EVALUATING AND ENHANCING LLMS FOR DEEP LEARNING CODE GENERATION WITH DL-BENCH

ALIREZA DAGHIGHFARSOODEH

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

SEPTEMBER 2025

© Alireza Daghighfarsoodeh, 2025

Abstract

Large Language Models (LLMs) have recently demonstrated remarkable capabilities in automated code generation, yet their performance on domain-specific tasks such as deep learning (DL) pipelines remains underexplored. This thesis addresses this gap by introducing **DL-Bench**, the first comprehensive benchmark dedicated to evaluating LLMs on DL-specific code generation. DL-Bench comprises 520 carefully curated function-level tasks spanning all stages of the machine learning workflow, including data pre- and post-processing, model construction, training, inference, and evaluation, and is systematically categorized by pipeline stage, task type, and input modality. This fine-grained design enables detailed performance analysis and exposes unique challenges of DL code generation, such as tensor shape mismatches, framework-specific errors, and brittle reliance on phrasing.

Building on this benchmark, we further investigate robustness strategies for LLMs by proposing a **prompt mutation pipeline** combined with **dual execution agreement**. The pipeline systematically generates semantically equivalent prompt variations through lexical, grammatical, and naming transformations, which are then paired with model-generated test cases to diversify candidate solutions. Using a dual agreement framework, correct solutions are identified by their consistent success across test suites, mitigating common misinterpretations. To validate this approach, we evaluate three state-of-the-art LLMs, O4-Mini, DeepSeek R1 Basic, and Gemini 2.5 Pro, exclusively on DL-Bench. Results show that while baseline performance on DL-Bench is substantially lower than on general-purpose

benchmarks, prompt mutations consistently yield measurable improvements (up to +2.9% pass@1), demonstrating their value in uncovering alternative correct solutions.

Overall, this thesis makes three key contributions: (i) the release of DL-Bench as a domain-specific, fine-grained benchmark for DL code generation, (ii) a systematic analysis of LLM weaknesses in DL contexts supported by a taxonomy of mutation effects, and (iii) the design and evaluation of a mutation-based dual agreement framework that enhances LLM reliability. These contributions provide both practical evaluation tools and methodological insights for advancing LLMs in specialized scientific programming domains. Future directions include scaling DL-Bench with multi-modal tasks, maintaining it as a live benchmark to track recency effects, and incorporating broader metrics such as code efficiency and maintainability.

Acknowledgements

This thesis would not have come to fruition without the support, encouragement, and kindness of many extraordinary individuals, to whom I am deeply indebted.

Above all, I want to thank my parents for their unconditional love and steadfast support. No matter where life has taken me, their belief in me has never wavered. Their encouragement has given me the confidence to pursue my goals wholeheartedly, and their presence in every chapter of my life is something I cherish deeply.

I am truly thankful to my supervisor, Professor Hung Viet Pham, whose guidance, patience, and support have shaped my academic journey in profound ways. His mentorship has played a key role in both my intellectual and personal development, and I feel privileged to have worked under his supervision. I am also grateful to Dr. Hadi Hemmati, a member of my supervisory committee, for his thoughtful feedback and valuable advice, which have been vital to the success of this research.

Throughout my time as a master's student, I've had the opportunity to work with and learn from an inspiring group of peers. I would like to extend my heartfelt thanks to Jiho Shin, Mohammad Mahdi Mohajer, Melika Sepidband, Hamed Taherkhani, Ramin Sharifi, Fatemeh Seyyed Dabbaghi, Danial Kordmodanlu, Nima Shiri Harzevili, Mohammad Abdollahi, and Chung-Yu Wang. Their friendship, support, and collaboration have made this journey all the more meaningful and rewarding.

Table of Contents

Abstract **ii**

Acknowledgements **iv**

Table of Contents **v**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Hypothesis 2

1.3 Thesis Overview 3

1.4 Thesis Contribution 3

1.5 Thesis Organization 4

2 DL-Bench: Code Generation Benchmark for Deep Learning Specific Tasks **5**

2.1 Introduction & Background 5

2.2 Approach 8

2.3 Dataset Statistics 10

2.3.1 Semantic Diversity Analysis . . . 12
2.3.2 Data Categories and Labeling Schema . . . 12
2.4 Experimental Setup . . . 14
2.5 Results . . . 15
2.5.1 Distinctive Challenges . . . 16
2.5.2 Performances of SOTA LLMs on DL-Bench and DS-1000 . . . 18
2.6 Qualitative Analysis . . . 21
2.6.1 Challenges in DL-Specific Code . . . 22
2.6.2 Bug Taxonomy . . . 25
2.7 LLM Bug Types and DL-specific Subtypes . . . 26
2.8 Distribution of Failures in Generated DL Code . . . 33
2.8.1 Example of bugs . . . 34
2.9 Discussion: DL-Bench in practice . . . 36
2.10 Discussion . . . 38
2.10.1 Data Leakage Considerations . . . 38
2.10.2 Benchmark Comparison and Evaluation Criteria . . . 39
2.11 Related Works . . . 40
2.12 Conclusion . . . 42

3 Mutation Based Code Generation 43

3.1 Introduction . . . 43
3.2 Background . . . 46
3.3 Approach . . . 48
3.3.1 TEST CASE GENERATION . . . 51
3.4 Experimental Setup . . . 52
3.4.1 Models . . . 52

3.4.2 Benchmarks . . . 52
3.4.3 Prompt Mutation Pipeline . . . 53
3.4.4 Research Questions . . . 54
3.5 Result . . . 56
3.5.1 Performance of Mutating . . . 56
3.5.2 Individual Mutation Impact . . . 58
3.5.3 Combination Impact . . . 59
3.6 Limitation & Threats to validity . . . 60
3.7 Related work . . . 61
3.7.1 Code Generation . . . 61
3.7.2 Prompt Mutation . . . 62
3.8 Conclusion . . . 63

4 Conclusions and Future work **65**
4.1 Thesis Findings and Contributions . . . 65
4.2 Future Work . . . 67

Bibliography **69**

List of Tables

2.1 Code generation benchmarks 6

2.2 Pass@1 scores for various SOTA LLM 18

2.3 Variance across five runs 18

2.4 Pass@k (%) on DL-Bench at temperature = 0.3. 19

2.5 Live DL-Bench results 19

2.6 Static code quality 20

2.7 Pass@1 across different categories 22

2.8 Distribution of bugs 33

2.9 Distribution of stages 36

2.10 Prompting techniques with DL-Bench's insight. 37

3.1 Examples of mutation types 48

3.2 Manual evaluation on mutations 56

3.3 Pass@1 Results 56

List of Figures

2.1 DL-Bench construction procedure 8
2.2 Template of generating prompt from code 9
2.3 Distribution of code samples in each category 10
2.4 An example prompt for Pre/Post processing 10
2.5 An example prompt for Model Construction 11
2.6 Example of General Task 11
2.7 An example of Classification Task. 11
2.8 Embeddings Projection 16
2.9 Taxonomy of bugs in DL generated code 24
2.10 General vs DL-specific bugs 24
2.11 Mismatching data shapes 34
2.12 Incorrect processing of parameters 34
2.13 Wrong usage of a third-party library. 35

3.1 Example of Mad-Gen output 44
3.2 Overview of Mad-Gen 48
3.3 Wrong Semantic 55
3.4 Preserve Meaning Mutation 55
3.5 Failed Mutation 55

3.6 Successful mutation 55

Chapter 1

Introduction

1.1 Motivation

Deep learning (DL) has become a cornerstone of modern artificial intelligence, powering applications in computer vision, natural language processing, recommender systems, and beyond. However, developing DL systems is challenging due to the complexity of their workflows, the variety of tasks and input modalities, and the need to integrate multiple libraries and frameworks. Large Language Models (LLMs) have recently emerged as powerful tools for code generation, offering the potential to reduce development time and assist non-specialist developers.

Despite their promise, existing benchmarks for evaluating LLM-generated code, such as DS-1000 Lai et al. [2023] and MLE-bench Chan et al. [2024], have critical limitations when applied to DL-specific scenarios. They typically focus on small pre/post-processing snippets, lack comprehensive coverage of all stages in the DL pipeline, and omit systematic categorization by task type or data modality. This limits their ability to reveal nuanced performance differences or provide insights into model weaknesses.

