, and provide explanatory comments for non-trivial sections of automation logic. These capabilities not only enhance productivity but also contribute to knowledge transfer, making it easier for new team members to understand and contribute to the automation process. Additionally, since the fine-tuning process incorporates domain-specific best practices, the model's recommendations are better aligned with the operational and technical constraints of SAP systems, ultimately improving the reliability and maintainability of the resulting playbooks (**hevner2010design**; **vaswani2017attention**).

The use of LLMs to automate IT tasks has seen significant growth in recent years. These models have the potential to greatly enhance developer productivity, especially when applied in domain-specific

contexts. One of the prominent projects in this area is Ansible Lightspeed, developed by Sahoo et al. (**sahoo2024ansible**). Ansible Lightspeed is a generative AI service specifically designed to generate Ansible YAML code. This service utilizes the IBM Watsonx Code Assistant for Red Hat Ansible (WCA-Ansible), a transformer-based decoder with 350 million parameters. The model was trained from the ground up on a diverse set of natural language, source code, and Ansible-specific data. By providing code recommendations based on natural language prompts, Ansible Lightspeed supports developers in streamlining their IT automation tasks. The effectiveness of the model was evaluated through interaction data from more than 10,000 users, achieving a notable 13.66 percent N-day user retention rate on day 30 (**sahoo2024ansible**). These figures demonstrate that a specialized model can achieve high acceptance rates among its users when deployed effectively.

In contrast to Ansible Lightspeed, our model is specifically tailored to automation tasks within SAP environments. While Ansible Lightspeed relies on general Ansible data, our model is trained on a dataset specifically curated for SAP-related tasks. This SAP-focused dataset is created using the Ansible Content Parser, a tool that meticulously extracts relevant content from existing SAP sources. Using this specialized data set, we anticipate that our model will offer more detailed and context-specific code recommendations that are perfectly suited to SAP environments.

There are several key differences between Ansible Lightspeed and our fine-tuned model within this research. First, the domain focus sets them apart. Ansible Lightspeed is designed for general IT automation tasks using Ansible, whereas our model is specifically targeted at automating SAP environments. This specialization ensures that our model can address the unique requirements and complexities associated with SAP systems. Secondly, the datasets used to train the models differ significantly. Ansible Lightspeed is trained on generic Ansible data, encompassing source code and natural language inputs from a wide range of domains. In contrast, our model leverages a custom dataset specifically designed for SAP environments. This allows us to create a dataset that compensates the unique requirements of SAP environments. Our model can provide code recommendations that are more accurate and more relevant by concentrating on this customized data, guaranteeing that they precisely match the requirements of SAP systems. Third, each model has a distinct user base. A flexible strategy that can be adjusted to different use cases is required because Ansible Lightspeed is made to accommodate a broad spectrum of developers that use Ansible for a variety of IT automation jobs. On the other hand, our strategy is designed for a niche market: customers that are particularly interested in using Ansible playbooks

to automate SAP systems. Finally, the training methodologies and the models themselves differ significantly. Ansible Lightspeed relies on the IBM Watsonx Code Assistant, which is designed for general-purpose code generation and leverages a broad dataset to ensure flexibility across various IT automation contexts. In contrast, our approach utilizes the Phi-4 model, which is fine-tuned specifically for SAP environments using a custom dataset. Existing code completion systems such as GitHub Copilot (**2025github**), Tab9 (**2025tab9**), Replit (**2025replit**), and Amazon Code Whisperer (**2025aws**) represent more generic approaches to source code generation. These tools support a variety of programming languages and tasks, demonstrating the versatility of LLMs in software development. However, studies have shown that specialized models, like our SAP-focused model and Ansible Lightspeed, can achieve higher acceptance rates and greater user satisfaction by tailoring optimizations to specific domains (**ziegler2022productivityassessmentneuralcode**; **pujar2023automatedcodegenerationinformation**).

We believe that our fine-tuned model, specifically focused on SAP scenarios, will deliver comparable or even superior results relative to existing generic models. By precisely addressing the needs of developers working in SAP environments, our model has the potential to significantly enhance their efficiency and productivity. Ultimately, this could set a new benchmark for future developments in domain-specific AI-driven code generation tools.

Solution Design

This chapter outlines the architectural and conceptual design choices that guided the development of the fine-tuning pipeline and model integration for this thesis. It begins by differentiating between small and large language models, contextualizing the selection of Phi-4 as a high-performance, resource-efficient LLM. The chapter then explores key architectural features of Phi-4, its training innovations, and its relevance to the problem domain. Furthermore, it introduces the principles and benefits of Low-Rank Adaptation (LoRA), which serves as the foundation for the fine-tuning strategy employed in this work.

Distinction of small and large language models

Artificial intelligence now relies heavily on language models, especially in natural language processing (NLP). The number of parameters, computational complexity, and data efficiency set Large Language Models (LLMs) apart. LLMs show gains in thinking, problem-solving, and generalization across a range of tasks as they grow. Even though models with more than 100 billion parameters, like GPT-4 and PaLM-2, have historically been categorized as "large," new developments in training techniques and data efficiency indicate that models like Phi-4 (14B) may be able to perform on par with much larger models, redefining the bar for LLM classification (**abdin2024phi4technicalreport**; **hoffmann2022training**).

A Large Language Model is typically characterized by the following three key factors:

1. the number of parameters – The number of trainable weights in the model, historically considered the defining metric of largeness
2. the scale and quality of its training data – The breadth and curation of the dataset used for pretraining significantly impact generalization and performance.
3. its capacity for cross-task generalization - The ability of the model to function effectively on zero-shot, few-shot, and fine-tuning tasks with little modification

Historically, models with tens or hundreds of billions of parameters have been classified as LLMs, assuming that increased scale directly correlates with superior performance (**kaplan2020scaling**).

However, recent research suggests that data quality and training methodology can enable smaller models to outperform larger counterparts (**hoffmann2022training**).

Phi-4, a 14-billion-parameter model, exemplifies this shift. While previous Phi models relied primarily on distillation from GPT-4, Phi-4 surpasses its teacher in reasoning-heavy benchmarks such as graduate-level problem-solving (GPQA) and mathematical reasoning (MATH) (**abdin2024phi4technicalreport**). Its architecture remains similar to its predecessors, but improvements in synthetic data generation, post-training refinement, and training curriculum optimization allow it to compete with much larger LLMs, including LLaMA-3-70B and GPT-4o-mini in select benchmarks (**2025micPhi4**).

Traditionally, the distinction between small and large language models has been parameter-centric, with models under 10 billion parameters considered "small" or "medium-sized" (**gunasekar2023textbooks**). However, scaling laws indicate that data efficiency and architectural optimizations allow for "small" models by parameter count to function as large models in performance (**hoffmann2022training**).

Table 1
Performance of Phi-4 on a set of standard benchmarks.

Benchmarks	**Models**						
	Phi-4 14B	**Phi-3** 14B	**Qwen 2.5** 14B instruct	**GPT** 4o-mini	**LLaMA-3.3** 70B instruct	**Qwen 2.5** 72B instruct	**GPT** 4o
MMLU	84.8	77.9	79.9	81.8	86.3	85.3	**88.1**
GPQA	**56.1**	31.2	42.9	40.9	49.0	50.6	50.6
MATH	**80.4**	44.6	75.6	73.0	66.3	74.6	74.6
HumanEval	82.6	67.8	72.1	86.2	78.9	87.1	**90.6**
MGSM	80.6	63.9	77.9	86.5	89.1	82.8	**90.4**
SimpleQA	3.0	7.6	7.6	39.4	9.3	8.6	9.3
DROP	75.5	58.3	59.7	79.9	82.4	80.9	**85.6**
MMLUPro	70.4	51.3	63.2	63.4	69.6	69.6	**73.0**
HumanEval+	82.8	69.2	79.1	82.4	77.8	84.0	**88.0**
ArenaHard	75.4	67.0	68.3	73.1	76.4	79.2	**85.6**
LiveBench	47.6	28.1	49.8	58.7	57.1	64.6	**72.4**
IFEval	63.0	57.9	78.7	78.7	**89.3**	85.6	84.8
PhiBench (internal)	56.2	43.9	49.8	58.7	57.1	64.6	**72.4**

Data sourced from (**abdin2024phi4technicalreport**). Best scores for each benchmark are highlighted in **bold**.

These results highlight that Phi-4 competes with or surpasses larger models, challenging the traditional parameter-based classification of LLMs. This performance is attributed to its advanced data curation, synthetic augmentation techniques, and novel post-training refinements, such as Di-

rect Preference Optimization (DPO) and Pivotal Token Search (PTS), which improve its reasoning abilities and factual accuracy (**2025micPhi4**).

The emergence of efficiently trained models like Phi-4 calls for a reconsideration of what defines a large language model. While previous generations of LLMs emphasized parameter count as the primary metric, scaling laws suggest that training efficiency and data optimization can achieve comparable results at a fraction of the size (**hoffmann2022training**).

Phi-4's reliance on high-quality synthetic and curated datasets allows it to outperform models with much larger parameter counts, challenging the notion that size alone dictates performance. Post-Training techniques like Pivotal Token Search (PTS) enhance its ability to generate more accurate and reliable responses, narrowing the performance gap between Phi-4 and models exceeding 70B parameters (**2025micPhi4**). Unlike trillion-parameter models that require massive computational infrastructure, Phi-4 achieves LLM-level performance at a fraction of the cost, making it more accessible for real-world applications (**abdin2024phi4technicalreport**).

In Conclusion traditional definitions of Large Language Models have been parameter-driven, models like Phi-4 challenge this paradigm by demonstrating LLM-level performance at a reduced scale. By leveraging data efficiency, architectural refinements, and synthetic data augmentation, Phi-4 competes with models 5× its size, suggesting that the classification of LLMs should now consider both parameter count and efficiency metrics. As AI research progresses, the emphasis will likely shift toward models that optimize performance while balancing computational feasibility, ensuring broader accessibility and responsible AI deployment.

Architectural and Functional Innovations of Phi-4

The Phi-4 model continues Microsoft's efforts to enhance natural language understanding and generation. Previous models like Turing-NLG and GPT-3 set benchmarks in terms of scale and capability, but Phi-4 aims to push these boundaries further by incorporating advanced learning techniques and broader datasets focusing on data quality (**abdin2024phi4technicalreport**). Unlike its predecessors, which relied primarily on organic data, Phi-4 strategically incorporates synthetic data throughout the training process, leading to substantial improvements in performance, particularly in reasoning and problem solving tasks.

Phi-4 is a 14 billion parameter transformer-based LLM, developed with a specific focus on data quality (**2025micPhi4**). This substantial increase in parameters allows Phi-4 to capture and generate more nuanced and contextually accurate text. The model was trained using a mixed precision training approach, optimizing computational efficiency while maintaining high accuracy and stability. Advanced hardware optimizations, including the use of GPUs and TPUs, support large-scale computations. Two significant innovations that make this possible are the Zero Redundancy Optimizer (ZeRO) and Megatron-LM, which contribute to the scalability and cost-efficiency of the training process. ZeRO, introduced as part of the DeepSpeed library, addresses the memory limitations of large-scale training by partitioning model states (such as optimizer states, gradients, and parameters) across multiple devices, thereby reducing redundancy and enabling efficient memory usage (**rajbhandari2020zero**). This partitioning happens in progressive stages (ZeRO-1, ZeRO-2, and ZeRO-3), allowing flexibility based on the scale of the model and the available hardware resources (**ren2021zero**). Complementary to ZeRO, Megatron-LM implements tensor and pipeline model parallelism to further distribute the computation across GPUs. Instead of assigning the full model to each GPU, different layers or subcomponents of the transformer are divided among devices, allowing for simultaneous forward and backward passes across large clusters (**shoeybi2019megatron**). Together, these innovations enable the training of models like Phi-4, which, despite having 14 billion parameters, achieves LLM-level performance at a fraction of the computational cost typically required by trillion-parameter models (**abdin2024phi4technicalreport**). This cost-performance balance is essential for applying models like Phi-4 in real-world enterprise settings, such as the automation of SAP infrastructure. The architecture of Phi-4 follows a decoder-only transformer with a default context length of 4096 tokens, extended to a 16K context length during midtraining (**2025micPhi4**). The training process involved approximately 10 trillion tokens, utilizing linear warm-up and decay schedules with a peak learning rate of 0.0003 and a global batch size of 5760. Additionally, the model employs techniques like rejection sampling and Direct Preference Optimization (DPO) during post-training, refining its outputs to achieve state-of-the-art performance.

Phi-4 demonstrates significant advancements in several key areas, which will be described in the following paragraph. One of the most important advantage is the models efficiency to performance ratio. Natural Language Understanding capabilities (**abdin2024phi4technicalreport**). Additionally, Phi-4 generates coherent, context-sensitive text that closely mimics human writing, making it particularly useful for content creation, customer service automation, and interactive AI applica-

tions. This aspect of Phi-4's capabilities highlights its strength in Natural Language Generation (**abdin2024phi4technicalreport**). The model's ability to learn from limited examples has been enhanced, enabling it to adapt quickly to new tasks with minimal additional training data, demonstrating its flexibility and efficiency in few-shot learning scenarios (**abdin2024phi4technicalreport**). This can be very useful for our use case because of the limited size of the dataset. Furthermore, by building on its diverse training set, Phi-4 offers robust multilingual support, including less commonly spoken languages, thus broadening its applicability globally and enhancing its utility in a variety of linguistic contexts (**abdin2024phi4technicalreport**). Leveraging its training on technical and programming data, Phi-4 can understand, generate, and even debug code snippets across various programming languages, highlighting its proficiency in code comprehension and generation (**abdin2024phi4technicalreport**). Synthetic data plays a pivotal role in Phi-4's training regime. High-quality synthetic datasets are designed to prioritize reasoning and problem-solving and are meticulously generated using techniques like multi-agent prompting, self-revision workflows, and instruction reversal. By addressing some of the shortcomings of conventional unsupervised datasets, these techniques make it possible to create datasets that improve the model's capacity for reasoning and problem-solving.

The features of Phi-4 enable a wide range of applications in many industries. By producing comprehensive and context-specific automation scripts, Phi-4 can greatly simplify activities, lessen developer workload, and increase accuracy and consistency in SAP environments, where automating intricate and crucial processes is crucial. In addition to being the result of recent advancements, Phi-4 serves as a basis for upcoming developments. Its efficiency, scalability, and capacity to adjust to domain-specific requirements are the goals of ongoing research. To further improve the model's resilience and protect data privacy, methods like federated learning and continuous training on decentralized data are being investigated. Microsoft's dedication to integrating cutting-edge AI capabilities throughout its product line is demonstrated by integration into platforms such as Visual Studio and Azure, which make these potent tools available to developers and businesses globally. In conclusion, Microsoft's Phi-4 model pushes the limits of natural language creation and processing, marking a substantial breakthrough in AI technology. It serves as a foundation for upcoming AI-driven advancements due to its scalable architecture, broad range of capabilities, and varied application potential. The knowledge and technologies underpinning Phi-4 will be used as

a benchmark as we further explore the solution architecture for this thesis, helping us create our refined, domain-specific models.

Low Rank Adaptation

Fine-tuning LLMs presents significant computational challenges, especially when operating under limited hardware resources. To address this issue, the technique of Low-Rank Adaptation (LoRA) was introduced. LoRA allows for efficient fine-tuning by injecting trainable low-rank matrices into existing layers of a pretrained model, thereby substantially reducing the number of parameters that need to be updated. This method was first proposed by Hu et al. (**hu2022lora**) and has since gained widespread adoption in the LLM community. The central idea of LoRA is based on the empirical observation that the weight updates required during fine-tuning often lie in a low-dimensional subspace. Instead of updating the original weight matrix $W \in \mathbb{R}^{d \times k}$ directly, LoRA introduces two smaller trainable matrices $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$ such that the weight update ΔW is defined as:

$$\Delta W = A \cdot B$$

Here, $r \ll \min(d, k)$ denotes the *rank* of the update, which is typically a small integer (e.g., $r = 4$ or $r = 8$). This design reduces the number of trainable parameters from $O(dk)$ to $O(r(d + k))$, allowing for significantly more efficient training while preserving model performance (**hu2022lora**). LoRA provides several advantages:

- **Memory efficiency:** Only the low-rank matrices are updated and stored, reducing memory consumption.
- **Modularity:** LoRA layers can be added or removed without altering the underlying pretrained model.
- **Performance retention:** In many cases, LoRA achieves performance comparable to full fine-tuning with a fraction of the computational cost (**dettmers2023qlora**).

In this thesis, LoRA is utilized via the `Unsloth` library, which provides an optimized framework for loading and fine-tuning LLMs. Unsloth supports efficient LoRA integration by enabling 4-bit quantized models to be fine-tuned using LoRA layers—further minimizing GPU memory requirements. The utilization of LoRa will be showcased with a code snippet in the Implementation chapter.

Evaluation and Performance Metrics

In this section the evaluation process and metrics will be defined, and implemented. The important point within this thesis was to utilize metrics, that suit our use case of code generation. Hereby code generation in llm's can be defined as text generation tasks, which have the mostly a common methodology and metrics defined for this task.

ROUGE Score

In the context of this thesis, which focuses on the generation of Ansible automation code using large language models, it is essential to apply metrics that can effectively evaluate the similarity between generated and reference YAML files. Given the syntactic sensitivity of configuration management languages like YAML and the importance of maintaining structural integrity in Ansible playbooks, the selection of evaluation metrics must account for both lexical overlap and partial structural similarity. One metric commonly employed for this purpose in text generation tasks is ROUGE—Recall-Oriented Understudy for Gisting Evaluation (**lin2004rouge**). Although originally developed to assess text summarization, ROUGE has been adapted here to measure the degree of overlap between generated Ansible code snippets and their corresponding reference implementations. The adoption of ROUGE in code generation tasks is supported by prior work, which has highlighted the value of n-gram and subsequence matching for evaluating the quality of structured text output (**gkatzia2015snapshot**). The ROUGE metric is implemented in several variants, each designed to capture different aspects of textual similarity:

- **ROUGE-N** variant measures the overlap of contiguous n-grams between the generated and reference sequences. ROUGE-1 refers to unigram overlap, while ROUGE-2 extends the comparison to bigrams. These variants are especially relevant for YAML code, where the exact sequence of keys and values is often significant (**lin2004rouge**).
- **ROUGE-L** In contrast to ROUGE-N, which focuses on contiguous n-gram matches, ROUGE-L evaluates the Longest Common Subsequence (LCS) between two sequences. This is particularly useful when syntactic variations occur without altering the semantic intent of the code. ROUGE-L thus supports the evaluation of partially correct output while preserving the importance of sequential order (**lin2004rouge**).
- **ROUGE-L-Sum** This extension of ROUGE-L accounts for multiple reference sentences or code blocks, aggregating the LCS-based F-scores across all pairs of candidate and reference

sequences (**lhoest2021datasets**). It is particularly beneficial in scenarios where a generated playbook consists of distinct functional blocks, such as task definitions, variables, and handlers, that must be evaluated holistically.

The mathematical formulation of ROUGE-N is defined as follows:

$$\text{ROUGE-N} = \frac{\sum_{S \in \{\text{Reference}\}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{S \in \{\text{Reference}\}} \sum_{\text{gram}_n \in S} \text{Count}(\text{gram}_n)} \tag{3.1}$$

Here, $\text{Count}_{\text{match}}(\text{gram}_n)$ denotes the number of n-grams shared between the generated output and the reference sequence (**lin2004rouge**). Within this thesis, the evaluation is limited to ROUGE-1 and ROUGE-2, as higher-order n-grams $n > 2$ have been shown to contribute less significantly to performance evaluation in code generation (**abs-2104-02443**).

For ROUGE-L-Sum, the calculation involves computing precision and recall for each reference sentence (**lhoest2021datasets**):

$$\text{ROUGE-L-Sum}_{\text{recall}} = \frac{\sum_{i=1}^{m} \text{LCS}(r_i, c)}{\sum_{i=1}^{m} |r_i|} \tag{3.2}$$

$$\text{ROUGE-L-Sum}_{\text{precision}} = \frac{\sum_{i=1}^{m} \text{LCS}(r_i, c)}{|c| \times m} \tag{3.3}$$

$$\text{ROUGE-L-Sum}_{\text{F-score}} = \frac{(1 + \beta^2) \times \text{precision} \times \text{recall}}{\text{recall} + \beta^2 \times \text{precision}} \tag{3.4}$$

Where r_i represents the i-th sentence in the reference text, c is the candidate text, m is the number of sentences in the reference, and β is typically set to 1.2, giving slightly more weight to recall than precision (**lin2004rouge**; **ng2015better**).

The implementation of ROUGE into this thesis provides a systematic, reproducible method for assessing the lexical and structural fidelity of model-generated Ansible code. Although ROUGE was not designed specifically for programming languages, its sensitivity to surface-level similarity and sequence alignment makes it a suitable proxy for evaluating YAML-based outputs when paired

with complementary evaluation strategies such as syntax checking or functional validation via tools like Ansible-lint, which will be explained in the following sections.

METEOR Score

In addition to the ROUGE metric, this thesis utilizes the METEOR (Metric for Evaluation of Translation with Explicit Ordering) score as part of the evaluation framework for generated Ansible YAML code. Originally developed for machine translation tasks, METEOR was introduced as an alternative to BLEU and ROUGE, aiming to provide more linguistically informed and semantically sensitive evaluations (**banerjee2005meteor**). Unlike ROUGE, which focuses primarily on n-gram overlap, METEOR incorporates several additional components into its evaluation process:

- **Exact Match**: Direct word-to-word matches between generated and reference text.
- **Stemming**: Lexical variation is accounted for by reducing words to their root forms.
- **Synonymy**: WordNet-based synonym detection enables alignment beyond strict string identity.
- **Word Order**: A fragmentation penalty is introduced based on how aligned segments are ordered.

The METEOR score is particularly valuable in the context of Ansible code generation for several reasons. First, while YAML syntax is relatively rigid, many constructs can be phrased in slightly different ways without altering their functionality. For instance, task names, comments, or descriptive fields can vary lexically while preserving semantic intent. METEOR's ability to align similar but non-identical tokens (e.g., via stemming or synonym matching) helps capture this nuance, which would be penalized under more rigid n-gram-based metrics.

Moreover, METEOR includes an alignment-based calculation that maps unigrams from the generated output to those in the reference. The final score is computed based on a harmonic mean of unigram precision and recall, further adjusted with a penalty term that captures word order dissimilarities:

$$\text{METEOR} = F_{mean} \cdot (1 - Penalty) \tag{3.5}$$

where:

$$F_{mean} = \frac{10 \cdot P \cdot R}{R + 9P} \tag{3.6}$$

and P and R denote unigram precision and recall respectively. The *Penalty* is computed based on the number of matched chunks and their fragmentation:

$$Penalty = \gamma \cdot \left(\frac{ch}{m}\right)^{\theta} \tag{3.7}$$

Here, ch is the number of chunks, m is the number of matches, and γ and θ are tunable parameters. For our use case we used the standard METEOR implementation, $\gamma = 0.5$ and $\theta = 3$ (**banerjee2005meteor**).

In this thesis, the METEOR score complements ROUGE by providing a more flexible match criterion that reflects real-world variability in Ansible code descriptions. However, it is important to note that while METEOR is effective for evaluating natural language components within YAML, it does not consider YAML schema correctness. Thus, syntactic validity is ensured through supplementary evaluation using `ansible-lint`.

The integration of METEOR into the evaluation pipeline serves to mitigate the limitations of exact-match-based metrics, particularly in scenarios where lexical diversity is tolerated or even encouraged. This is especially relevant in Ansible playbooks where sections such as task names or comments may vary across functionally equivalent implementations.

chrF Score

To further complement the evaluation framework presented in this thesis, the chrF metric is employed as an additional method to assess the quality of generated Ansible code. The chrF (character n-gram F-score) metric, proposed by Popović (**popovic2015chrf**), is specifically designed to evaluate the similarity between reference and candidate sequences at the character level. The utilized metrics ROUGE and METEOR measure the generated text at the token or word level, whereas chrF allows detecting cases where the generated text contains minor character-level deviations that would otherwise lead to a zero match at the token level. This characteristic makes chrF especially relevant in Ansible playbook generation, where YAML syntax is highly sensitive to formatting,

whitespace, and indentation. A single character can result in an invalid or non-functional playbook. Therefore, character-level similarity can serve as a fine-grained measure for syntactic correctness. This is particularly important when evaluating the output of large language models, which may vary lexically but still preserve the intended structure or semantics of the code. The chrF score is based on the F_β measure computed over character-level n-gram precision and recall. The default configuration, denoted as chrF++, incorporates both character n-grams (usually with n ranging from 1 to 6) and word n-grams for added robustness. In this thesis, the simpler chrF formulation is used, defined as:

$$\text{chrF}_\beta = \frac{(1 + \beta^2) \cdot \text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \tag{3.8}$$

The chrF metric calculates an F_β score, which is based on the harmonic mean of character-level n-gram precision and recall. Hereby, precision is defined as the ratio of correctly matched character n-grams between the generated output and the total number of character n-grams in the generated sequence. Recall describes the proportion of matched character n-grams in relation to the total number of character n-grams in the reference sequence. The weighting parameter β allows for adjusting the influence between recall and precision. Following standard practices, a value of $\beta = 2$ is used within this thesis, which leads to a stronger weighting of recall. This weighting strategy supports the objective of achieving a high degree of structural coverage in the generated output. Especially for the generation of YAML-based Ansible code, this emphasis on recall is reasonable, as the correctness and completeness of the structure often have a higher priority than minor lexical inaccuracies.

In the case of Ansible YAML generation, chrF provides useful insights into how closely the generated code resembles the reference output at the lowest granularity. It is particularly sensitive to structural fidelity, indentation patterns, and naming conventions that are common in configuration files. The metric is computed for each generated output and its corresponding reference, and then averaged over the entire evaluation set. The chrF metric does not capture semantic correctness or task functionality, its character-based granularity allows it to detect subtle formatting or syntax issues, and complements metrics like ROUGE and METEOR, in order to increase comprehensive assessment of output quality.

Custom Evaluation Metric: Ansible Lint Score

The evaluation metrics ROUGE and METEOR assess lexical and semantic similarities between generated and reference texts. These metrics often fall short in capturing the syntactic correctness and

adherence to domain-specific practices essential for executable code. This limitation is particularly pronounced in infrastructure-as-code scenarios, where minor syntactic deviations can lead to critical deployment failures. To address this, the character-based chrF metric is additionally utilized in this thesis. Unlike traditional word-level metrics, chrF evaluates the similarity between reference and generated text at the character n-gram level, offering a finer-grained assessment that is particularly valuable in whitespace- and formatting-sensitive languages such as YAML. Through this method, minor deviations in structure, indentation, or variable naming are captured more effectively than with lexical metrics alone. The chrF metric therefore serves as a complementary indicator, helping to quantify structural fidelity without relying solely on exact token matches. Nonetheless, while chrF strengthens the analysis of structural similarity, it still does not provide information about the practical correctness or executability of the generated output. To address this remaining gap, recent research has explored the integration of static analysis tools into the evaluation pipeline of LLM-generated code. For instance, Dolcetti et al. (**dolcetti2024helping**) propose a framework that leverages both testing and static analysis to assess and enhance the quality of code produced by LLMs, highlighting the utility of linters in identifying potential vulnerabilities and code errors. Similarly, Peng et al. (**peng2025cweval**) introduce CWEval, an outcome-driven evaluation framework that simultaneously assesses the functionality and security of LLM-generated code, emphasizing the importance of robust evaluation benchmarks that incorporate static analysis. Building upon these insights, this thesis introduces a custom evaluation metric that utilizes `ansible-lint` to perform static analysis on generated Ansible YAML files. By quantifying the extent to which generated code adheres to established Ansible best practices, this metric provides a more nuanced assessment of code quality, complementing traditional lexical and semantic evaluation methods. The integration of `ansible-lint` into the evaluation process ensures that the generated code is not only syntactically correct but also aligns with the conventions and standards expected in real-world deployment scenarios. The evaluation pipeline proceeds in two stages. First, valid YAML segments are extracted from the model's output. Afterwards, each segment is analyzed using `ansible-lint` to quantify rule violations and calculate a quality score. Each YAML segment is written to a temporary file and analyzed using the `ansible-lint` CLI tool. The number of best-practice checks is computed by identifying rule identifiers in the lint output using regular expressions. The final score is computed as:

$$\text{score} = \frac{\text{total\_checks} - \text{failed\_checks}}{\text{total\_checks}} \quad \text{if total\_checks} > 0$$

If no lint checks are triggered, a default score of 1.0 is assigned. The score is clamped to the $[0, 1]$ interval to ensure consistency:

$$\text{score} = \max(0, \min(\text{score}, 1))$$

This process is repeated for all YAML segments in the model output, and an overall evaluation score is derived by computing the arithmetic mean of individual section scores:

$$\text{overall\_score} = \frac{1}{n} \sum_{i=1}^{n} \text{score}_i$$

A score of 1.0 indicates that the generated YAML segment passes all Ansible lint checks, while lower scores reflect the presence of syntactic or stylistic issues. This metric allows for a more functionally grounded assessment of the generated output and is especially relevant in production-grade infrastructure automation tasks where correctness and convention adherence are non-negotiable. However, it is important to note that `ansible-lint` evaluates code based solely on static analysis rules and does not verify the runtime behavior or functional correctness of the generated YAML. Therefore, the metric should be understood as a static approximation of overall code and adherences to Ansible best practices and syntactic validity, but not whether the code executes successfully in a real deployment environment.