To address these gaps, **DL-Bench** introduces a function-level benchmark designed specifically for DL code generation. DL-Bench categorizes functions along three axes, pipeline stage, DL task type, and input data type, enabling fine-grained performance analysis and a more realistic evaluation of LLM capabilities. Its results show that current state-of-the-art LLMs achieve much lower pass@1 scores on DL-Bench than on general ML benchmarks, underscoring the difficulty of DL-specific code generation.

In parallel, recent methods such as prompt mutation with dual execution Chen et al. [2022] offer complementary strategies for improving robustness in code generation. While not the primary focus of this thesis, the dual execution methodology will be discussed as a potential future enhancement to DL-Bench-based evaluation.

1.2 Research Hypothesis

This thesis is guided by the hypothesis that:

> A DL-specific benchmark that categorizes functions by pipeline stage, task type, and input modality, as implemented in DL-Bench, provides a more challenging and representative evaluation of LLM code generation than general ML benchmarks, revealing weaknesses and guiding targeted improvements.

Key expectations include:

1. DL-Bench will expose significant performance gaps between general Data Science code generation and DL-specific code generation.

2. Categorization will enable identification of stage-, task-, and data-type–specific weaknesses in LLMs.

3. Insights from DL-Bench can guide the design of more robust evaluation and prompting strategies, including those inspired by dual execution.

1.3 Thesis Overview

This thesis focuses primarily on DL-Bench, presenting its motivation, construction methodology, categorization scheme, and evaluation results. The benchmark is positioned as a tool to:

- Fill the gap left by general-purpose benchmarks by providing DL-specific, function-level evaluation.
- Enable fine-grained analysis of LLM strengths and weaknesses.
- Support future improvements in code generation through better diagnostic capabilities.

In addition, the dual execution approach is briefly examined as a complementary method to enhance robustness in future extensions of DL-Bench.

1.4 Thesis Contribution

The contributions of this thesis are:

1. **Comprehensive DL-specific benchmark:** A detailed presentation and evaluation of DL-Bench as a challenging, semantically distinct dataset for DL code generation.
2. **Fine-grained categorization:** A three-axis classification of benchmark instances, pipeline stage, task type, and input data type, enabling detailed performance analysis.
3. **Empirical evaluation:** A comparative study of state-of-the-art LLMs on DL-Bench, highlighting significant challenges and areas for improvement.
4. **Forward-looking discussion:** Consideration of integrating DL-Bench with robustness-enhancing methods such as prompt mutation and dual execution for future work.

1.5 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** presents DL-Bench in detail, including its motivation, construction process, categorization scheme, and quantitative and qualitative evaluation results.
- **Chapter 3** introduces the dual execution approach, describing its methodology, potential integration with DL-Bench, and its role in improving robustness in code generation.
- **Chapter 4** outlines future work, discussing how DL-Bench could be extended with prompt mutation, dual execution, and other robustness-oriented strategies.

Chapter 2

DL-Bench: Code Generation Benchmark for Deep Learning Specific Tasks

2.1 Introduction & Background

In recent years, machine learning (ML) and deep learning (DL) have advanced significantly and have been integrated into various fields Hordri et al. [2016], Kamilaris and Prenafeta-Boldú [2018], Gamboa [2017]. DL coding has its challenges Arpteg et al. [2018], and because of its widespread use, many DL systems are developed by domain experts who are often not software developers Park et al. [2021], Singaravel et al. [2020], which amplifies the problems even more.

Recently, with the rise of Large Language Models (LLMs) such as ChatGPT, LLMs are considered among the best solutions for coding tasks Wang et al. [2021], Feng et al. [2020], Achiam et al. [2023] as demonstrated by numerous code generation benchmark datasets. However, until recently, as shown in Table 2.1, most of these benchmarks focused on general

Table 2.1: Code generation benchmarks (SO-Stack Overflow, EM-exact match, ES-edit similarity, CS-competition specifics).

Benchmark	Size	Language	Level	Source	Metrics
Human EvalChen et al. [2021]	164	Python	Function	Manual	Pass@k
MBPPAustin et al. [2021]	974	Python	Function	Manual	Pass@k
APPSHendrycks et al. [2021]	10000	Python	Function	Manual	Pass@k
CoderEvalYu et al. [2024]	460	Python-Java	Function	GitHub	Pass@k
EvoCodeBenchLi et al. [2024]	275	Python	Function	GitHub	Pass@k
RepoEvalZhang et al. [2023]	1600	Python	Function	GitHub	EM,ES
ClassEvalDu et al. [2023]	100	Python	Class	Manual	Pass@k
BigCodeBenchZhuo et al. [2024]	1140	Python	Function	Manual	Pass@k
LiveCodeBenchJain et al. [2024]	511	Python	Function	Code competition	Pass@k
NumpyEval Zan et al. [2022]	101	Python	Function	SO	Pass@k
PandasEval Zan et al. [2022]	101	Python	Function	SO	Pass@k
DSP Chandel et al. [2022]	1119	Python	Code-Cell	Jupyter Notebook	Pass@k
JuICeAgashe et al. [2019]	1.52Mil	Python	Code-Cell	Jupyter Notebook	EM
DS-1000Lai et al. [2023]	1000	Python	Statement	SO	Pass@k
MLE-BenchChan et al. [2024]	75	Python	Workflow	Kaggle	CS
AICoderEvalXia et al. [2024]	492	Python	Workflow	Web	Pass@k
DL-Bench	**520**	**Python**	**Function**	**GitHub**	**Pass@k**

programming tasks. Shin et al.Shin et al. [2023] are the first to underline the distinct challenges of generating ML/DL code compared to general code. However, their generated code evaluation relies on less suitable similarity metrics as very different code snippets can have the same functionality, and a small change in a code snippet can drastically alter its semantics.

A few datasets, such as MLE-bench Chan et al. [2024] or AICoderEval Xia et al. [2024], offer examples of ML-specific code generation on the ML workflow level, which does not fit the LLM's usage, where developers need help with generating specific functions. These benchmarks often evaluate LLMs based on the final ML system's performance (e.g., accuracy, F1, etc.). Among these, DS-1000 Lai et al. [2023] provides small (a few lines) ML-specific code snippets, primarily focused on pre/post-processing tasks. It also does not provide any categorizations, such as ML tasks, DL phases, or input types, which could provide valuable insights for code generation improvement.

To address these gaps, we introduce DL-Bench, a novel dataset designed to benchmark DL-specific code generation at a functional level. Each entry includes the code generation prompt, the ground-truth code at the function level, and an extensive set of unit tests. Unlike DS-1000 and MLE-bench, DL-Bench provides a more comprehensive and diverse set of function-level samples that cover all phases in the DL pipeline for various ML tasks and input data types.

These entries are categorized into three aspects: (1) *The DL/ML pipeline stages*: pre/post-processing, model construction, training, inference, and evaluation, (2) *The DL/ML tasks*: classification, object detection, image segmentation, time-series prediction, recommendation, and regression, and (3) *The input data types*: text, image, and array. These categorizations enable a more in-depth evaluation and analysis of future techniques in generating DL-specific code.

We qualitatively compare DL-Bench with its most related benchmarks (DS-1000 and AICoderEval) by examining four aspects of dataset divergence. First, we show that DL-Bench occupies a semantically distinct region of the embedding space, hence contains novel problem domains and different solution patterns. Second, we reveal that DL-Bench emphasizes DL constructs heavily. Third, we demonstrate that DL-Bench problems require more complex solutions, hence are more challenging for LLMs. Finally, state-of-the-art LLMs (e.g., O3-Mini, DeepSeak-V3) struggle to solve DL-Bench's problems with significantly lower **29.5%** pass@1 score on DL-Bench than on DS-1000 (53.3%).