Implementation

This chapter provides a detailed overview of the implementation process carried out during the development of this thesis. It begins with the construction and preparation of the dataset using the Ansible Content Parser, followed by a description of the data cleaning and preprocessing steps. Subsequently, the relevant technologies and libraries used to support fine-tuning are introduced, along with a summary of the hardware resources employed. The latter part of the chapter focuses on the fine-tuning procedure itself, including model configuration, optimization strategies, and evaluation metrics. Finally, the implemented checkpointing strategy is presented to ensure reproducibility and fault tolerance during the training process.

Dataset creation with ansible content parser

In this thesis, the initial phase of fine-tuning a LLM involved the development of a custom dataset designed to provide the model with the necessary training data. The primary emphasis was placed on Ansible code, particularly in the context of SAP environments. To ensure a sufficient volume of data for learning, general Ansible code was also incorporated to aid in syntax acquisition.

Ansible Lightspeed, developed by Red Hat, incorporates a robust open-source parser designed to efficiently transform data from a chosen GitHub repository into a JSONL (JSON Lines) file format. This file format is particularly advantageous for machine learning applications due to its structure. Each entry within the JSONL file is placed on a separate line, where each line comprises a JSON object. These objects are meticulously organized into multiple crucial columns, e.g. input and output. This separation ensures that the data is ready for training machine learning models, streamlining the process by clearly delineating the inputs and expected outputs, which are essential for supervised learning tasks. Furthermore, this structured approach aids in maintaining data integrity and enhancing the parser's operational efficiency when dealing with complex datasets from extensive repositories.

In order to generate the raw dataset, specific requirements need to be fullfiled defined by Red Hat. The parser does only support UNIX based operating systems, e.g. macOS or Linux. Additionally, Python 3.10 or higher must be installed on the system to ensure proper functionality. The first step is to install the most recent version of pip and ansible-content-parser library. The Python package

manager pip is used to install and manage software packages written in Python. It is the standard tool for handling Python libraries and allows users to easily install dependencies from the Python Package Index (PyPI). In the context of this thesis, pip is used to install the ansible-content-parser library required for dataset generation. After installation, the parser can be executed via the terminal by providing the relevant repository URL and output directory. Hereby the REPO_URL refers to the Git repository containing the Ansible content to be parsed, while OUTPUT_DIR specifies the directory in which the parsed data will be saved.

```
$ ansible-content-parser REPO_URL OUTPUT_DIR
```

The JSON element below represents a single entry example from the generated ftdata.jsonl file. Each row consists of several attributes, as mentioned before, with the most significant being the input and output fields. The input contains content extracted directly from the source repository, which is a YAML-formatted snippet such as an Ansible playbook section or task block. The output also originates from the same repository and contains the subsequent code segment that follows the input snippet. The dataset consists of multiple sections within the same playbook without duplicates.


```
{
    "data_source_description": "",
    "input": "---\n# Ansible Playbook for SAP BW/4HANA Sandbox
        installation\n\n# Use include_role / include_tasks inside
        Ansible Task block, instead of using roles declaration or Task
        block with import_roles.\n# This ensures Ansible Roles, and the
         tasks within, will be parsed in sequence instead of parsing at
         Playbook initialisation.\n\n\n#### Begin Infrastructure-as-
        Code provisioning ####\n\n- name: Ansible Play to gather input
        for gathering vars and VM provisioning\n  hosts: localhost\n
        gather_facts: false\n\n  # pre_tasks used only for Interactive
        Prompts only and can be removed without impact\n  pre_tasks:\n\
        n    - name: Playbook Interactive - Check if standard execution
         with an Ansible Extravars file is requested by end user",
    "license": "",
    "module": "ansible.builtin.set_fact",
    "output": "      ansible.builtin.set_fact:\n
        playbook_enable_interactive_prompts: \"{{ true if (
```

```
        sap_vm_provision_iac_type is undefined and
        sap_vm_provision_iac_platform is undefined) else false }}\"\n",
    "path": "deploy_scenarios/sap_bw4hana_sandbox/ansible_playbook.yml"
        ,
    "repo_name": "ansible",
    "repo_url": "https://github.com/sap-linuxlab/ansible.
        playbooks_for_sap"
}
```

The core elements of the dataset are the input and output columns, which serve as the primary components for the fine-tuning process of the language model. These columns are generated through the parsing process, where the Ansible Content Parser extracts code directly from the repositories and divides it into sections. Each section is then accordingly assigned to the input or output column, depending on the playbook structure,which essentially transforms the dataset into a sequence-to-sequence learning task, where the input column represents a partial or preceding code segment, and the output column contains the logically succeeding segment. This structure closely mimics real-world code generation and completion tasks, where a model is required to predict the next segment of code based on a given context. Consequently, the prompt provided to the model (the input) resembles the natural context that developers would provide when extending or completing an Ansible playbook, making this dataset particularly suitable for the targeted fine-tuning process. In order to ensure that the model also learns to handle complete playbook generation with questionnaire prompts, the dataset preparation process includes a step where the full Ansible playbook is reconstructed by concatenating the input and output columns. This concatenated form represents a complete, logically ordered playbook, providing the model with examples of fully assembled automation workflows. This dual approach — training on both partial segments and complete playbooks — aims to enhance the model's ability to generate syntactically correct and contextually appropriate Ansible code, even in cases where only partial context is available in the input.

Selection of relevant repositories

The selection of suitable repositories is an essential step when creating a high-quality dataset for training a LLM tailored to Ansible playbook generation. The quality, diversity, and relevance of the

selected repositories directly influence the model's ability to generate meaningful and technically correct code (**lozhkov2024starcoder**). To ensure that only useful and representative data is included, a multi-stage selection process is applied. This process is inspired by the methodology used in the creation of The Stack v2 (**lozhkov2024starcoder**), a dataset developed for training the StarCoder2 model.

The first step is to find a comprehensive source of open-source code repositories, namely those that provide Ansible playbooks for our particular use case (**lozhkov2024starcoder**). This is crucial because the domain for which the model will be optimized must be reflected in the training data. Only the most recent version of each repository is taken into consideration to guarantee that only the most pertinent and recent code is used. Furthermore, only the main branch - which usually denotes the version of the representative codebase that is stable - is extracted.

A deduplication step comes after the repositories have been collected. Because many repositories on sites like GitHub are forks or almost identical copies of other projects, this is required. Overfitting, in which the model learns particular structures instead of broad patterns, might result from training on duplicate data. This is avoided by verifying the content of each repository and retaining only distinct repositories in the final dataset (**lozhkov2024starcoder**).

The next stage in the dataset construction after deduplication is license verification. This is usually considered when ethical and legal concern involves in licensing of the training data, which is relevant when the training dataset consists of code repositories with potentially restrictive software licenses. In the context of this thesis, the focus is on academic fine-tuning and evaluation of the Phi-4 LLM, there is no legal requirement to comply as no code or model is distributed or commercially made available. As a result, regardless of their licensing state, all accessible repositories pertinent to the research topic are taken into account. The Starcoders license verification step excludes repositories with restrictive licenses like GPL or those without a clear license statement and only selects repositories with liberal licenses like MIT or Apache 2.0 to ensure compliance. For most of the repositories, where such information was missing, the authors used the ScanCode Toolkit to detect licenses at the file level, which were then classified according to permissiveness using identifiers and a custom propagation method (**lozhkov2024starcoder**). The authors further filtered the dataset to include only permissively licensed and unlicensed files, explicitly excluding copyleft and commercial licenses due to associated legal risks (**lozhkov2024starcoder**). Starcoders approach

provides a framework for ensuring license compliance in open-source LLM publication. However, the objectives of this thesis differ substantially from that context. In this work, the dataset was used exclusively for academic purposes, and no fine-tuned model, dataset, or code derived from the training data is distributed. The use of copyrighted or licensed material purely for internal research or noncommercial academic purposes is generally considered to fall under fair use or fair dealing. This interpretation is supported by current academic practices and aligns with the usage patterns allowed under many permissive open-source licenses, such as MIT, BSD, and Apache-2.0 **licensing**. Furthermore, as the model output was only analyzed in aggregate for evaluation purposes, and no training samples or generated completions are published, there is no redistribution of licensed content, and therefore no breach of licensing terms.

Afterwards, a manual inspection process is conducted. This involves reviewing samples from different repositories to visually assess their quality (**lozhkov2024starcoder**). Manual inspection is particularly useful for identifying repositories that, while technically valid, contain code copied from tutorials or that exhibit poor coding practices unsuitable for training a high-quality model.

By following this structured, multi-stage selection process, the final dataset achieves a balance between relevance and quality. This approach ensures that the trained model learns from realistic, high-quality examples of Ansible playbooks, improving its ability to generate meaningful and practically useful code. This process is a technical necessity that reflects the increasing importance of responsible data sourcing in machine learning research.

Data preparation and cleaning

Data preparation represents a critical foundation for effective machine learning model development, particularly for large language models (LLMs) where the quality of training data directly influences performance outcomes (**bommasani2021opportunities**). This section details the comprehensive preprocessing methodology implemented to ensure dataset integrity prior to fine-tuning the Phi model. As emphasized by Sambasivan et al. (**sambasivan2021everyone**), data quality issues can propagate through the machine learning pipeline, creating cascading failures that impact model reliability, fairness, and generalization capabilities.

Eliminate duplicates

As a first step, the dataset was examined for duplicate rows. Duplicate data can skew the model's learning process by overrepresenting certain patterns or outputs, thereby introducing unwanted bias and reducing the diversity of learned representations (**lee2021deduplicating**). Each data record in the dataset represented an input-output pair, which serves as a training example. A record was considered a duplicate if the values in both the input and output columns were identical to another row. This approach ensured that each training example was unique and avoided overfitting on repeated sequences.

Handling missing values

Subsequently, the dataset was analyzed for missing or null values in either the input or output columns. Missing values can occur due to incomplete data extraction, preprocessing errors, or inconsistencies in the original source. Their presence not only renders a training sample unusable but can also introduce noise into the model's learning process (**kandel2011wrangler**). Only records with complete and valid information were retained, as both the prompt (input) and expected response (output) are critical for supervised learning in fine-tuning scenarios (**zhou2024reviewhandlingmissingdata**). Multiple studies have confirmed that removing incomplete entries, rather than applying imputation techniques, is optimal for language modeling tasks where the semantic integrity of each example is paramount (**raffel2020exploring**).

Quality assurance

The integration of duplicate elimination and missing value handling constitutes a robust quality assurance framework aligned with established best practices in machine learning data curation (**gebru2021datasheets**). This approach prioritizes dataset integrity over raw volume, recognizing that even a marginally smaller dataset of higher quality produces superior model performance compared to larger, noise-contaminated alternatives (**longpre2023pretrainer**). Our quality assurance protocol enforced three primary criteria:

1. **Uniqueness:** Each training example must be distinct to prevent overrepresentation bias.
2. **Completeness:** All records must contain valid input and output data.
3. **Consistency:** Data formatting and structure must be uniform across all examples.

These criteria conform to the data quality dimensions proposed by Wang and Strong (**wang1996beyond**) and subsequently adapted for machine learning contexts by Hutchinson et al. (**hutchinson2021towards**).

Dataset Splitting and Preprocessing

To facilitate robust evaluation during and after the training process, we implemented a stratified partitioning strategy that divided the cleaned dataset into three distinct subsets: training, validation, and testing. The partitioning followed a 70/15/15 ratio distribution, which has been empirically validated as optimal for language model fine-tuning scenarios by Wei et al. (**wei2022emergent**). The dataset, sourced from Hugging Face under the name `FurkanGuerbuez/ansible_training`, was initially loaded as a single training set and subsequently divided using the `train_test_split` method provided by the `datasets` library. This approach ensures that the model is trained on a substantial portion of the data (70%) while reserving sufficient examples for validation (15%) and final evaluation (15%). The use of a fixed random seed (42) guarantees reproducibility of the partitioning process, as recommended by Pineau et al. (2021) in their guidelines for reproducible machine learning research.

```
from datasets import load_dataset

# Load the complete dataset
dataset = load_dataset("FurkanGuerbuez/ansible_training", split="train")

# Split into 70% train and 30% test/validation
dataset_split = dataset.train_test_split(test_size=0.3, seed=42)

# Split the 30% again into equal parts: 15% validation, 15% test
validation_test_split = dataset_split["test"].train_test_split(test_size
    =0.5, seed=42)

# Reconstruct the full dataset dictionary
split_dataset = {
    "train": dataset_split["train"],
    "validation": validation_test_split["train"],
    "test": validation_test_split["test"]
}
```

Tokenization and Label Preparation

For effective supervised fine-tuning in an autoregressive framework, each input-output pair required transformation into a unified sequence format compatible with the model's training objective. This preprocessing stage involved concatenating input and output fields into a single text string, applying tokenization, and constructing appropriate label tensors for the language modeling loss computation. The implementation utilized a model-specific tokenizer to convert text into token IDs, with special attention given to label preparation. Following the methodology established by Brown et al. (**brown2020language**) and refined by Ouyang et al. (**ouyang2022training**), padding tokens were masked with a sentinel value (-100) to prevent them from influencing the loss calculation:

```
def preprocess_function(entry):
    """Tokenizes input-output pairs for autoregressive training."""
    instruction = entry["input"]
    response = entry["output"]
    text = f"\n{instruction}\r\n{response}"

    # Tokenize the combined text
    encoding = tokenizer(text, truncation=True)

    # Set labels identical to input_ids, masking padding tokens
    encoding["labels"] = [
        token if token != tokenizer.pad_token_id else -100
        for token in encoding["input_ids"]
    ]
    return encoding
```

This function was systematically applied to each partition of the dataset. To optimize the training process, extraneous metadata columns were removed, retaining only the essential tokenized representations:

```
columns_to_remove = ['data_source_description', 'input', 'license', '
    module',
                     'output', 'path', 'repo_name', 'repo_url']

```

```
train_data_token = split_dataset['train'].map(preprocess_function,
    remove_columns=columns_to_remove)
val_data_token = split_dataset['validation'].map(preprocess_function,
    remove_columns=columns_to_remove)
```

This approach follows the data minimalism principle advocated by Bender et al. (**bender2021dangers**), which emphasizes focusing computational resources on information directly relevant to the learning objective. The comprehensive data preparation methodology described here establishes a robust foundation for the subsequent fine-tuning process. By systematically addressing duplicates, missing values, and format inconsistencies, while implementing a principled partitioning strategy, we have created a high-quality dataset optimized for training large language models for this thesis. This meticulous approach to data preparation is essential for maximizing model performance and ensuring reliable generalization to various inputs (**longpre2023pretrainer**).