Furthermore, our qualitative analysis indicates that the difficulty of generating code varies significantly across categories. For example, O3-Mini reaches an accuracy of 41.1% for pre/post-processing tasks but only 31.6% for model construction. The pass@1 rate varies even more among task types, ranging from 54.6% for recommendation tasks to 27.9% for segmentation tasks on O3-Mini. These large gaps in performance across categories highlight the importance of insights that DL-Bench can bring to help improve the LLM DL code

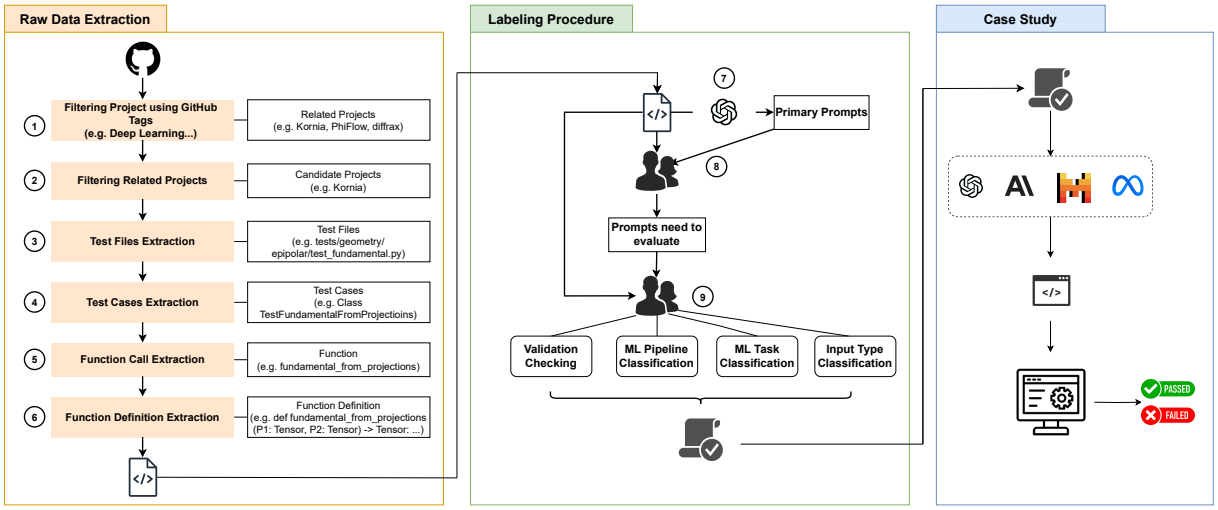


Figure 2.1: DL-Bench construction procedure

generation capability. Additionally, we construct a bug taxonomy of the issues found in the generated DL code. When compared to LLM-generated general code, LLM-generated DL code exhibits a higher frequency of *deviation from the prompt* issues and a new issues category *arithmetic and logical errors*.

Finally, we demonstrate a potential usage where DL-Bench can be used to guide few-shot prompting. In this usage, DL-Bench, on average, can consistently improve represented LLMs by **4.2%** on DS-1000. DL-Bench's data is available in our Kaggle repository[1]. The evaluation code is also available in our GitHub repository[2].

2.2 Approach

DL-Bench consists of 520 instances of AI and DL data points (filtered from over 2,000 raw data points). The data is curated from 30 GitHub repositories (selected from an initial pool of 160 related repositories).

[1]https://kaggle.com/datasets/b4b26b3d3ffe9930789d43da1377265a445add5023f87c9dc4bfcf4b50f93a62
[2]https://anonymous.4open.science/r/DL-Bench-71ED/

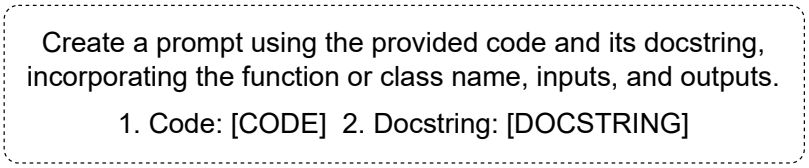


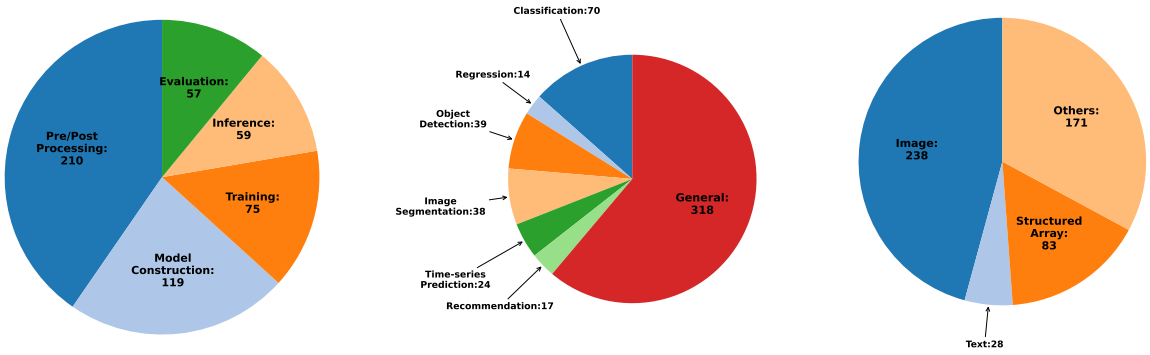
Figure 2.2: Template of generating prompt from code

The construction process of DL-Bench consists of two main phases: The Raw Data Extraction and the Labeling Procedure. The raw data extraction involves six semi-automatic steps. Since DL-Bench is designed to have diverse and realistic code samples, the first step (1) is to construct DL-Bench from code crawled from highly rated GitHub repositories (i.e., with the most stars), updated after the training cutoff of GPT-4o to mitigate data leakage, filtered using 30 DL-related terms such as "neural-networks", "pytorch", "computer-vision". We then manually select (step (2)) 160 high quality candidate DL projects (i.e., involve the integration of DL and AI-related frameworks, comprehensive test cases, clear and well-written docstrings, and detailed contribution guidelines). We then employed a bespoke utility to extract the test files and then test cases from each repository (step (3) and (4)). By performing static analysis, we were able to track and collect all of the functions under test in step (5) to form the raw data that is the base of DL-Bench.

Once the raw data is extracted, the labeling procedure starts. To speed up the task of constructing the prompt for each code sample, we utilize LLM (i.e., GPT-4o) as a code-explanation toolNam et al. [2024] to generate the first prompt candidate for each function under test (step (7)). Four co-authors were then tasked with manually filtering (step (8)) each entry to ensure that each function is highly relevant (i.e., contributes to a DL task such as image recognition, utilizes at least one recognized DL framework, and implements a relatively advanced and sophisticated algorithm). Finally, we conduct a manual labeling process involving four co-authors (step (9)) to refine the prompt and label each code sample

with the appropriate category from our three chosen types of categories: DL pipeline phases, ML task types, and input types.

2.3 Dataset Statistics



(a) Pipeline stages (b) Task types (c) Data types

Figure 2.3: Distribution of code samples in each category

DL-Bench consists of 520 instances of AI and DL data points, filtered from over 2,000 raw data points. These data are curated from 30 GitHub repositories, selected from an initial pool of 160 repositories. To ensure an accurate evaluation of code generation techniques, each prompt instance in DL-Bench is accompanied by at least three test cases, with an average of six test cases per instance.

Each data point is labeled according to its corresponding stage in the ML pipeline, the ML task it addresses, and the type of input data it processes. These labels enable fine-grained analysis of LLM performance across diverse ML-specific contexts.

Write a Python function **`draw_point2d`** to set `[x, y]` coordinates in an image tensor (**grayscale** or **multi-channel**) to a given color, returning the **modified image**.

Figure 2.4: An example prompt for Pre/Post processing

Create the `__init__` method for the **FCNN** class initializes a fully connected neural network with **input/output units**, **activation functions**, and **hidden layer sizes**. If not provided, default **hidden_units** to **(32, 32)**.

Figure 2.5: An example prompt for Model Construction

Figure 2.3 presents the distribution of DL-Bench data across ML stages, ML tasks, and input data types. In terms of ML pipeline stages (Figure a), the dataset covers all five stages, with Pre/Post-Processing being the most represented (210 samples). An example prompt from this stage is shown in Figure 2.4, which highlights the "draw_point2d" function used to annotate output images. Model Construction is the second most common stage (119 samples), such as the initialization function of a fully connected neural network shown in Figure 2.5.

Create a Python function **to_image** that accepts an input of type `Union[torch.Tensor, PIL.Image.Image, np.ndarray]` and returns a **tv_tensors.**`Image`. The function should check the input type and convert it accordingly

Figure 2.6: Example of General Task

Create a Python function **classification_metrics** that takes `ground_truth` and `retrieved` dictionaries and returns per class precision, recall, and F1 scores. Class 1 is assigned to duplicate file pairs while class 0 is for non-duplicate file pairs.

Figure 2.7: An example of Classification Task.

As illustrated in Figure b, over 61% of DL-Bench entries are labeled *General*, indicating their applicability across multiple ML tasks. For instance, the **to_image** function shown in Figure 2.6 is used for pre-processing image inputs without targeting a specific ML task. Among the task-specific entries, classification is the most common (70 samples), such as the metric calculation function shown in Figure 2.7. Regression tasks are the least represented, with only 14 examples.

Figure c displays the distribution across data types. Image data is the most prevalent, with 238 instances (46%). Cases where input types are ambiguous or not explicitly processed are grouped under *Others* (171 instances). An example is the model constructor function in Figure 2.5, where input data information is not available. Text data is the least represented due to frequent transformations into generalized formats like tensors.

2.3.1 Semantic Diversity Analysis

To analyze the domain specificity of benchmarks, we collected tokens from foundational DL/ML papers, such as "Attention Is All You Need" Vaswani et al. [2017], "YOLOv4" Bochkovskiy et al. [2020], and "BERT" Devlin et al. [2019]. We extracted commonly used terms in DL architectures and training, such as `cnn`, `transformer`, `dropout`, `adam`, and `cross_entropy`.

We computed a DL-relevance ratio for each benchmark instance, defined as:

$$r_i = \frac{|\{t \in T(d_i) \mid t \text{ contains a DL/ML keyword}\}|}{|T(d_i)|}$$

where d_i is the instance description and $T(d_i)$ is its token set. This ratio measures the concentration of DL/ML-specific terminology in each benchmark and helps assess how tailored each dataset is to DL tasks. For example, prompts featuring terms like `conv2d`, `batchnorm`, and `dropout` will have higher relevance scores than general-purpose algorithmic prompts. A complete list of extracted DL keywords and their usage statistics is available in our repository.

2.3.2 Data Categories and Labeling Schema

DL-Bench provides fine-grained categorization for each benchmark instance across three dimensions: the stage in the ML pipeline, the ML task type, and the input data type. These

dimensions were determined through iterative annotation and review, allowing researchers to analyze LLM performance across domain-relevant axes.

Stage in the ML Pipeline. We annotate each function according to its role in the ML workflow:

- **Pre/Post-Processing:** Functions that manipulate or transform data before model inference or after output generation.
- **Model Construction:** Code that defines architectures (e.g., layers, activation functions, loss functions) for DL models.
- **Training:** Functions involving model optimization, loss computation, and backpropagation.
- **Inference:** Logic that applies a trained model to new data without updating weights.
- **Evaluation & Metrics:** Code that assesses model performance using metrics such as accuracy, recall, or precision.

ML Task Type. Instances are labeled according to the type of ML task they support:

- **Classification:** Code performing discrete class predictions.
- **Regression:** Functions generating continuous numeric outputs.
- **Object Detection:** Code that outputs bounding boxes or uses NMS for identifying objects.
- **Image Segmentation:** Pixel-wise labeling or mask generation for images.
- **Time Series Prediction:** Forecasting temporal sequences using past observations.

- **Recommendation:** Code that computes rankings or scores for recommending items.
- **General:** Functions that serve across multiple tasks or cannot be pinned to one specific ML task.

Input Data Type. We identify the primary input data format of the function:

- **Image:** Functions that receive or manipulate image tensors or apply vision-specific operations.
- **Text:** Functions that involve tokenization, embeddings, or other natural language processing steps.
- **Structured Array:** Code processing structured formats like CSVs, NumPy arrays, or tabular tensors.
- **Others:** When input data type is implicit or unavailable (e.g., network initialization), we assign the "Others" label.

This tri-label schema allows researchers to analyze not only overall model accuracy but also how performance varies depending on the role, task, and data handled by the generated code. This flexibility makes DL-Bench uniquely useful for debugging, model comparison, and fine-grained performance breakdowns in ML-centric code generation.

2.4 Experimental Setup

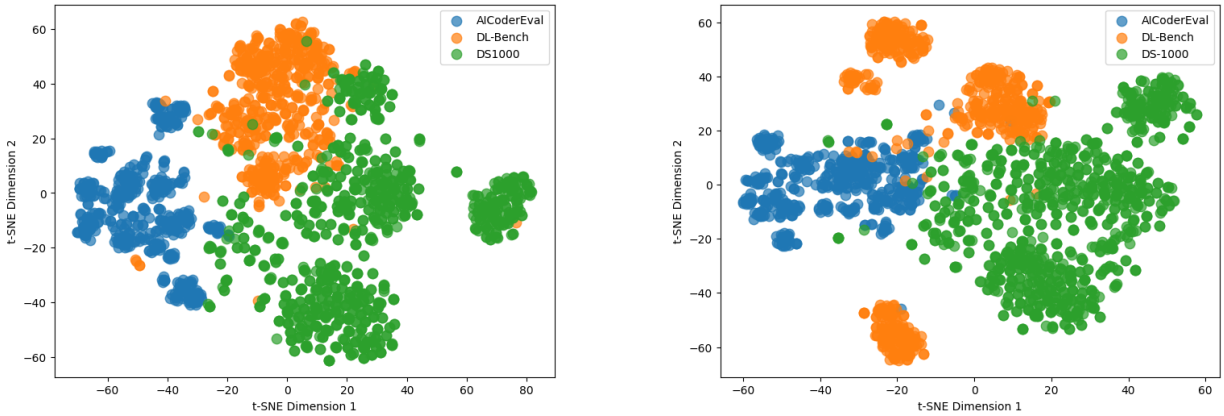
To differentiate DL-Bench from prior benchmarks and demonstrate its potential, we perform a study evaluating the performance of state-of-the-art LLM on the DL-Bench benchmark. We first quantitatively compare DL-Bench with its most related (similar scope and topic) benchmark DS-1000. We first show that DL-Bench contains problems that are not overlapped with

DS-1000 by comparing the embedding of the prompts. We show that DL-Bench problems are more related to DL and ML and require more complex code to solve, hence would be more challenging for LLMs. We then confirm this by performing the experiments on seven representative LLMs. These LLMs cover the current spectrum of parameter scales, licensing regimes, and training specialisations: ***Llama 3.2 70B*** (Best Openly released checkpoint), ***GPT-4o*** and ***Claude-3.5 Sonnet*** (high-performing commercial general-purpose systems), ***O3-mini*** and ***DeepSeek-V3*** (compact open-weight models with competitive reasoning ability), ***Qwen-Coder*** (a code-specialised model), and ***Mixtral 8*22B*** (a lightweight baseline suited to modest compute budgets).

The LLMs are evaluated based on the commonly used pass@k, which measures the likelihood that at least one of the k-generated solutions passes all test cases Lyu et al. [2024]. To minimize non-determinism and improve reproducibility, we set the temperature to zero for LLMs Bommasani et al. [2021]. This allowed us to accurately measure the quality of the generated code across various DL tasks, offering deeper insights into the LLM's strengths and weaknesses.

2.5 Results

To differentiate DL-Bench from prior benchmarks and demonstrate its potential, we perform a quantitative comparison between DL-Bench and its related benchmarks (DS-1000 and AICoderEval). We first analyze the data in each benchmark to show that DL-Bench contains novel and challenging DL-specific problems that require more complex solutions. Then we, empirically, show that DL-Bench is more challenging to solve than DS-1000 by comparing the performance of seven representative LLMs.



(a) Prompt (b) Code solution

Figure 2.8: Prompt and code solution embeddings projection for DL-Bench, DS-1000, and AICoderEval

2.5.1 DL-Bench contains distinctive and more challenging problems and solutions when compared to DS-1000 and AICoderEval.

In this section, we evaluate how DL-Bench diverges from its closest data-science and ML benchmarks, DS-1000 and AICoderEval. First, we contrast the semantic spaces of their input prompts and code solutions by comparing embedding distributions. Second, we gauge each benchmark's DL orientation by tracking the prevalence of DL-specific tokens in the reference code Finally, we measure code complexity to provide a holistic view of DL-Bench's problems relative difficulty.

Semantic Comparison: To compare the semantic prompt space, we embed each natural-language prompt with the `all-MiniLM-L6-v2` sentence-transformer [3]. The average cosine similarity values between DL-Bench and related benchmarks are relatively low (**0.188** for DS-1000 and **0.184** for AICoderEval). Such notable semantic divergence in DL-Bench's input prompts from related benchmarks indicates that DL-Bench covers distinct domains or task formulations. Additionally, when projecting the embeddings into two dimensions using

[3]384-dimensional encoder released by `https://www.sbert.net`

t-SNE Van der Maaten and Hinton [2008] (Figure 2.8a), the visualization reveals separable clusters corresponding to DL-Bench and DS-1000. This distinct clustering further supports the semantic uniqueness of DL-Bench and highlights its complementary role in benchmark diversity.