Technology stack and important libraries

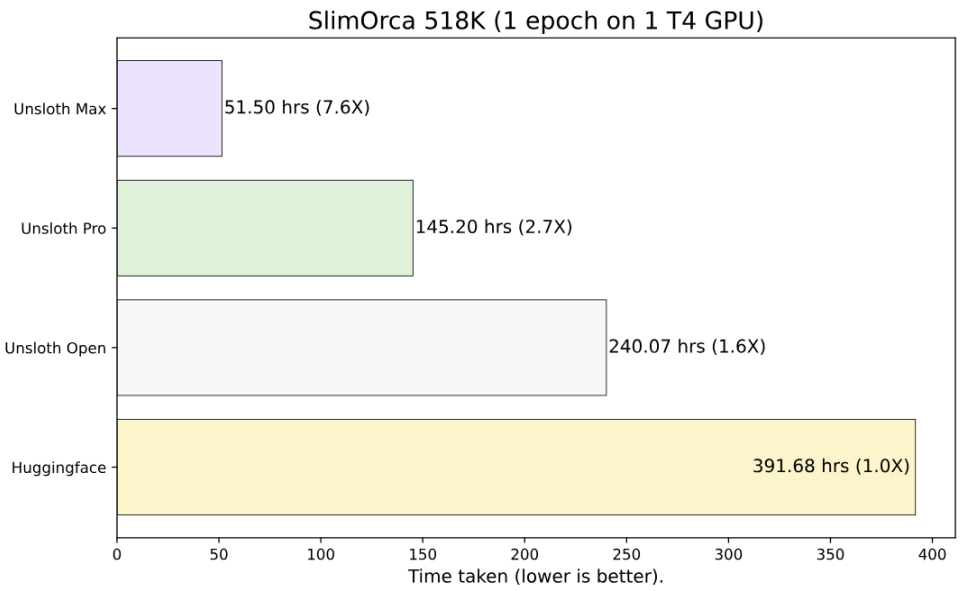
In the following section, the core libraries and frameworks utilized in the implementation of this thesis are introduced. These components play a central role in enabling the fine-tuning of the Phi-4 large language model. Their selection is primarily driven by performance considerations, with a focus on maximizing computational efficiency, scalability, and compatibility with modern GPU architectures. Each library contributes to the overall training pipeline, either by optimizing memory usage, accelerating computations, or simplifying model integration.

Unsloth

The first and most important library utilized within the technology stack is Unsloth, which maximizes resource efficiency. Unsloth claims to reduce memory usage by 60 percent, while enabling six times larger batch sizes. This is achieved through software optimizations that utilize the given resource capacities. Optimizing autograd for efficient Low-Rank Adaptation (LoRA) fine-tuning is crucial. When computing the gradients of a function coupled with LoRA adapters, six matrix differentials must be computed due to the freezing of the original weight matrices. In PyTorch, these gradients are automatically derived by saving all operations during forward propagation as a computational graph, which is then used for backpropagation. However, this process increases computation time and memory usage due to the storage of the computational graph, reducing overall efficiency.

With Unsloth's manual autograd, gradients are derived manually, eliminating the need to store and use a computational graph. This reduces memory usage and increases performance, as no automatic gradient computation is required. The downside of manual derivation is that Unsloth has to implement these for each LLM, hence Unsloth has limited selection of LLM's. Figure 2 shows the performance in training time on identical dataset and hardware by comparing the same HuggingFace model and the Unsloth optimized model.

Figure 2
Training time comparision



Unsloth Max achieves up to 7.6x speedup over Hugging Face. Image adapted from Unsloth (**unsloth2024**).

PyTorch Framework

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It provides two high-level features that are critical for modern machine learning workflows: a tensor computation library with strong GPU acceleration support and an automatic differentiation engine for building and training neural networks (**paszke2019pytorch**). Unlike traditional machine learning libraries, PyTorch supports dynamic computation graphs, which allow developers to define, modify, and execute the network structure at runtime. This approach facilitates a high level of flexibility, making PyTorch particularly suited for research and prototyping tasks. Additionally, the PyTorch ecosystem provides three domain-specific libraries that extend its functionality for specialized machine learning tasks. These include torchvision for computer vision applications,

torchaudio for audio signal processing and feature extraction, and torchtext for natural language processing (NLP) workflows. In the context of this thesis, the torch library is utilized, initializing a tensor to store predictions, ensuring it has the correct shape to accommodate the batch size and the number of classes the model predicts. This initialization is done for the evaluation step of the LLM where you might encounter cases where the model doesn't produce any predictions for certain examples. In such cases, having a zero-filled tensor ensures a placeholder for these empty predictions, which allows to proceed with metrics calculations without encountering errors due to missing values.

```
# Ensure logits are always tensors, even if generation is empty
if isinstance(predictions, FastLanguageModel.models._utils.EmptyLogits):
    predictions = torch.zeros((labels.shape[0], model.config.num_labels),
        dtype=torch.float32)
    predictions = predictions.to(labels.device)
```

CUDA GPU Acceleration

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the full power of NVIDIA GPUs for general-purpose computing. CUDA allows for massive parallelization of mathematical operations that are essential for deep learning, such as matrix multiplications, convolutions, and tensor transformations.

In the context of PyTorch, CUDA is used to accelerate the forward and backward passes during model training. When enabled, PyTorch automatically offloads eligible operations to the GPU, significantly reducing training time, especially for large models and datasets. In order to ensure compatibility between PyTorch and CUDA, precompiled binaries are made available for different CUDA versions, which is important since not all PyTorch versions are compatible. For example, in this thesis, the following installation command is used to install PyTorch with CUDA 12.4:

```
$ pip install torch torchvision torchaudio torchtext --index-url https://
    download.pytorch.org/whl/cu124
```

This command retrieves optimized PyTorch binaries from the official PyTorch repository that are compatible with CUDA 12.4. The use of this custom index URL ensures that the installed packages

can leverage GPU acceleration effectively, provided that the system includes a compatible NVIDIA GPU and driver.

Evaluate and NumPy: Calculation libraries

The evaluate library is an open-source Python package developed by Hugging Face, designed to simplify the computation of evaluation metrics in machine learning workflows. It provides a standardized interface for accessing a wide range of commonly used metrics across various domains, such as natural language processing, computer vision, and structured prediction tasks. A key advantage of the evaluate library is its integration with the Hugging Face ecosystem and its compatibility with popular datasets and models. It enables users to load metrics with a single command and apply them to predictions and references with minimal code.

```
# Install and load ROUGE score
pip install rouge_score
from evaluate import load
rouge = load("rouge")
# Calculate ROUGE score
rouge_results = rouge.compute(predictions=modified_predictions,
    references=decoded_references, use_aggregator=True)
```

Internally, many of the metrics are implemented using highly optimized libraries such as scikit-learn, which ensures both reliability and efficiency. In the context of this thesis, evaluate is utilized to measure the performance of the fine-tuned Phi-4 model. Specific metrics such as ROUGE, METEOR, and CHRF are employed to assess the quality of generated Ansible code sequences in comparison to reference completions. These metrics and their results are discussed in detail in the Results chapter 5. In addition to these standard measures, a custom evaluation metric based on ansible-lint is also integrated to assess syntactic correctness from a domain-specific perspective.

Numerical Python (NumPy) is a fundamental package for scientific computing in Python. It provides a high-performance multidimensional array object, called the ndarray, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is widely regarded as the backbone of the Python data science ecosystem and is used extensively in fields such as machine learning, data analysis, and numerical simulations. One of NumPy's core advantages is its ability to perform element-wise operations and broadcasting over large arrays without the need for explicit

loops, which significantly improves performance compared to standard Python lists. Furthermore, NumPy offers tools for integrating C/C++ and Fortran code, which allows for low-level optimization and compatibility with other scientific computing libraries. In the context of this thesis, NumPy is primarily used to pre-process and transform data set elements, statistical evaluations, and perform lightweight mathematical operations prior to model input or during result analysis. Its efficient memory model and vectorized computation capabilities make it an ideal choice for handling large-scale data during the model fine-tuning process.

Hardware Resources

Training and fine-tuning Large Language Models (LLMs) requires substantial computational resources due to the models' architecture complexity and parameter counts. While recent advancements in optimization techniques have improved efficiency, the hardware demands remain significant (**park2025review**). This section explores the specific computational requirements for LLM fine-tuning, with particular focus on memory optimization techniques, available cloud computing solutions, and the benefits of integrated development environments.

Microsoft's Phi-4 model, with its 14 billion parameters, presents an interesting case study in hardware requirements. Despite being significantly smaller than leading models like GPT-4 (estimated at 1.76 trillion parameters), Phi-4 still requires substantial computational resources for fine-tuning (**2025micPhi4**). The model's architecture, which features improved attention mechanisms and deeper layers, contributes to its hardware demands despite its relatively modest parameter count. According to benchmarks published by Chen and colleagues (**chen2025practical**), fine-tuning Phi-4 with a reasonable batch size requires a minimum of of 24 GB GPU RAM for base fine-tuning, 32-40 GB VRAM and high-bandwidth memory for optimal performance

The computational demands of LLM fine-tuning have created a robust market for cloud-based GPU/TPU resources. First, the Google Cloud Platform, which offers various GPU types (NVIDIA T4, V100, A100) and custom TPUs. Next, the Amazon Web Services (AWS), which Provides EC2 instances with NVIDIA GPUs and specialized machine learning instances. Lastly, Microsoft Azure, which Features NVIDIA GPU-equipped virtual machines optimized for AI workloads. Each provider offers different pricing models, hardware options, and software integrations that can significantly impact development efficiency and cost (**visnjic2023cloudgpu**).

For this thesis research, computational resources are procured through a monthly subscription to Google Colab Pro+, which provides critical advantages for LLM development. Subscriptions grant priority access to premium hardware including NVIDIA T4, P100, V100 GPUs, and in some cases, A100 GPUs or TPU v2/v3 accelerators. The platform offers extended runtime limits with longer execution sessions compared to the free tier, allowing for more extensive training runs necessary for effective model fine-tuning. Google Colab Pro+ provides increased memory allocations of up to 52GB of RAM and sufficient VRAM to accommodate models like Phi-4. The cost-effective pay-as-you-go pricing model eliminates the need for capital expenditure on dedicated hardware, making advanced AI research more accessible to individual researchers and small teams. According to comparative analyses by Bisong (**bisong2019building**), Google Colab represents one of the most accessible entry points for researchers conducting LLM fine-tuning without dedicated institutional resources.

One of the most significant advantages of the Google Colab environment is its seamless integration with Google Drive. This powerful connection transforms how researchers work with large AI models by solving many common workflow challenges. Researchers can store their training and validation datasets directly in Google Drive and access them effortlessly from Colab notebooks, eliminating cumbersome data transfer processes. When fine-tuning produces valuable model checkpoints and weights, these can be automatically saved to persistent storage without manual intervention. The system also preserves important research artifacts like evaluation results, training logs, and visualizations in an organized manner for future reference and analysis. Perhaps most valuable for research teams, this integration enables truly collaborative development where multiple researchers can simultaneously access shared files and notebooks through common Drive folders. As Johnson and colleagues noted in their study (**johnson2023colab**), this seamless integration substantially reduces the friction typically associated with data and model management in machine learning workflows, allowing researchers to focus more on innovation and less on technical overhead.

The integrated nature of Google Colab eliminates several technical challenges that typically plague machine learning research. Researchers no longer need to wrestle with driver installation and configuration, as GPU drivers, CUDA, and cuDNN come pre-installed and properly configured from the start. The platform handles the notoriously difficult framework compatibility management, resolving common incompatibilities between ML frameworks, CUDA versions, and Python environments that often consume days of troubleshooting. System administration overhead disappears as Google

manages all updates, security patches, and system maintenance behind the scenes. Researchers can also achieve consistent environment reproducibility by using initialization cells, ensuring experiments run under identical conditions every time. These advantages free researchers to concentrate on what truly matters - model development and innovation - rather than getting bogged down in infrastructure management. Koch and Barraza's study (**koch2024colab**) specifically identified this reduction in technical overhead as a significant productivity factor in LLM research, demonstrating how cloud-based environments like Colab are transforming the research landscape by removing traditional barriers to entry.

While cloud solutions like Google Colab offer convenience, they do present certain performance trade-offs compared to dedicated hardware. The platform's variable resource allocation means GPU availability can fluctuate depending on overall demand, occasionally resulting in queue times during peak usage. Network transfer overhead becomes apparent when moving large datasets between Google Drive and Colab instances, introducing latency that can slow down training preparation. Even with paid subscription tiers, session time limits impose constraints on continuous training runs, requiring thoughtful checkpointing strategies. The hardware variety inherent to cloud platforms means the specific GPU assigned to a session may vary between sessions, potentially affecting performance consistency. To address this last challenge, for all training and evaluation runs in this thesis, the same GPU hardware resource is explicitly selected to overcome potential bias in training and evaluation times, ensuring fair and consistent comparisons between models and techniques. Despite these limitations, Colab's combination of accessibility, seamless integration with existing workflows, and significant reduction in technical complexity makes it an appropriate choice for the LLM fine-tuning work presented in this thesis.

Fine-Tuning Process

The following section provides a comprehensive walkthrough of the fine-tuning process applied to the Phi-4 model. Building on the initialization and configuration strategies outlined previously, this part of the implementation focuses on adapting the pre-trained model to our specific downstream task using supervised fine-tuning. Key components include the integration of parameter-efficient fine-tuning techniques via LoRA, the training loop setup using Hugging Face's Trainer API, and the definition of relevant hyperparameters and optimizer configurations. By documenting each stage in detail—from tokenizer setup and model adaptation to loss calculation and evaluation

strategy—this section establishes the methodological foundation for reproducible and resource-efficient large language model training.

Model Initialization and Configuration

After the successful installation of the required library, the model initialization process for the fine-tuning pipeline was executed with meticulous attention to configuration details. The Phi-4 model, a central component in this research, was loaded using the Unsloth library's specialized optimization techniques. This library is specifically designed to enhance efficiency when fine-tuning large-language models, particularly for challenging long-context tasks that require significant computational resources.