To demonstrate the distinctiveness of DL-Bench's solutions, we compare the semantic representation of its ground-truth code with that of DS-1000 and AICoderEval. To this end, we use `CodeBERT` Feng et al. [2020] to generate code embeddings for each reference implementation, and apply cosine similarity computation. DL-Bench problems require code solutions that are semantically different from DS-1000 and AICoderEval, with the average cosine similarity of **0.538** and **0.638** respectively. This indicates each benchmark covers a distinct set of tasks and requires different solution patterns. Figure 2.8b visualizes these differences by applying t-SNE to the ground-truth code embedding space. The projection reveals separated clusters for DL-Bench, DS-1000, and AICoderEval, providing further evidence that DL-Bench offers complementary coverage and contributes novel content to existing DL/ML code generation benchmarks.

DL-relevance Analysis: To measure DL-Bench's DL/ML relevancy, we compute a domain-relevance metric based on the DL/ML tokens ratio in the reference code. Averaged across instances, DL-Bench attains a ratio of **0.785**, far exceeding DS-1000 (0.131) and AICoderEval (0.437). Put differently, more than three-quarters of the lexical footprint in DL-Bench code is devoted to DL/ML concepts, whereas only one-eighth in DS-1000 and less than half in AICoderEval references such terms.

Solution Complexity: To gauge the relative complexity of DL-Bench's problems, we compare three structural metrics: lines of code (LOC), cyclomatic complexity, and cognitive complexity (extracted with radon[4]). On average, solutions in DL-Bench span **14.8 LOC**,

[4] `https://pypi.org/project/radon/`

Table 2.2: Pass@1 (%) scores for various SOTA LLMs on DS-1000 and DL-Bench, sorted by DL-Bench performance

Benchmark	O3-Mini	DeepSeek-V3	GPT-4o	Claude 3.5 Sonnet	Llama 3.1 70B	Mixtral 8*22B	QwenCoder	Avg.
DL-Bench	**36.9**	31.4	31.2	30.3	27.5	26.1	23.4	29.5
DS-1000	61.0	61.7	51.1	**61.9**	40.9	39.3	57.3	53.3

Table 2.3: Variance across five runs on DL-Bench (temperature = 0). Lower variance indicates more stable performance.

Run / Stat	O3-Mini	DeepSeek V3	GPT-4o	Claude 3.5 Sonnet	LLaMA 3.1 70B	Mistral 8×22B	Qwen Coder 2.5
Run 1	36.9	31.4	31.2	30.3	27.5	26.1	23.4
Run 2	35.6	30.6	29.4	29.8	27.1	23.9	21.7
Run 3	34.6	28.5	28.3	29.9	25.8	24.0	22.7
Run 4	34.6	32.1	30.8	31.2	28.3	22.5	23.8
Run 5	33.7	30.2	31.6	31.4	25.2	23.1	22.5
AVG	35.1	30.5	30.2	30.5	26.7	23.9	22.8
VAR	1.49	1.86	1.89	0.55	1.60	1.86	0.67
STD	1.22	1.36	1.37	0.74	1.26	1.36	0.82

nearly double AIcoderEval (8.5) and more than quadruple DS-1000 (3.6). Cognitive complexity follows a similar pattern (**4.26** vs. 0.31 and 0.008), underscoring more complex nested structures and longer call chains in DL-Bench's solutions.

Finding 1: DL-Bench's problems focus on DL-specific domain and occupy a distinct semantic space. Furthermore, DL-Bench contains difficult problems that require significantly more complex code solutions that pose significant challenges to advanced LLMs.

2.5.2 Performances of SOTA LLMs on DL-Bench and DS-1000

This analysis investigates how the existing ML code generation benchmark (DS-1000) and DL-Bench evaluate seven representative LLMs covering a spectrum of parameter scales, licensing regimes, and training specializations. Since AICoderEval has not been published and does not provide sufficient and reliable evaluation scripts, we have decided to exclude it from this evaluation. A commonly used pass@k Lyu et al. [2024], which measures the likelihood that at least one of the k-generated solutions passes all test cases, is used in this evaluation.

Table 2.4: Pass@k (%) on DL-Bench at temperature = 0.3.

Model	**Pass@1**	**Pass@3**	**Pass@5**
O3-mini	36.9	38.1	40.2
DeepSeek V3	31.4	34.6	36.7
GPT-4o	31.2	33.9	37.9
Claude 3.5 Sonnet	30.3	32.5	35.6
LLaMA 3.1 70B	27.5	29.4	33.9
Mistral 8×22B	26.1	27.3	31.4
Qwen Coder 2.5	23.4	26.1	29.0

Table 2.5: Live DL-Bench results by data recency window (cutoff: Oct 2023).

Model	**Overall**	**After Oct 2023**	**After Jan 2024**	**After May 2024**	**After Sep 2024**
Claude 3.5 Sonnet	30.3	30.3	29.7	28.2	27.6
DeepSeek V3	31.4	31.4	29.5	27.2	27.1
GPT-4o	31.3	31.3	29.4	26.2	25.4
LLaMA 3.1 70B	27.5	27.5	27.4	26.1	24.6
Mistral 8×22B	24.6	24.6	23.5	22.4	22.9
O3-mini	36.3	36.3	32.8	29.6	30.0
Qwen Coder 2.5	23.4	23.4	22.5	23.4	24.1

To minimize non-determinism and improve reproducibility, we set the temperature to zero for all LLMs Bommasani et al. [2021]. We intentionally avoided using specialized prompt strategies, opting instead for vanilla prompts to focus on the model's baseline performance. However, the use of advanced prompt engineering strategies could yield different results. In a later section, we demonstrate a potential usage of DL-Bench as a guided few-shot dataset. Table 2.2 shows the pass@1 values of SOTA LLMs on these DL-Bench and DS-1000.

Our evaluation shows that even the most advanced model, such as O3-Mini, struggles with ML/DL-specific code generation. Specifically, O3-Mini achieves 61.0% pass@1 in DS-1000 but only 36.9% pass@1 on DL-Bench. Similarly, all other tested LLMs get much lower pass@1 scores in DL-Bench than DS-1000. The overall weak performance of these models highlights the ongoing challenges in generating reliable, executable ML/DL-specific code, supporting the need for deeper analysis to identify problematic areas that DL-Bench can provide.

Table 2.6: Static code quality comparison on passing cases: Ground-truth (GT) vs LLM-generated (Gen). CC: Cyclomatic Complexity; CBO: Coupling Between Objects.

Model	**GT_CC**	**Gen_CC**	**GT_CBO**	**Gen_CBO**
O3-mini	2.6	5.0	1.8	3.8
DeepSeek V3	2.7	4.0	1.8	3.2
GPT-4o	2.4	3.8	1.5	3.3
Claude 3.5 Sonnet	2.3	4.4	1.7	3.8
LLaMA 3.1 70B	2.2	3.9	1.6	3.3
Mistral 8×22B	2.5	3.7	1.6	3.3
Qwen Coder 2.5	2.4	3.8	1.5	3.3

Beyond pass@1, we also computed pass@3 and pass@5 at temperature 0.3 to examine potential gains from sampling multiple generations. The results (Table 2.4) show modest improvements, with O3-Mini increasing from 36.9% to 40.2% at pass@5, and GPT-4o from 31.2% to 37.9%. However, even with more attempts, no model reaches a level suitable for high-stakes production ML/DL coding, reinforcing the benchmark's difficulty.

We further analyzed result stability by running each model five times on DL-Bench. As shown in Table 2.3Variance across runs is consistently low (0.55–1.89), indicating that the benchmark produces stable measurements despite inherent stochasticity in LLM outputs. This property is essential for fair model comparison and makes DL-Bench suitable for longitudinal evaluations.

Another dimension explored is the live benchmark evaluation, where we measured performance on progressively newer subsets of the dataset (after Oct 2023, Jan 2024, May 2024, and Sep 2024). As shown in table 2.5, All models exhibit performance decay as the code recency window narrows, with O3-Mini dropping from 36.3% overall to 30.0% on the most recent subset. This suggests that recency plays a role in model capability, possibly due to training data cutoffs or lagging adaptation to recent ML/DL library changes.