The implementation leverages the `FastLanguageModel.from_pretrained()` function, which serves as the gateway to accessing the pre-trained Phi-4 model while applying crucial optimizations. Two configuration parameters play a particularly important role in this initialization process. The first is the maximum sequence length, deliberately set to 7,000 tokens—a substantial context window that enables the model to process extensive input sequences containing complex code structures, detailed documentation, or lengthy analytical content. This expanded context capacity is essential when working with software development tasks that frequently involve understanding relationships between distant elements in code or documentation. The second critical parameter is `load_in_4bit`, which is activated by setting it to True. This enables 4-bit quantization, an advanced memory optimization technique that dramatically reduces the model's memory footprint. In practical terms, this quantization approach compresses the model's weights from their native 16-bit or 32-bit floating-point representation down to just 4 bits per parameter. This compression yields approximately a 4x to 8x reduction in memory usage compared to full-precision models. Despite this substantial compression, carefully implemented 4-bit quantization maintains remarkably good performance on most NLP tasks, making it an excellent trade-off for resource-constrained environments.

The following code implementation clearly demonstrates this initialization process, beginning with the necessary imports from the Unsloth library and PyTorch. The configuration variables are explicitly defined at the top level: `max_seq_length` is set to 7000, `load_in_4bit` to True, and the model source is specified as "unsloth/Phi-4" (with an alternative commented path to a locally fine-tuned version stored in Google Drive). The model and tokenizer are then loaded

simultaneously using the `FastLanguageModel.from_pretrained()` function, applying all specified configuration parameters. Additionally, the implementation includes a `model_init()` function that simply returns the configured model instance. This function plays an important role in training frameworks that require model initialization to be wrapped in a callable, allowing for potential reinitialization during techniques like hyperparameter optimization or cross-validation. This architectural choice enhances flexibility in the experimental setup, making it easier to integrate with various training pipelines and evaluation frameworks. By combining these optimization techniques—expanded context length and 4-bit quantization—the implementation achieves a balance between model capability and computational efficiency. This approach allows the research to proceed with sophisticated fine-tuning experiments even within the constraints of cloud-based GPU resources, demonstrating a thoughtful approach to managing the computational demands of modern LLM research.

```
from unsloth import FastLanguageModel
import torch
max_seq_length = 7000
load_in_4bit = True
STRING_MODEL = "unsloth/Phi-4"
#STRING_MODEL = "/content/drive/MyDrive/finetuned_phi4"

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = STRING_MODEL,
    load_in_4bit = load_in_4bit,
    max_seq_length = max_seq_length
)

def model_init():
    return model
```

Integration of Unsloth and LoRA

The integration of LoRA in Unsloth is implemented through the `FastLanguageModel` abstraction, providing an elegant and resource-efficient approach to model adaptation. After loading a pretrained model, the `model.add_lora()` method injects LoRA modules into both attention and feedforward layers. This implementation offers precise control over the adaptation process

through several key hyperparameters. The code implementation shows how the PEFT (Parameter-Efficient Fine-Tuning) model is configured with carefully selected parameters. The rank parameter (r) is set to 16, establishing the dimensionality of the low-rank matrices that will be used for adaptation. This value balances expressive power against parameter count—larger values can capture more complex adaptations but require more memory. Target modules are explicitly defined to include all projection layers in both attention mechanisms ("q_proj", "k_proj", "v_proj", "o_proj") and feedforward networks ("gate_proj", "up_proj", "down_proj"), ensuring comprehensive coverage of model components that benefit most from adaptation.

The `lora_alpha` parameter, set to 16, controls the scaling factor applied to the LoRA updates, effectively managing their influence on the overall model output. With `lora_dropout` set to 0, the implementation prioritizes stability in the adaptation process rather than introducing additional regularization through dropout. The `bias` parameter is set to `"none"`, indicating that the bias terms remain frozen during training. Memory efficiency is further enhanced through the `use_gradient_checkpointing` parameter, set to `"unsloth"`, leveraging Unsloth's optimized implementation of gradient checkpointing that trades computation for memory by recalculating certain activations during backpropagation rather than storing them. A fixed `random_state` (3407) ensures reproducibility across experiments, while advanced options like `use_rslora` and `loftq_config` are disabled for this implementation.

```
model = FastLanguageModel.get_peft_model (
    model,
    r = 16,
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth",
    random_state = 3407,
    use_rslora = False,
    loftq_config = None )
```

This implementation ensures that only the LoRA parameters—a tiny fraction of the full model—are updated during training, while the base model remains frozen. As a result, the model can be fine-tuned with a minimal memory footprint, even on consumer-grade hardware. The memory savings are substantial: rather than updating billions of parameters, this approach focuses on just millions of carefully placed parameters, reducing memory requirements by orders of magnitude.

The use of LoRA through Unsloth was instrumental in enabling practical fine-tuning of the Phi-4 model within this project. Without these parameter-efficient techniques, adapting a 14-billion parameter model would require prohibitive computational resources beyond the scope of most research projects.

LoRA represents a critical advancement in parameter-efficient fine-tuning for LLMs, and its implementation in Unsloth offers a sophisticated and resource-conscious way to adapt large-scale models for specific downstream tasks. This approach avoids the substantial computational and storage costs of traditional fine-tuning while maintaining comparable performance on downstream tasks, making it an essential technique for modern LLM adaptation.

This approach ensures that only the LoRA parameters are updated during training, while the base model remains frozen. As a result, the model can be fine-tuned with a minimal memory footprint, even on consumer-grade hardware. The use of LoRA through Unsloth was instrumental in enabling practical fine-tuning of the Phi-4 model within this project. As mentioned in the Solution Design chapter, LoRA represents a critical advancement in parameter-efficient fine-tuning for LLMs. Its implementation in Unsloth offers an elegant and resource-conscious way to adapt large-scale models for specific downstream tasks, without incurring the full cost of traditional fine-tuning.

Training Procedure

The following code snippet establishes the supervised fine-tuning setup for our language model. The implementation follows the recommended practices for transformer model training as outlined by Wolf et al. (**wolf-etal-2020-transformers**) in their seminal work on the Transformers library. First, we are importing essential libraries: SFTTrainer from the TRL (Transformer Reinforcement Learning) package, various components from Hugging Face's transformers, and a hardware capability checker from unsloth. The SFTTrainer provides a streamlined interface for supervised fine-tuning of

language models as described by Raffel et al. (**raffel2020exploring**) in their exploration of transfer learning with transformer models.

The `TrainingArguments` configuration defines the precise parameters that govern our training process. We are using a small batch size of 2 examples per device, but compensating with 4 gradient accumulation steps. This approach effectively simulates a larger batch size of 8 while being gentler on memory requirements as the efficiency and cost of the google hardware resources is kept low, a technique validated by Ott et al. (**ott2018scalingneuralmachinetranslation**) in their work on scaling neural machine translation. The warmup period spans 100 steps, giving the optimizer time to stabilize before learning begins, following the recommendations of Popel and Bojar (**Popel_2018**) for transformer training stability. We have scheduled a complete 3-epoch training run, which represents a balance between thorough learning and computational efficiency. According to Li et al. (**li2023using**), this duration typically provides sufficient exposure to the dataset without risking overfitting. Logging and evaluation checkpoints occur every 100 steps, providing regular insights into training progress, which aligns with best practices for monitoring convergence as discussed by Mosbach et al. (**mosbach2021stabilityfinetuningbertmisconceptions**). The learning rate of 8e-5 was carefully selected after preliminary experiments indicated that this value produces optimal convergence without overshooting. This falls within the recommended range for fine-tuning transformer models as established by Devlin et al. (**devlin2018bert**) in their work on BERT pre-training. We are leveraging the memory-efficient 8-bit AdamW optimizer, which reduces memory footprint while maintaining performance comparable to full-precision optimizers. This optimization technique was validated by Dettmers et al. (**dettmers20228bitoptimizersblockwisequantization**), who showed that 8-bit optimizers can reduce memory usage by up to 75% without compromising model quality. A linear learning rate schedule gradually decreases the learning rate throughout training, a strategy shown to improve convergence in transformer models by Howard and Ruder (**howard2018universallanguagemodelfinetuning**). The random seed is set to 3407 to ensure reproducibility throughout the training runs, adhering to the reproducibility standards advocated by Pineau et al. (**Pineau2021**). Once the training parameters are configured, we instantiate the SFTTrainer with our fine-tuning model, the associated tokenizer, both validation and training datasets, a maximum sequence length parameter that constrains input sizes, a specialized data collator for sequence-to-sequence tasks, our previously defined training arguments, and a custom metrics computation function to evaluate model performance. This approach to trainer configuration follows

the structure recommended by Lhoest et al. (**lhoest2021datasetscommunitylibrarynatural**) in their work on datasets and evaluation metrics for natural language processing. In order to run the training procedure, the defined trainer with its configurations can be run with the function call `trainer.train()`.

```
trainingargs = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 100,
        num_train_epochs = 3, #3 epochs full training run.
        logging_steps = 100,
        eval_steps=100,
        eval_strategy="steps",
        learning_rate = 8e-5,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported,
        optim = "adamw_8bit",
        #weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
        overwrite_output_dir=True,
        report_to = "none"
    )
trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    eval_dataset=reduced_val_data_token,
    train_dataset = train_data_token,
    max_seq_length = max_seq_length,
    data_collator = DataCollatorForSeq2Seq(tokenizer = tokenizer),
    args = trainingargs,
    compute_metrics=compute_metrics,
)
```

Evaluation Metrics

The evaluation of the generated Ansible YAML code was conducted using both standard and custom metrics. As described in the Solution Design chapter, the following metrics were applied: ROUGE, METEOR, chrF, and a custom Ansible Lint Score. This section provides a technical explanation of how these metrics were implemented and computed.

ROUGE, METEOR, and chrF Evaluation. To compute the established metrics - as mentioned before - the `evaluate` library was used. The following metrics were loaded and evaluated against the test set:

```
from evaluate import load

rouge = load("rouge")
meteor = load("meteor")
chrf = load("chrf")
```

To ensure fair comparisons, model predictions were truncated to match the reference output lengths:

```
def calculate_rouge_with_length_manipulation(decoded_predictions,
    decoded_references):
    modified_predictions = []
    for pred, ref in zip(decoded_predictions, decoded_references):
        ref_length = len(ref)
        modified_predictions.append(pred[:ref_length])
    return modified_predictions
```

Here, `decoded_predictions` contained the outputs generated by the model, and `decoded_references` were constructed from the input-output pairs of the test dataset. The truncation ensured consistency in length between predictions and references before evaluation.

The computed metrics were then evaluated as follows:

```
rouge_results = rouge.compute(predictions=modified_predictions,
                              references=decoded_references,
                              use_aggregator=True)

```

```
meteor_results = meteor.compute(predictions=modified_predictions,
                                references=decoded_references)

chrf_results = chrf.compute(predictions=modified_predictions,
                            references=decoded_references)
```

Ansible Lint Score Evaluation. The Ansible Lint Score was implemented to validate syntax and adherence to best practices in generated YAML code. As described in the Solution Design chapter, this custom metric applies `ansible-lint` to extract rule violations. The following steps were implemented in Python:

First, valid YAML blocks were extracted from the model output using regular expressions:

```
def extract_yaml_sections(input_text):
    yaml_pattern = re.compile(r"```yaml\n(.*?)\n```", re.DOTALL)
    yaml_sections = yaml_pattern.findall(input_text)
    if yaml_sections is None:
        return "Empty"
    return yaml_sections
```

Each YAML block was then analyzed with `ansible-lint`, and a score was computed based on the ratio of passed checks:

```
def calculate_ansible_lint_scores(yaml_sections):
    scores = []
    for idx, section in enumerate(yaml_sections, 1):
        with tempfile.NamedTemporaryFile(mode="w", suffix=".yml", delete=
            False) as tmpfile:
            tmpfile.write(section)
            tmpfile_path = tmpfile.name

            result = !ansible-lint {tmpfile_path}
            total_checks = len(re.findall(r"\[(.*?)\]", " ".join(result))
                )
            failed_checks = len(result)
```

```
11
12              score = (total_checks - failed_checks) / total_checks if
                    total_checks else 1.0
13              score = max(0, min(score, 1))
14              scores.append(score)
15      return scores
```

The final overall lint score was computed by averaging over all extracted YAML sections:

```
scores_array = calculate_ansible_lint_scores(yaml_sections)
overall_score = np.mean(scores_array)
```

This score was used to reflect syntactic and stylistic correctness of the generated YAML, supplementing the token-level and character-level metrics with domain-specific validation.

Checkpointing and Model Saving

Throughout the training process, we implemented a robust checkpointing strategy to preserve model progress and facilitate subsequent evaluation procedures, in case of abrupt connection issues or other problems encountering while training process without losing all progress in model training. Given the extensive 6-hour training duration, safeguarding intermediate model states proved essential for both recovery capabilities. Our implementation stores model checkpoints using the safetensor file format, a specialized binary format optimized for efficient storage and loading of deep learning models (**safetensors**). This format provides advantages over traditional pickle-based approaches, including improved security and accelerated loading times, particularly beneficial for large language models. The checkpoint files were systematically saved to a dedicated Google Drive location, which offered several advantages over local storage. This approach ensured persistence of model weights beyond the ephemeral notebook environment, as Google Colab sessions typically terminate after prolonged periods of inactivity or upon reaching maximum runtime limits. By establishing a direct connection to Google Drive, we created a seamless pathway for accessing model checkpoints in subsequent sessions without requiring complex data transfer protocols. The checkpointing frequency was calibrated to balance storage efficiency with recovery granularity. Specifically, checkpoints were saved every 500 training steps. Additionally, we implemented a rotating checkpoint strategy that

maintained only the three most recent checkpoints, thereby conserving storage resources. The fine tuned model can then be saved into the drive, with the provided function form the SFTTrainer library.

```
trainer.save_model("/content/drive/MyDrive/EXAMPLE_FILENAME")
```

Results

This chapter outlines the outcomes of the fine-tuning process of the two iterations. Each iteration is evaluated based on runtime metrics, evaluation loss progression, and standard NLP metrics that are suitable for our use case such as ROUGE and METEOR. Additionally, a custom Ansible Lint Score is introduced to assess the syntactic validity of the generated code. The two iterations were conducted using the same model architecture and configuration to ensure comparability. Each iteration is fine-tuned on the base Phi-4 LLM. Differences in performance and training behavior are analyzed and discussed in detail to provide insight into the reliability, consistency and reproducibility of the fine-tuning procedure.