We also examined static code quality for passing cases, comparing Cyclomatic Complexity (CC) and Coupling Between Objects (CBO) between ground-truth (GT) and generated (Gen) solutions. As shown in Table 2.6, Across all models, generated code exhibits higher

complexity and coupling (e.g., O3-Mini: GT_CC = 2.6 vs Gen_CC = 5.0; GT_CBO = 1.8 vs Gen_CBO = 3.8), suggesting lower maintainability and potentially higher defect proneness. This insight indicates that even when LLMs produce correct outputs, they tend to do so with more complex and less modular code.

Our separability and ranking agreement analysis between DL-Bench and DS-1000 yielded a huge (more than 2.0) Cohen's-d effect size of **3.13** and a large (more than 1.0) Fisher's ratio of **4.9**, confirming that DL-Bench is markedly more challenging with significantly lower and distinct distribution of pass@1 scores. However, the average Spearman correlation of **0.50** ($p = 0.25$) shows that the ranking of models is moderately consistent with DS-1000. This suggests that DL-Bench presents harder problems and contains additional aspects that can capture slightly different relative performance among LLMs.

Finding 2: Our evaluation indicates that current SOTA LLMs struggle to generate correct, executable code for ML/DL tasks with an average pass@1 score of **29.5%** on DL-Bench. Although O3-Mini is the strongest among the tested models, it still falls short of meeting practical standards with a pass@1 score of only **36.9%**. Even when sampling multiple generations (pass@5), no model surpasses 40.2%. Performance variance is low, indicating stable and reliable measurement, but recent code additions further reduce success rates. Moreover, static code quality analysis reveals that generated code, even when correct, is more complex and coupled than ground truth, raising maintainability concerns. Empirically, DL-Bench presents more challenging problems and captures aspects that result in a slightly different ranking among LLMs compared to DS-1000.

2.6 Qualitative Analysis

This section provides a deeper analysis of which kinds of DL-specific code are harder to generate, and the common issues that generated DL-specific code has.

Table 2.7: Pass@1 (%) scores on DL-Bench across stages, ML/DL tasks, and input data types, sorted by DL-Bench performance

Category	O3-Mini	DeepSeek-V3	GPT-4o	Claude 3.5 Sonnet	Llama 3.1 70B	Mixtral 8*22B	QwenCoder	Avg.
Stages in pipeline								
Pre/Post Processing	**41.1**	35.4	35.8	33.9	31.0	28.2	25.4	**32.9**
Model Construction	**31.6**	27.8	25.2	25.2	20.8	17.6	15.1	23.3
Training	**32.7**	29.9	32.0	27.5	29.3	28.0	28.9	29.7
Inference	**40.0**	27.7	30.5	28.5	27.5	28.5	25.0	29.6
Evaluation & Metrics	**37.1**	30.0	26.8	32.7	26.2	25.4	25.4	29.0
ML tasks								
Classification	**37.6**	27.0	29.0	30.0	25.0	30.4	24.6	29.1
Regression	**41.8**	22.4	28.0	28.5	13.2	22.4	14.2	24.3
Object Detection	**31.4**	22.6	29.2	21.6	12.2	10.8	10.8	19.8
Image Segmentation	27.9	**28.7**	15.2	18.4	20.6	15.7	23.0	21.3
Time Series Prediction	**40.6**	20.8	37.0	29.1	20.8	20.8	20.8	27.1
Recommendation	54.9	36.2	47.0	**58.8**	35.2	47.5	41.1	**45.8**
General	**37.3**	34.6	33.0	31.5	32.6	28.3	24.3	31.6
Input data types								
Image	**35.2**	31.7	29.1	27.5	23.3	20.3	18.4	26.5
Text	**53.5**	29.2	40.9	45.4	35.3	45.4	29.2	**39.8**
Structured Array	**38.7**	29.9	28.9	30.1	26.5	30.1	22.8	29.5
Others	**36.2**	32.3	35.0	32.3	33.3	29.9	30.3	32.7

2.6.1 Which kinds of DL-specific code pose a greater challenge for SOTA LLMs?

We analyze the performance differences among categorizations that DL-Bench provides. Table 2.7 presents the pass@1 scores that each LLM achieves for generated code in each categorization that DL-Bench provides: stages in DL/ML pipeline, ML tasks, and input data types. Among all LLMs, the most advanced LLM, O3-Mini, consistently outperforms others in all categorizations. However, in some ML tasks, such as object detection and recommendation, DeepSeek-V3 and Claude 3.5 perform better.

Stages in pipeline: Among stages in the DL/ML pipeline, *pre/post processing* generated code has the highest average pass@1 score of **31.9%**. Code in these stages varies significantly because it prepares and cleans the input and formats output data for various models. This makes samples of this type the most available in training data and could explain the higher pass@1 scores. On the other hand, LLMs struggle to generate code for the *model construction*

stage, with the lowest average pass@1 score of **23.3%**. This is because the code for this stage is more complex, very project-specific, and often longer.

Finding 3: LLMs perform best (average pass@1 score of 32.9%) in pre/post processing stages and worst (average pass@1 score of 23.3%) in model construction. These differences could be due to the high availability of training data for pre/post processing stages, and the more complex and project-specific nature of code in model construction,

ML tasks: Table 2.7 presents a significant disparity in the pass@1 score of generated code across ML tasks. Notably, scores for the *recommendation* task are the highest (45.8% average), with the best score of 58.8% for Claude 3.5 Sonnet. On the other end of the scale, *object detection* and *image segmentation* tasks' scores are the lowest (averaged 19.8% and 21.3% respectively). These results indicate that each ML task type has its characteristics that LLMs can or cannot yet capture. Specifically, image processing code for object detection and image segmentation remains challenging for LLMs.

Finding 4: Different ML/DL tasks vary in complexity, affecting LLMs' code generation abilities with varying pass@1 scores averaged from 45.8% to 19.8%. Each LLM can have its strengths and weaknesses when generating code for different ML tasks.

Input data types: Across different types of input, the result in Table 2.7 indicates a more consistent pass@1 of all LLMs, except for textual data, where LLMs exhibit better performance (averaged 39.8%). We assume that most textual input data types are tokenized and converted before being processed in the DL model, which makes functions that deal directly with textual input data types quite standard and easier to generate. On the other hand, performance for image-related tasks perform the worst with averaged score of 26.5%. This can be attributed to the inherent complexity and lack of consistent structure in image

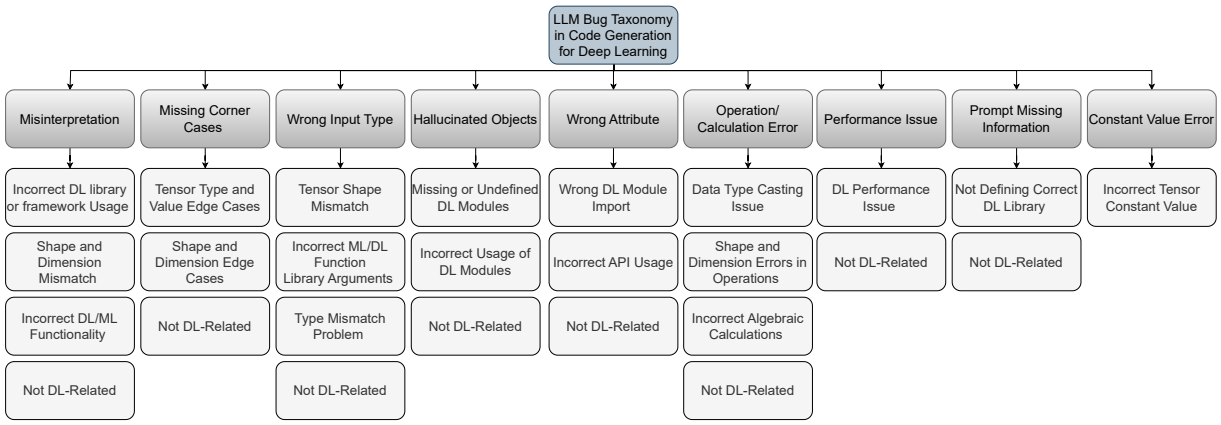


Figure 2.9: Taxonomy of bugs in DL generated code. Only categories with DL-related subcategories are shown.

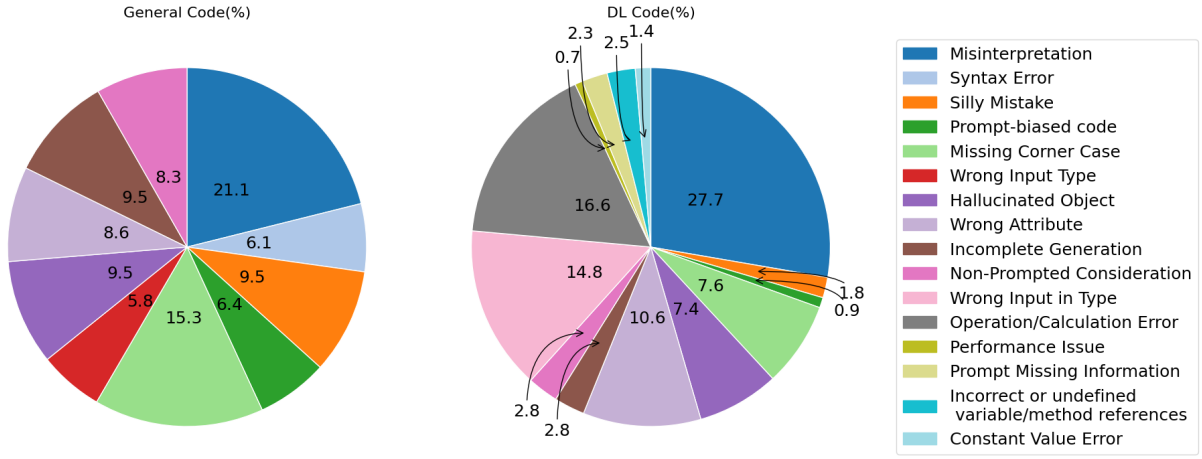


Figure 2.10: Distribution of bugs in general code vs. DL code generated by LLM

data, such as varying shapes, resolutions, and channel configurations (e.g., grayscale vs. RGB).

Finding 5: Among input data types, image data with more complex structures is the hardest to generate code for, with the lowest average pass@1 score of 26.5%. In contrast, textual data tasks achieved higher performance (average 39.8%), likely due to more deterministic coding in the pre-processing stages.

2.6.2 What are the common bugs in generated DL-specific code?

To investigate this question, we build a taxonomy of common bug patterns and issues that arise in DL code generated by GPT-4o (the best model at the time of analysis). This taxonomy is an expansion of Tambon et.al Tambon et al. [2024]'s bug taxonomy for LLM-generated regular code. Following the same procedure as our labeling process, three authors manually investigate all GPT-4o failures and categorize them following Tambon et.al's taxonomy. At the same time, the annotators identify the DL-specific sub-categories for each failure. The result is the taxonomy presented in Fig 2.9. The appendix gives a detailed explanation of each bug type and sub-category.

Differences in failures of the DL and general generated code: Tambon et. alTambon et al. [2024] analyzed failures when CodeGen models generate code for the general tasks. Figures 2.10 show the distributions of the bug types when generating general code vs DL code. On the one hand, *misinterpretation* (blue) is a common bug when generating both general code and DL code; however, due to more complex logic and arithmetic requirements, LLMs more often make this mistake when generating DL code. On the other hand, since GPT4o is much more capable compared to CodeGen models used by prior work, errors such as *incomplete generation* (brown), *silly mistake* (orange), and *syntax error* (light blue) occur at a much lower rate.

Additionally, we introduce several new categories of bugs that only arise in DL code generation. Firstly, *errors in arithmetic and logical operations*(dark gray) occur when incorrect calculations or flawed logical code are generated. Secondly, *performance*(yellow) issues involve inefficiently generated code with slow execution times, excessive memory consumption, or suboptimal utilization of resources. Lastly, *prompt missing information*(light yellow) occurs when the prompts are missing details to fully address the problem at hand, resulting

in incomplete or partially implemented solutions. These new categories identify important challenges that are unique to DL code generation.

Finding 6: *Misinterpretation* is a common issue in both generated general code and DL code; however, due to more complex logic and arithmetic requirements, LLMs are more likely to make this mistake when generating DL code. *Errors in arithmetic and logical operations*, *performance*, and *prompt missing information* emerged as new issues that are specific to DL code generation.

Bugs in human-written compared to LLM-generated DL code: Prior study Islam et al. [2019] has identified the most common types of bugs in human-written DL code (including logic errors, API misuse, and data-related issues), with API misuse and data flow bugs being the most prevalent issues in TensorFlow and Pytorch, respectively. Although API misuse remains a frequent issue in DL generated code, data structural problems, such as tensor mismatches and dimensional errors, occur more frequently. Human-written and LLM-generated DL code both often contain logic errors. This similarity may stem from the fact that LLMs are trained on human-written code, thereby inheriting logical structures and concepts from human programmers.

Finding 7: Due to LLMs' weaknesses, LLM-generated DL code contains more data structural problems, such as tensor and dimension mismatches. However, due to reliance on human-generated training data, LLM-generated DL code shared bug patterns such as logic and API misuse errors.

2.7 LLM Bug Types and DL-specific Subtypes

In this section, we provide details for the common types of errors in LLM-generated code as well as our DL-specific subtypes.

Misinterpretation: ***Generated code deviates from the prompt intention*** The produced solution does not fulfill the user's original requirements or strays from the specified goals. This often indicates that the LLM has misunderstood or incompletely parsed the prompt.

- **Incorrect DL Library or Framework Usage:** The generated code does not match the requested library or framework. For example, if the prompt asks for a TensorFlow implementation of a CNN, but the LLM generates the model using PyTorch instead, or if a user requests a NumPy-based neural network operation but the output code uses TensorFlow functions.
- **Shape and Dimension Mismatch:** The LLM produces code with incorrect tensor dimensions that do not follow the prompt specifications. For example, if the prompt requests a fully connected layer expecting an input of shape $(64, 128)$, but the generated code initializes it with an input shape of $(128, 64)$, leading to a mismatch in matrix operations.
- **Incorrect DL/ML Functionality:** The generated code does not implement the correct functionality as described in the prompt. For instance, if the prompt asks for a binary classification model using a sigmoid activation function, but the output code instead applies a softmax activation function intended for multi-class classification, altering the intended behavior.

Syntax Error: ***Missing parenthesis, semicolon, or other syntax issues*** Straightforward syntactic mistakes such as unclosed quotes, unmatched braces, or misplaced punctuation prevent the code from compiling or running properly.

Silly Mistake: ***Redundant conditions, unnecessary casting*** Simple but avoidable errors, such as repeating the same condition twice or performing extra type conversions

with no purpose. While these do not always break the code, they reduce readability and hint at confusion in the model's reasoning.

Prompt-biased Code: ***Code overly relies on examples from the prompt*** The LLM anchors too strongly to the examples provided in the prompt, resulting in a solution that works only for the specific inputs shown rather than generalizing the logic for broader applicability.

Missing Corner Cases: ***Edge cases not handled*** The generated solution neglects special scenarios such as empty inputs, boundary values, or invalid parameters, leading to unreliable behavior outside of typical inputs.

Tensor Type and Value Edge Cases: These bugs occur when operations fail due to unexpected tensor types or values. For example, using a tensor with `float32` data type in a function that expects integers or encountering issues when dividing by zero in a tensor.

Shape and Dimension Edge Cases: Bugs of this type happen when operations fail because of unexpected edge-case shapes. For example, trying to perform a convolution on a tensor with a batch size of 0 or a single dimension, such as $(1, 28, 28)$, when a shape like $(32, 28, 28)$ is expected.

Wrong Input Type: ***Incorrect input type in function calls*** The code passes incompatible data types to functions or methods (e.g., providing a string instead of a list), which causes runtime failures or nonsensical outputs.

Tensor Shape Mismatch: The generated code provides tensors with incorrect shapes to functions, leading to shape-related errors. For example, passing a 3D tensor of shape $(batch, height, width)$ to a function that expects a 4D tensor of shape

($batch, channels, height, width$), causing a runtime error in deep learning frameworks like PyTorch or TensorFlow.

Incorrect ML/DL Function Library Arguments: These occur when invalid arguments are passed to functions. For instance, using `stride=-1` in a convolution function, which is not logically or mathematically valid.

Type Mismatch Problem: The generated code uses tensors with incompatible data types in operations. For example, passing a tensor with data type `float32` to a function that expects `int64`, or attempting to index a tensor with a floating-point value instead of an integer, leading to type-related execution failures.

Hallucinated Object: ***Nonexistent or undefined objects used*** The LLM invents objects, classes, or modules that do not exist or have not been imported or defined. These errors result in runtime failures or developer confusion.