First Iteration

The first iteration serves as the initial application of the fine-tuning pipeline on the Phi-4 model. The configuration parameters, including learning rate schedule, batch size, and optimizer settings, were defined prior to training and held constant throughout the experiment. The objective of this run was to establish a baseline for both performance and runtime behavior, enabling a meaningful comparison with the second iteration. The results in this section reflect how the model adapted to the task under these fixed conditions and provide insight into its convergence characteristics.

Runtime Training Results

Table 2 provides an overview of key training metrics collected during the fine-tuning process. The table summarizes both quantitative performance indicators and runtime characteristics recorded at defined training intervals. The `loss` column reflects the model's training loss at each checkpoint. As training progresses, the loss decreases consistently from an initial value of 0.4628 to 0.0450 at step 1300. This trend suggests effective learning and convergence. The `grad_norm` values, offer insight into the magnitude of gradient updates and indicate stable optimization behavior over time. The applied `learning_rate` follows a controlled decay, beginning at 8×10^{-6} and gradually decreasing toward smaller values in increasing training stages. Evaluation was performed every 100 training steps. The column `eval_loss` captures the validation loss and shows a continuous improvement, decreasing from 0.2503 to 0.0289. The runtime metrics `eval_runtime`, `eval_samples_per_second`, and `eval_steps_per_second` remain relatively stable throughout, indicating consistent computational performance during evaluation. The final row summarizes the

overall training runtime and throughput. The total `train_runtime` amounts to approximately 8371.85 seconds (approximately 2.33 hours). An average of 1.426 training samples and 0.178 training steps were processed per second. Additionally, the field `total_flos` quantifies the floating point operations conducted during training and reaches a total of 1.13×10^{18}. This value can serve as a basis for comparing computational cost across different experiments. The final `train_loss` of 0.151394 further confirms that the model has reached a satisfactory level of convergence. The Table 2 illustrates a steady and reliable fine-tuning process, both in terms of model performance and computational behavior. Since the `eval_loss` remains consistently lower than the `loss`, there is no indication of overfitting during fine-tuning. This suggests that the model generalizes well to the validation data and that the dataset does not lead the language model to memorize patterns from the training set.

Table 2
Training and Evaluation Metrics - First Iteration

loss	grad_norm	learning_rate	epoch	step	eval_loss	eval_runtime	eval_samples_per_second	eval_steps_per_second	train_runtime	train_samples_per_second	train_steps_per_second	total_flos	train_loss
0.4628	0.423335	0.000008	0.201005	100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.201005	100	0.250302	3.4477	NaN	0.580	NaN	NaN	NaN	NaN	NaN
0.3913	0.602808	0.000007	0.402010	200	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.402010	200	0.178318	3.4482	NaN	0.580	NaN	NaN	NaN	NaN	NaN
0.2828	0.552760	0.000006	0.603015	300	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.603015	300	0.083876	3.4461	2.902	0.580	NaN	NaN	NaN	NaN	NaN
0.2412	0.526656	0.000005	0.804020	400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.804020	400	0.053626	3.4463	NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.1687	0.382029	0.000004	1.004020	500	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.004020	500	0.046554	3.4558	2.894	0.579	NaN	NaN	NaN	NaN	NaN
0.1115	0.514400	0.000005	1.205025	600	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.205025	600	0.038296	3.4384	2.908	0.582	NaN	NaN	NaN	NaN	NaN
0.1144	0.509548	0.000005	1.406030	700	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.406030	700	0.035347	3.4435	2.904	0.581	NaN	NaN	NaN	NaN	NaN
0.0905	0.253039	0.000006	1.607035	800	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.607035	800	0.033022	3.4417	2.906	0.581	NaN	NaN	NaN	NaN	NaN
0.0829	0.587453	0.000007	1.808040	900	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.808040	900	0.032370	3.4413	2.906	0.581	NaN	NaN	NaN	NaN	NaN
0.0728	0.824504	0.000008	2.009045	1000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.009045	1000	0.031341	3.4438	2.904	0.581	NaN	NaN	NaN	NaN	NaN
0.0538	0.108236	0.000009	2.210050	1100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.210050	1100	0.032454	3.4467	2.901	0.580	NaN	NaN	NaN	NaN	NaN
0.0496	0.407553	0.000010	2.411055	1200	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.411055	1200	0.031820	3.4462	2.902	0.580	NaN	NaN	NaN	NaN	NaN
0.0450	0.655987	0.000011	2.611105	1300	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.611105	1300	0.030594	3.4404	2.907	0.581	NaN	NaN	NaN	NaN	NaN
0.0523	0.297614	0.000012	2.812060	1400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.812060	1400	0.028896	3.4579	2.892	0.578	8371.848	1.426	0.178	1.129752e+18	0.151394

Evaluation Metrics

The effectiveness of the fine-tuned language model was evaluated using standard text generation metrics including ROUGE, METEOR, and a custom Ansible Lint Score. These results reflect the model's ability to generate syntactically and semantically accurate Ansible code completions based on the given input context.

ROGUE. The model achieved high performance across all ROUGE variants. Specifically, the scores were:

- `ROUGE-1`: 0.9030
- `ROUGE-2`: 0.8751
- `ROUGE-L`: 0.8937

Figure 3
Training procedure curve - First Iteration

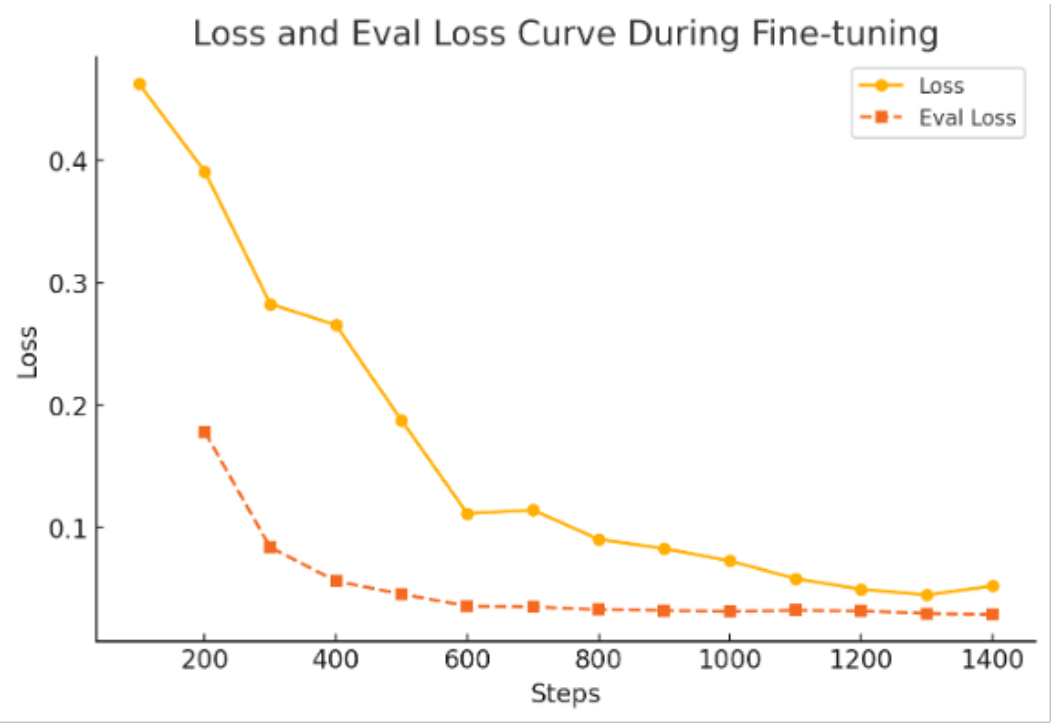


Illustration of the loss and evaluation loss curve.

- `ROUGE-Lsum`: 0.8968

These values indicate a strong overlap between the generated and reference outputs at both unigram and bigram levels (ROUGE-1 and ROUGE-2), as well as good preservation of longer subsequence structures (ROUGE-L and ROUGE-Lsum). This suggests that the model not only captures token-level accuracy but also maintains structural fidelity in the generated output, which is particularly relevant for code generation tasks where syntax and indentation patterns are critical.

METEOR. The score for the first iteration was 0.8738. Unlike ROUGE, METEOR takes into account synonymy and stemming, and is therefore well-suited for evaluating fluency and semantic correctness. The high METEOR value indicates that the model was able to generate outputs that are not only lexically similar but also semantically aligned with the reference data.

ChrF. The character-level evaluation metric produced a score of 97.27. This metric measures the exactness of the character sequences, which is crucial in the context of Ansible code,

where formatting and spacing errors can lead to execution failures. The accompanying parameters (`char_order: 6`, `word_order: 0`, `beta: 2`) reflect the weighting configuration for this score and were kept consistent across iterations for comparability.

Ansible Lint Score. To assess the syntactic correctness of the generated code, a custom evaluation based on Ansible Lint was introduced. The overall Ansible Lint Score for the first iteration was 0.56. This score reflects the proportion of generated outputs that complied with Ansible's style and structural guidelines. While not directly tied to semantic similarity, this metric is a practical indicator of how deployable the generated code would be in real-world infrastructure automation contexts. Within this iteration, the results are not yet satisfactory for the intended use case, as only approximately every second generated code section is syntactically correct.

Relation to Training Outcomes. The high ROUGE and METEOR scores are consistent with the loss reduction observed during training. As described in the previous section, the model showed a strong and consistent convergence, reaching a final training loss of 0.151 and an evaluation loss of 0.0289. These metrics, together with the evaluation scores, suggest that the model learned effectively from the dataset and was able to generalize well to unseen samples.

However, the moderate Ansible Lint Score indicates that there is still room for improvement in the syntactic correctness of the generated outputs. This could be due to the nature of the training data, which may have included stylistic inconsistencies, or due to limitations in how well the language model internalized formatting-specific rules. The reason for the syntactic issues in the generated code can be traced back to the dataset preprocessing step. During preparation, the input and output columns were concatenated into a single string using the format `"Instruction: {INPUT} Response: {OUTPUT}"`. While this structure was intended to clearly separate prompt and target for the model, it led to unintended consequences during generation. Specifically, the model learned to reproduce the formatting token `Response:` as part of its output, resulting in syntactically incorrect Ansible code segments.

The results of the first iteration indicate that the fine-tuned model was able to generate high-quality completions with strong lexical and semantic similarity to the reference data. This is supported by the consistently high values across all core NLP evaluation metrics. However, the Ansible Lint Score revealed only moderate syntactic correctness in the generated code. As a result, a second iteration

was conducted in order to improve code structure reliability and address the observed issues related to syntactic formatting and execution readiness.

An example output of the model for the following input:

```
  - name: Install ldap-client
  apt: name={{item}} state=present
  with_items:
   - libnss-ldap
   - libpam-ldap
   - nscd

- name: Create SSL directory for ldap
  file: path=/etc/ldap/ssl/certs state=directory

- name: Copy cert.pem
  copy: src={{sensu_ssl_crt_file}} dest=/etc/ldap/ssl/certs/ssl_crt.pem

- name: Copy key.pem
  copy: src={{sensu_ssl_key_file}} dest=/etc/ldap/ssl/certs/ssl_key.pem

- name: Configure ldap
  template: src=ldap.j2 dest=/etc/ldap/ldap.conf

- name: Update nsswitch.conf
  template: src=nsswitch.j2 dest=/etc/nsswitch.conf

- name: Update common-sessions
  template: src=common_sessions.j2 dest=/etc/pam.d/common-session

- name: Restart nscd
```

is:

```
- name: SAP Install Media Detect - Organize all files - Copy files to {{
    sap_install_media_detect_target_directory }}
```

```
  ansible.builtin.copy:
    src: "{{ sap_install_media_detect_source_directory }}/{{ line_item.
        file }}"
    dest: "{{ sap_install_media_detect_target_directory }}/{{ line_item.
        file }}"
    ...
    loop_control:
      loop_var: line_item
      label: "{{ line_item.file }}"
  when: sap_install_media_detect_source == 'remote_dir'
...
    - name: SAP Install Media Detect - Extract - Unpack archive {{
        line_item + '/xml' }} /*xml* to {{ line_item }} # noqa command-
        instead-of-module no-changed-when
Response:
# noqa command-instead-of-module no-changed-when
      ansible.builtin.command: "unzip {{ line_item + '/xml' }}/{{
          sap_install_media_detect_ftp_files }}*{{
          __sap_install_media_detect_system_id }}*xml* -d {{ line_item
          }}"
      when:
        - __sap_install_media_detect_register_extract_zip is changed
...
```

As illustrated in the example above, the generated output includes the string `Response:` within the Ansible code block. This formatting artifact is a direct result of the dataset preprocessing step, where input and output strings were concatenated using the template `"Instruction: {INPUT} Response: {OUTPUT}"`. While intended to signal the response portion during training, this pattern was learned by the model and reproduced during generation, leading to syntactically invalid YAML output. This issue was one of the main motivations for conducting a second fine-tuning iteration, in which the dataset formatting was revised to avoid injecting such artificial markers into the target output.

Second Iteration

The second iteration was carried out to address the syntactic limitations identified during the first run, particularly the moderate Ansible Lint Score and formatting issues caused by the dataset structure. As in the first iteration, the fine-tuning process was applied to the Phi-4 model using the same configuration parameters, including learning rate, batch size, and optimizer setup. The objective of this second run was to improve the syntactic correctness of the generated output while maintaining the strong semantic and lexical performance already achieved. The following results provide insights into the model's behavior during this repeated training cycle under unchanged conditions.

Runtime Training Results

Table 3 summarizes the training and evaluation metrics for the second iteration of the fine-tuning process. The same model architecture, dataset, and training configuration were used as in the first iteration to ensure comparability. As before, the table documents the evolution of loss values, evaluation performance, and runtime behavior at regular intervals throughout the training. The initial loss value is notably higher in the second iteration, starting at approximately 1.34. This may be attributed to different random initializations or variations in data shuffling. Over the course of the training, the loss decreases consistently and reaches a value of 0.0544 at step 1300. The final recorded training loss is 0.239186, which is higher than in the first iteration. This suggests that while the model learned effectively, convergence was not as pronounced. `grad_norm` values, where recorded, remain within a reasonable range and indicate stable updates. The learning rate follows the same schedule as in the first run, starting at 8×10^{-6} and gradually decaying over time. Evaluation was again conducted every 100 training steps. The `eval_loss` metric shows a continuous decline from 0.6727 to 0.0351, reflecting improved generalization to the validation data. Runtime-related metrics such as `eval_runtime`, `eval_samples_per_second`, and `eval_steps_per_second` remain consistent across all checkpoints, confirming stable evaluation throughput. The total training runtime for the second iteration was approximately 9141.06 seconds (approximately 2.54 hours). The average throughput reached 1.306 training samples and 0.163 training steps per second. The total number of floating point operations, `total_flos`, was approximately 1.38×10^{18}, which is in line with the first iteration and confirms the use of identical configurations.

Despite using the same configuration as in the first iteration, the final training loss is higher. This variation is likely due to differences in initialization or stochastic elements in the training

process. Nevertheless, both training and evaluation losses demonstrate a consistent downward trend, indicating a stable and effective learning process. In summary, the second iteration confirms the reproducibility of the training pipeline under identical settings, while also highlighting the impact of randomness on model performance.

Figure 4
Training procedure curve - Second Iteration

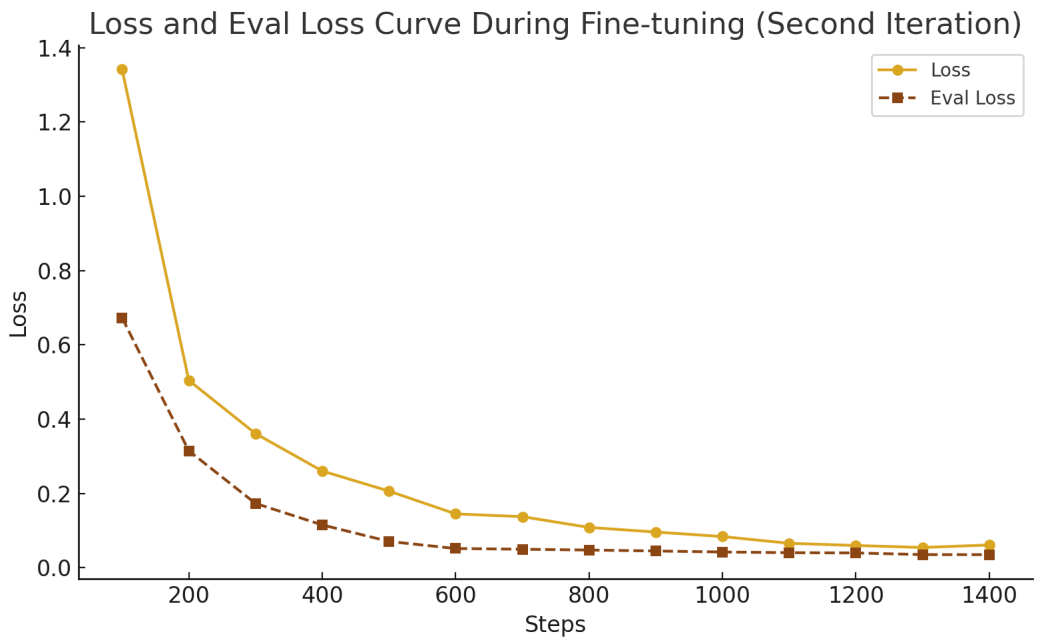


Illustration of the loss and evaluation loss curve.

An example from the second iteration demonstrates the improvements achieved by refining the dataset structure. Given the following input:

```
  - name: Install ldap-client
  apt: name={{item}} state=present
  with_items:
   - libnss-ldap
   - libpam-ldap
   - nscd

- name: Create SSL directory for ldap
  file: path=/etc/ldap/ssl/certs state=directory

```

```
- name: Copy cert.pem
  copy: src={{sensu_ssl_crt_file}} dest=/etc/ldap/ssl/certs/ssl_crt.pem

- name: Copy key.pem
  copy: src={{sensu_ssl_key_file}} dest=/etc/ldap/ssl/certs/ssl_key.pem

- name: Configure ldap
  template: src=ldap.j2 dest=/etc/ldap/ldap.conf

- name: Update nsswitch.conf
  template: src=nsswitch.j2 dest=/etc/nsswitch.conf

- name: Update common-sessions
  template: src=common_sessions.j2 dest=/etc/pam.d/common-session

- name: Restart nscd
```

the model generated the following output:

```
- name: SAP Install Media Detect - Organize all files - Copy files to {{
    sap_install_media_detect_target_directory }}
  ansible.builtin.copy:
    src: "{{ sap_install_media_detect_source_directory }}/{{ line_item.
        file }}"
    dest: "{{ sap_install_media_detect_target_directory }}/{{ line_item.
        file }}"
    remote_src: true
    owner: root
    group: root
    mode: '0755'
  loop: "{{ __sap_install_media_detect_fact_files_sapfile_results }}"
  loop_control:
    loop_var: line_item
    label: "{{ line_item.file }}"
  when: sap_install_media_detect_source == 'remote_dir'
```

```
14
15 - name: SAP Install Media Detect - Organize all files - Ensure archive
       extraction directories are absent
16   ansible.builtin.file:
17     path: "{{ line_item }}"
18     state: absent
19   loop: "{{ [__sap_install_media_detect_software_main_directory + '/'] |
         product(__sap_install_media_detect_fact_extraction_directories) |
         map('join') | list }}"
20   loop_control:
21     loop_var: line_item
22   when:
23     - sap_install_media_detect_extract_archives
24
25 - name: SAP Install Media Detect - Organize all files - Create archive
       extraction directories
26   ansible.builtin.file:
27     path: "{{ line_item }}"
28     state: directory
29     owner: root
30     group: root
31     mode: '0755'
32   loop: "{{ [__sap_install_media_detect_software_main_directory + '/'] |
         product(__sap_install_media_detect_fact_extraction_directories) |
         map('join') | list }}"
33   loop_control:
34     loop_var: line_item
35   when:
36     - sap_install_media_detect_extract_archives
```

Compared to the first iteration, this result demonstrates a clear improvement. The previously included formatting token `Response:`, which was a side effect of the dataset structure, is no longer present in the output. This indicates that the changes made to the data preprocessing pipeline—specifically

the removal of instruction markers from the training targets—successfully prevented the model from reproducing non-syntactic tokens during inference.

Moreover, the generated code remains structurally valid and adheres to Ansible syntax, suggesting that the model not only learned the desired patterns but also maintained syntactic discipline. However, the content of the output still shows signs of domain drift, as the generated task sequence is unrelated to the LDAP installation from the input and instead focuses on SAP installation media detection. This suggests that, while the syntactic formatting has improved, the semantic relevance between input and output could benefit from further refinement, potentially by adjusting the training dataset composition or prompt design.

In summary, this example confirms that the second iteration achieved its primary goal of eliminating formatting artifacts, while also highlighting the need for continued improvements in domain alignment and contextual coherence.

Table 3
Training and Evaluation Metrics – Second Iteration

loss	grad_norm	learning_rate	epoch	step	eval_loss	eval_runtime	eval_samples_per_second	eval_steps_per_second	train_runtime	train_samples_per_second	train_steps_per_second	total_flos	train_loss
1.3416	0.208290	0.000008	0.201005	100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.201005	100	0.672711	8.0082	1.249	0.624	NaN	NaN	NaN	NaN	NaN
0.5042	0.284414	0.000007	0.402010	200	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.402010	200	0.314670	6.7721	1.477	0.738	NaN	NaN	NaN	NaN	NaN
0.3611	0.368941	0.000006	0.603015	300	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.603015	300	0.172971	6.7802	1.475	0.737	NaN	NaN	NaN	NaN	NaN
0.2603	0.362303	0.000006	0.804020	400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	0.804020	400	0.115192	6.7667	1.478	0.739	NaN	NaN	NaN	NaN	NaN
0.2063	0.344204	0.000005	1.004020	500	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.004020	500	0.070670	6.7772	1.476	0.738	NaN	NaN	NaN	NaN	NaN
0.1448	0.427165	0.000005	1.205025	600	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.205025	600	0.051613	6.7557	1.480	0.740	NaN	NaN	NaN	NaN	NaN
0.1374	0.475965	0.000005	1.406030	700	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.406030	700	0.049662	6.7730	1.476	0.738	NaN	NaN	NaN	NaN	NaN
0.1084	0.255770	0.000006	1.607035	800	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.607035	800	0.047524	6.7696	1.477	0.739	NaN	NaN	NaN	NaN	NaN
0.0960	0.450637	0.000007	1.808040	900	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	1.808040	900	0.044999	6.7715	1.477	0.738	NaN	NaN	NaN	NaN	NaN
0.0838	0.743654	0.000008	2.009045	1000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.009045	1000	0.042229	6.7669	1.478	0.739	NaN	NaN	NaN	NaN	NaN
0.0657	0.134980	0.000009	2.210050	1100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.210050	1100	0.040389	6.7672	1.478	0.739	NaN	NaN	NaN	NaN	NaN
0.0597	0.396367	0.000010	2.411055	1200	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.411055	1200	0.039667	6.7754	1.476	0.738	NaN	NaN	NaN	NaN	NaN
0.0544	0.603925	0.000011	2.611105	1300	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.611105	1300	0.035206	6.7664	1.478	0.739	NaN	NaN	NaN	NaN	NaN
0.0611	0.258240	0.000012	2.812060	1400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.812060	1400	0.035067	6.7787	1.475	0.738	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	2.994975	1491	NaN	NaN	NaN	NaN	9141.0578	1.306	0.163	1.379025e+18	0.239186

Evaluation Metrics

The evaluation of the second iteration shows an overall improvement in syntactic correctness, while preserving the strong semantic and lexical performance observed in the first iteration. The training setup, model architecture, and evaluation metrics remained consistent to ensure comparability with the initial run.

ROUGE. In this iteration, the model again achieved high scores across all ROUGE variants:

- `ROUGE-1`: 0.9089
- `ROUGE-2`: 0.8815

- ROUGE-L: 0.8997
- ROUGE-Lsum: 0.9047

Compared to the first iteration, all ROUGE scores show a slight but consistent improvement. These results confirm that the model continues to generate completions that align closely with the reference outputs on a token level and retain coherent structure. The marginal increase in ROUGE-L and ROUGE-Lsum further indicates improved preservation of longer sequences, which is particularly relevant in the context of structured Ansible code.

METEOR. The METEOR score also increased slightly to 0.8851, reinforcing the observation that the model produces not only lexically accurate but also semantically meaningful completions. This aligns well with the downward trend observed in both training and evaluation loss curves, and supports the conclusion that the model generalized effectively to unseen data.

ChrF. The character-level score increased marginally from 97.27 to 97.84. While the difference is minor, it indicates a slight improvement in the model's ability to generate outputs with accurate character sequences. This is particularly relevant in code generation tasks where formatting issues, such as indentation or missing colons, may result in runtime errors.

Ansible Lint Score. One of the most relevant improvements in this iteration is reflected in the Ansible Lint Score. The overall score increased from 0.56 to 0.76, suggesting that a significantly larger portion of the generated outputs adhere to Ansible's syntactic and stylistic guidelines. This confirms that the model not only learned to complete the code more accurately, but also generated results that are closer to real-world, executable Ansible snippets.

Relation to Training Outcomes. The improvements in evaluation metrics are consistent with the training results. Although the second iteration began with a higher initial loss, the model exhibited stable convergence behavior and reached a final training loss of 0.2392. Evaluation loss decreased continuously, indicating that the model did not overfit and maintained generalization capabilities. The improved Ansible Lint Score validates the effectiveness of repeating the fine-tuning procedure and confirms that the second iteration better meets the practical requirements of the use case.

Overall, the second iteration demonstrated measurable improvements in both linguistic and structural quality. While the gains in ROUGE and METEOR were incremental, the notable increase in Ansible Lint Score confirms that the syntactic quality of the generated code was significantly enhanced. These results show that repeating the fine-tuning with the same configuration can lead to better generalization and more reliable output — particularly in code generation tasks where correctness is critical.

Discussion

This chapter reflects on the final results of the evaluation and places them into context. First, the model performance will be discussed in light of the applied evaluation metrics, followed by a comparison of the outcomes for the different iterations. Subsequently, the chapter outlines potential future directions and highlights current limitations of the approach, providing suggestions for further development and application.

Comparison of the Results

As seen in the Results Chapter, the fine-tuning process was conducted in two independent iterations using identical model configurations, datasets, and training parameters. This controlled setup allows for a direct comparison of training behavior and final performance outcomes between both runs. While the overall trends observed in both iterations were similar, several differences in the metrics and final results are noteworthy and merit further analysis.

The first iteration demonstrated a steady and relatively smooth convergence, with the training loss decreasing from 0.4628 to 0.0450 by step 1300, and a final loss of 0.151394. In contrast, the second iteration began with a higher initial loss of 1.3416 and converged more slowly, reaching a training loss of 0.0544 at step 1300 and a final value of 0.239186. This suggests a less efficient learning process in the second run. A explanation for this discrepancy is the inherent stochasticity of model training, particularly in weight initialization and data shuffling. Despite identical configurations, small variations in random seeds or batch composition can impact the trajectory of learning. The higher starting loss in the second iteration indicates that the initial model state was further from a useful solution, which may have caused the optimizer to follow a less optimal path through the loss landscape. Evaluation metrics across both runs confirm that the model improved its generalization capabilities with increasing training steps. In both iterations, the evaluation loss decreased steadily from the initial to the final checkpoints. The first iteration achieved a minimum evaluation loss of 0.0289, while the second iteration reached a slightly higher minimum of 0.0350. Although the difference is small, it aligns with the trend observed in the training loss and reinforces the hypothesis that the second iteration's learning trajectory was less efficient overall. However, the continuous improvement in both cases suggests that the model architecture and training setup were fundamentally sound and capable of generalization. The recorded `grad_norm` values further

support the assessment of training stability. Across both iterations, the gradients remained within a stable range, what showcases that the optimization process did not encounter extreme fluctuations. The learning rate decay schedule was applied identically in both runs and contributed to the observed convergence. Since neither the gradient magnitude nor learning rate differed between runs, they are unlikely to be responsible for the performance gap. Runtime-related metrics such as evaluation throughput and total training time were similar across both runs. The first iteration required approximately 8371 seconds, while the second lasted 9141 seconds. The slightly increased duration in the second run can be attributed to natural variations in system resource availability or evaluation timing. Floating point operations (`total_flos`) remained consistent, with 1.13×10^{18} and 1.38×10^{18} recorded in the first and second iterations, respectively. This confirms that the computational workload remained equivalent.

The comparison highlights the importance of repeatability in model training and the impact of stochastic components on final outcomes. Even under controlled conditions, differences in initialization and random sampling can lead to noticeable variation in training efficiency and final performance. These effects are expected and well-documented in machine learning literature, particularly in fine-tuning of language models. Despite the variability, both iterations show consistent trends in loss reduction and evaluation performance, validating the robustness of the training pipeline. The second iteration, while slightly less efficient, still converged and yielded a usable model, underlining the overall reliability of the setup. In conclusion, while the second iteration exhibited higher initial and final losses compared to the first, the observed differences fall within an acceptable range and are likely due to stochastic variation. No evidence was found to suggest structural issues in the model configuration or training procedure. These findings emphasize the value of multiple training runs and performance averaging when evaluating model quality.

This section directly contributes to answering Research Question 3, which investigates to what extent the fine-tuned language model meets the requirements in terms of performance, accuracy, and applicability. Following the dataset preparation and fine-tuning phases described in Research Questions 1 and 2, two independent fine-tuning iterations were conducted using identical model configurations and training setups. The goal was to evaluate the consistency and reliability of the training pipeline, as well as the quality of the generated Ansible code in both linguistic and functional dimensions.

Future outlook

In the following section the limitations and future outlook will be discussed for this research. This thesis gives different opportunities to conduct a following research based on the results of this thesis. The presented approach and evaluation methods can serve as a foundation for refining model architectures or extending the dataset to additional configuration management domains. Furthermore, the integration of runtime validation or real-world deployment testing could offer valuable extensions to the existing evaluation framework.

As mentioned earlier, the dataset used in this thesis was generated from various GitHub repositories containing Ansible Playbooks that implement specific use cases. Consequently, the code within this dataset inherits limitations and dependencies related to code quality and security practices of the respective repository maintainers. While efforts were made to mitigate these risks — such as selecting well-known and actively maintained repositories — the model may still replicate code patterns that reflect these underlying issues. Furthermore, the current evaluation lacks a semantic assessment by professional Infrastructure-as-Code (IaC) developers. This limitation could be addressed in future research through expert interviews and structured human evaluation methodologies.

Additionally, for future work, the fine-tuned model can be embedded into a web application with a simple user interface, where users can enter specific prompts and receive corresponding model-generated responses. The SAP Business Technology Platform (SAP BTP) offers a suitable environment for developing such a Fiori-based web application. To achieve this, the fine-tuned model must be encapsulated within a Docker container, equipped with a FastAPI or Flask-based service for handling inference requests. Subsequently, the containerized model can be deployed to SAP AI Core, which enables the exposure of a secure REST endpoint. The Fiori application can then interact with this endpoint by issuing HTTP requests to obtain the model output. This setup provides a practical basis for extending the solution for development and demonstration purposes within the SAP UCC Chair. An advantage of this deployment approach is that all data requests to the model remain within the controlled SAP BTP infrastructure, and are not processed or stored by third-party providers. This is particularly relevant in scenarios where the prompts or generated outputs may contain sensitive information, such as internal SAP server configurations or confidential automation procedures. By self-hosting the model within SAP AI Core, potential data privacy concerns associated with external inference services are effectively mitigated.

Conclusion

The objective of this thesis was to develop a customized and efficient approach for generating Ansible code using a fine-tuned large language model. The research aimed to address the current lack of automation support tools within the SAP UCC environment, specifically focusing on reducing manual effort and increasing the accuracy of Ansible playbook creation. For this purpose, the thesis introduced a structured methodology based on the principles of Design Science Research (DSR) and applied it in a real-world context to achieve practical relevance.

In the beginning, a detailed motivation and problem statement were outlined, emphasizing the increasing need for reliable IT automation in enterprise environments. Based on this, three research questions were defined, which guided the thesis throughout all development and evaluation phases.

To address the first research question, a high-quality dataset was created using the Ansible Content Parser. This tool enabled the extraction of YAML-based Ansible playbooks from open-source repositories. A multi-step process was applied to ensure data quality and consistency, which included repository selection, license inspection, deduplication, and cleaning procedures. The final dataset was structured as input-output pairs that were used as training data for the fine-tuning process.

In the second phase, the Phi-4 model was fine-tuned using Low-Rank Adaptation (LoRA) through the Unsloth library. The decision to use Phi-4 was based on its ability to provide LLM-level performance while maintaining computational efficiency. The fine-tuning process included preprocessing steps, tokenization, and evaluation-aware label generation. Several iterations of training were conducted on a managed cloud GPU infrastructure to ensure reproducibility and scalability.

To answer the third research question, the fine-tuned model was evaluated using a set of established and custom metrics. These included ROUGE, METEOR, and CHRF scores to measure text-based similarity, as well as a custom ansible-lint-based metric to assess syntactic validity and adherence to Ansible best practices. The results of both training iterations showed significant improvements in code generation quality, particularly in generating semantically consistent and executable YAML blocks.

In summary, the thesis demonstrated that a domain-specific fine-tuning approach can significantly enhance the performance of LLMs in infrastructure-as-code scenarios. The developed solution bridges the gap between general-purpose language models and the highly specific requirements of enterprise automation systems like SAP. It was shown that through careful dataset construction and tailored fine-tuning strategies, high-quality and context-aware Ansible code can be generated, leading to tangible benefits in terms of development speed, accuracy, and maintainability.

Appendix

Jupyter Notebook Code

in this step, we are installing all necessary libraries

```
!pip install torch torchvision torchaudio --index-url https://
    download.pytorch.org/whl/cu124
!pip install tensorflow
!pip install optuna
!pip install triton --index-url https://download.pytorch.org/whl/
    cu124
!pip install --no-deps trl peft accelerate bitsandbytes
!pip install "unsloth[colab-new] @ git+https://github.com/unslothai/
    unsloth.git"
!pip install --force-reinstall --no-cache-dir --no-deps xformers --
    index-url https://download
!pip install --force-reinstall --no-cache-dir --no-deps "unsloth[
    cu124-torch260] @ git+https://github.com/unslothai/unsloth.git"
!pip install ansible-lint
```

initialize and load the phi-4 model from unsloth (or from our drive, when we want to load our fine-tuned model)

```
from unsloth import FastLanguageModel
import torch

max_seq_length = 7000
load_in_4bit = True
STRING_MODEL = "unsloth/Phi-4"

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name=STRING_MODEL,
    load_in_4bit=load_in_4bit,
    max_seq_length=max_seq_length
)
```

```
13
14      def model_init():
15          return model
16
17      print("CUDA Available:", torch.cuda.is_available())
18      print("CUDA Device:", torch.cuda.get_device_name(0) if torch.cuda.
            is_available() else "CPU Only")
```

LORA Adapters for more efficiency –> standared values are used

```
1       model = FastLanguageModel.get_peft_model(
2           model,
3           r=16,
4           target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
5                           "gate_proj", "up_proj", "down_proj"],
6           lora_alpha=16,
7           lora_dropout=0,
8           bias="none",
9           use_gradient_checkpointing="unsloth",
10          random_state=3407,
11          use_rslora=False,
12          loftq_config=None
13      )
```

Splitting the dataset and creating a function. Ratio is 70-15-15

```
1       !pip install scikit-learn
2
3       from sklearn.model_selection import train_test_split
4       from datasets import Dataset, DatasetDict
5
6       def split_dataset(dataset, train_ratio=0.7, val_ratio=0.15,
            test_ratio=0.15, seed=42):
7           assert train_ratio + val_ratio + test_ratio == 1.0
8           train_data, temp_data = train_test_split(dataset, test_size=1 -
                train_ratio, random_state=seed)
```

```
        val_size = val_ratio / (val_ratio + test_ratio)
        val_data, test_data = train_test_split(temp_data, test_size=1 -
            val_size, random_state=seed)
        return train_data, val_data, test_data
```

Loading and Preprocessing the dataset

```
    from unsloth.chat_templates import get_chat_template
    from datasets import load_dataset

    def preprocess_function(entry):
        instruction = entry["input"]
        response = entry["output"]
        text = f"\n{instruction}\r\n{response}"
        encoding = tokenizer(text, truncation=True)
        encoding["labels"] = [
            token if token != tokenizer.pad_token_id else -100
            for token in encoding["input_ids"]
        ]
        return encoding

    dataset = load_dataset("FurkanGuerbuez/ansible_training", split="
        train")
    dataset_split = dataset.train_test_split(test_size=0.3, seed=42)
    validation_test_split = dataset_split["test"].train_test_split(
        test_size=0.5, seed=42)
    split_dataset = {
        "train": dataset_split["train"],
        "validation": validation_test_split["train"],
        "test": validation_test_split["test"]
    }

    train_data_token = split_dataset['train'].map(preprocess_function,
        remove_columns=['data_source_description'])
```

```
    val_data_token = split_dataset['validation'].map(preprocess_function,
         remove_columns=['data_source_description'])
    test_dataset = split_dataset['test'].map(preprocess_function,
        remove_columns=['data_source_description', 'input'])
```

Metrics for the Trainer, with the eval dataset

```
    from trl import SFTTrainer
    from transformers import TrainingArguments, DataCollatorForSeq2Seq
    from unsloth import is_bfloat16_supported

    trainingargs = TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=100,
        num_train_epochs=3,
        logging_steps=100,
        eval_steps=100,
        eval_strategy="steps",
        learning_rate=8e-5,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported,
        optim="adamw_8bit",
        lr_scheduler_type="linear",
        seed=3407,
        output_dir="outputs",
        overwrite_output_dir=True,
        report_to="none"
    )

    trainer = SFTTrainer(
        model=model,
        tokenizer=tokenizer,
        eval_dataset=val_data_token,
        train_dataset=train_data_token,
```

```
        max_seq_length=max_seq_length,
        data_collator=DataCollatorForSeq2Seq(tokenizer=tokenizer),
        args=trainingargs,
    )
```

Verifying if masking is done correctly

```
tokenizer.decode(trainer.train_dataset[10]
["input_ids"])
space = tokenizer(" ", add_special_tokens = False).input_ids[0]
tokenizer.decode([space if x == -100 else x for x in trainer.
    train_dataset[5]["labels"]])
```

Here the train dataset will be prepared for evaluation. The generated outputs by the model will be saved in a json file.

```
import torch
import random
import json
from transformers import GenerationConfig, TextStreamer
# Prepare the model for inference using Unsloth
from unsloth import FastLanguageModel
FastLanguageModel.for_inference(model)

input_texts = split_dataset['test']['input']

prompts = []
for input_text in input_texts:
    prompt = {"role": "user", "content": input_text}
    prompts.append(prompt)

#restrict test dataset, because of runtime duration
prompt_text = prompts[0]["content"]
num_random_prompts = 500

random_indices = random.sample(range(len(prompts)), num_random_prompts)
```

```
21
22  decoded_texts = []
23
24  print(prompt_text)
25
26  for index in random_indices:
27      # Convert text to tokens
28      prompt_text = prompts[index]["content"]
29      inputs = tokenizer(prompt_text, return_tensors="pt").to("cuda")
30      # Define generation config
31      generation_config = GenerationConfig(
32      #    temperature=0.9,       # More controlled output
33      #    top_p=0.9,             # Nucleus sampling for variety
34      #    top_k=40,              # Limits randomness
35          max_new_tokens=1800,    # Ensures proper output length
36          do_sample=True,        # Enables sampling instead of greedy
                decoding
37      #    pad_token_id=tokenizer.pad_token_id,
38      #    eos_token_id=tokenizer.eos_token_id,  # Stops generation
            properly
39      )
40
41      # text streamer for better readability
42      text_streamer = TextStreamer(tokenizer)
43      outputs = model.generate(**inputs, streamer = text_streamer,
            generation_config=generation_config)
44
45      # Generate response
46      decoded_text = tokenizer.decode(outputs[0], skip_special_tokens=
            True)
47
48      # Collect decoded texts in a list
49      decoded_texts.append(decoded_text)
```

Saving the generated outputs as a file

```
    with open("generated_outputs_2.json", "w") as f:
    json.dump(decoded_texts, f, indent=4)  # indent for better
        readability
```

run ansible lint on the generated file, and extract the yaml code sections! This step will ensure that the sections generated are syntactical correct!

```
import re
import tempfile
import numpy as np

def extract_yaml_sections(input_text):
    # This regex looks for "```yaml" followed by a YAML block until the
        next "```" or end of string
    yaml_pattern = re.compile(r"```yaml\n(.*?)\n```", re.DOTALL)

    # Extract all matches
    yaml_sections = yaml_pattern.findall(input_text)

    if yaml_sections is None:
        return "Empty"

    return yaml_sections

yaml_sections = decoded_texts

def calculate_ansible_lint_scores(yaml_sections):
    scores = []  # Initialize an empty array to store scores

    for idx, section in enumerate(yaml_sections, 1):
        with tempfile.NamedTemporaryFile(mode="w", suffix=".yml",
            delete=False) as tmpfile:
            tmpfile.write(section)
            tmpfile_path = tmpfile.name
```

```
            # Run ansible-lint and capture output
            result = !ansible-lint {tmpfile_path}

            # checks the result string on "X failure(s)" and writes the
                X into failures variable
            match = re.search(r"(\d+)\s+failure\(s\)", output)
            failures = int(match.group(1)) if match else 0

            # Scoring: sore is 1 if 0 failures, score is 0 otherwise
            score = 1 if failures == 0 else 0

            print("SCORE: ", score)
            scores.append(score)

    return scores

# Calculate scores for all yaml_sections
scores_array = calculate_ansible_lint_scores(yaml_sections)

# Evaluate the overall average score
overall_score = np.mean(scores_array)

print(f"Ansible Lint Scores: {scores_array}")
print(f"Overall Ansible Lint Score: {overall_score}")
```

Evaluation metrics

```
!pip install sacrebleu evaluate rouge_score meteor chrf

from evaluate import load
rouge = load("rouge")
meteor = load("meteor")
chrf = load("chrf")

# Decode and truncate predictions
```

```
    modified_predictions = []
    for pred, ref in zip(decoded_predictions, decoded_references):
        ref_length = len(ref)
        modified_predictions.append(pred[:ref_length])

    rouge_results = rouge.compute(predictions=modified_predictions,
        references=decoded_references)
    meteor_results = meteor.compute(predictions=modified_predictions,
        references=decoded_references)
    chrf_results = chrf.compute(predictions=modified_predictions,
        references=decoded_references)

    print(f'Model - ROUGE: \n{rouge_results}\n')
    print(f'Model - METEOR: \n{meteor_results}\n')
    print(f'Model - chrF: \n{chrf_results}\n')
```

Now we want to train the model with the configurations

```
    trainer_stats = trainer.train()
```

Post training steps: Output the training statistics and save the model

```
    trainer.save_model("/content/drive/MyDrive/
        finetuned_phi4_second_iteration")

    import pandas as pd
    df = pd.DataFrame(trainer.state.log_history)

    df.to_csv('ansible_playbook_predictions.csv', index=False)

    pd.DataFrame(trainer.state.log_history)
```