Missing or Undefined DL Modules: This happens when a model, layer, or module that hasn't been properly defined or initialized is used. For example, attempting to forward-pass input through a neural network layer that hasn't been added to the model.

Incorrect Usage of DL Modules: The generated code references deep learning modules, functions, or classes that do not exist or belong to the wrong framework. For example, calling `torch.nn.Dense()` instead of `torch.nn.Linear()`, or attempting to use `tensorflow.layers.Conv2D` instead of `tf.keras.layers.Conv2D`. These hallucinated module names cause import errors or incorrect function calls.

Wrong Attribute: ***Incorrect/nonexistent attributes for objects or modules*** The LLM references valid objects but assigns them invalid or incorrect attributes. These subtle

errors often result from misunderstandings of library APIs or typos in the generated code.

Wrong DL Module Import: Bugs of this nature arise when modules are imported incorrectly. For example, importing `jax` functions when the rest of the code is written in PyTorch, leading to incompatibilities during execution.

Incorrect API Usage: These bugs occur when a library API function is called incorrectly. For example, using the `train()` method instead of `fit()` for a Keras model or passing parameters in the wrong order to an optimizer.

Non-Prompted Consideration: ***Non-requested features added*** The LLM includes functionality unrelated to the requirements, often due to extraneous training data or contextual noise. This bloats the code and complicates its scope.

Operation/Calculation Error: ***Errors in arithmetic or logical operations*** The LLM makes errors in mathematical calculations or logical expressions, such as confusing addition with subtraction or mixing up operator precedence. These subtle mistakes produce incorrect results.

Data Type Casting Issues: These bugs occur when tensors or variables are cast into incompatible data types. For instance, casting a `float32` tensor into `int32` without considering the loss of precision, which may disrupt training.

Shape and Dimension Error in Operations: The generated code performs mathematical operations on tensors with incompatible shapes or dimensions, leading to incorrect computations or runtime failures. For example, attempting to add two tensors of shapes $(32, 64)$ and $(64, 32)$ without proper broadcasting, or performing a matrix multiplication between tensors with mismatched inner dimensions, such as $(4, 3) \times (5, 4)$, causing a shape misalignment error.

Incorrect Algebraic Calculation: These bugs refer to mathematical errors in computations. For instance, incorrectly normalizing data by dividing by the mean instead of the standard deviation, leading to improper scaling of input features.

Performance Issue: This category includes inefficiencies in the generated code that impact runtime or resource usage. Examples include unnecessary nested loops, unoptimized algorithms, or excessive use of memory. While the code may produce correct results, its suboptimal implementation can make it impractical for large datasets or real-time applications. Performance issues often arise because the LLM generates a brute-force solution without understanding optimization principles.

DL Performance Issues: These bugs refer to inefficiencies in implementation that degrade model performance. For instance, not using GPU acceleration for operations or improper batching strategies leads to high memory consumption and slow training.

Prompt Missing Information: ***Incomplete or unclear prompts*** The bug arises due to insufficient detail or ambiguity in the input prompt, leading the LLM to make assumptions or guess certain details when generating the code. For example, if the prompt does not specify edge case handling or input constraints, the model may overlook these aspects entirely. This highlights the importance of crafting precise and comprehensive prompts when using LLMs for code generation.

Not Defining the Correct DL Library in the Prompt: This occurs when the prompt or instructions fail to specify the appropriate library or framework. For example, a user asks a language model to generate PyTorch code but does not explicitly state this, leading to TensorFlow code generation instead.

Incorrect or Undefined Variable/Method References : Variables or methods that are not defined or incorrectly referenced The LLM generates code that includes references to variables or methods that do not exist or are improperly used, leading to runtime errors such as NameError or AttributeError.

Constant Value Error: *Incorrect constant value assignment* The LLM assigns incorrect or miscalculated constant values, such as setting a time-out period to `10ms` instead of `1000ms`, leading to unexpected behavior.

Incorrect Tensor Constant Value: This type of bug arises when tensors are initialized with incorrect values, leading to flawed model behavior. For example, initializing weights or biases with all zeros instead of random values causes issues in training dynamics.

2.8 Distribution of Failures in Generated DL Code

Table 2.8 presents the distribution of bugs in LLM-generated DL code. The most prevalent issue is **deviation from the prompt**, accounting for the largest portion of errors. Unlike general LLM-generated code, DL code is more prone to **arithmetic and logical errors**, reflecting the complexity of numerical computations. Additionally, **incorrect input types in function calls** represent a significant share of the identified bugs, highlighting a common source of failures in generated DL code.

Table 2.8: Distribution of bugs in LLM generated code for deep learning

Category	DL Related Categories	# of Occurances	
Misinterpretation: Generated code deviates from prompt intention	Incorrect DL library or framework Usage	10	120
	Shape and dimension mismatch	45	
	Incorrect DL/ML Functionality	13	
	Not DL-related	52	
Syntax Error: Missing parenthesis, semicolon, or other syntax issues			0
Silly Mistake: Redundant conditions, unnecessary casting	Not DL-related	8	8
Prompt biased Code: Code overly relies on examples from the prompt	Not DL-related	4	4
Missing Corner Case: Edge cases not handled	Tensor Type and Value Edge Cases	8	33
	Shape and Dimension Edge Cases	15	
	Not DL-related	10	
Wrong input type:Incorrect input type in function calls	Tensor shape mismatch	3	64
	Incorrect ML/DL function library arguments	16	
	Type mismatch problem	23	
	Not DL-related	22	
Hallucinated Objects: Nonexistent or undefined objects used	Missing or Undefined DL Modules	9	32
	Incorrect Usage of DL Modules	12	
	Not DL-related	11	
Wrong Attribute: Incorrect/nonexistent attributes for objects or modules	Wrong DL Module import	8	46
	Incorrect API Usage	17	
	Not DL-related	21	
Non-Prompted Consideration:Non-requested features added	Not DL-related	12	12
Operation/Calculation Error:Errors in arithmetic or logical operations	Data Type Casting Issues	5	72
	Shape and Dimension Errors in Operations	28	
	Incorrect Algebraic Calculations	18	
	Not DL-related	21	
Performance Issue: Poor Performance	DL performance issue	2	3
	Not DL-related	1	
Prompt missing information:Incomplete or unclear prompts	Not defining correct dl library	4	10
	Not DL-related	6	
Incorrect or undefined variable/method references: Variables or methods that are not defined or incorrectly referenced	Not DL-related	11	11
Constant Value Error:Incorrect constant value assignment	Incorrect Tensor Constant Value	6	6

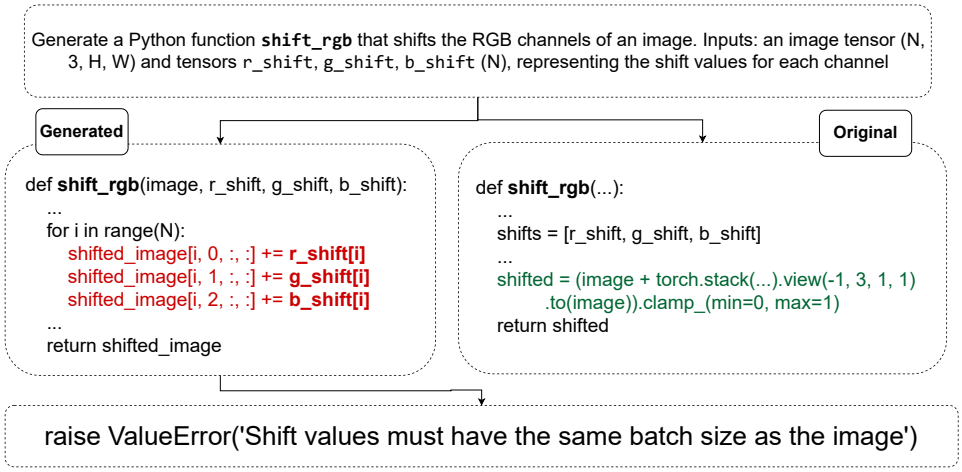


Figure 2.11: Mismatching data shapes: shifting variables need to be broadcasted to the image shape

2.8.1 Some examples of incorrect LLM-generated DL code:

Example 1: Figure 2.11 highlights an instance of dimensional mismatches in LLM-generated DL code. In this case, GPT-4o incorrectly assumes that each shift value can be applied directly to all pixels in the image channel, causing a shape mismatch.

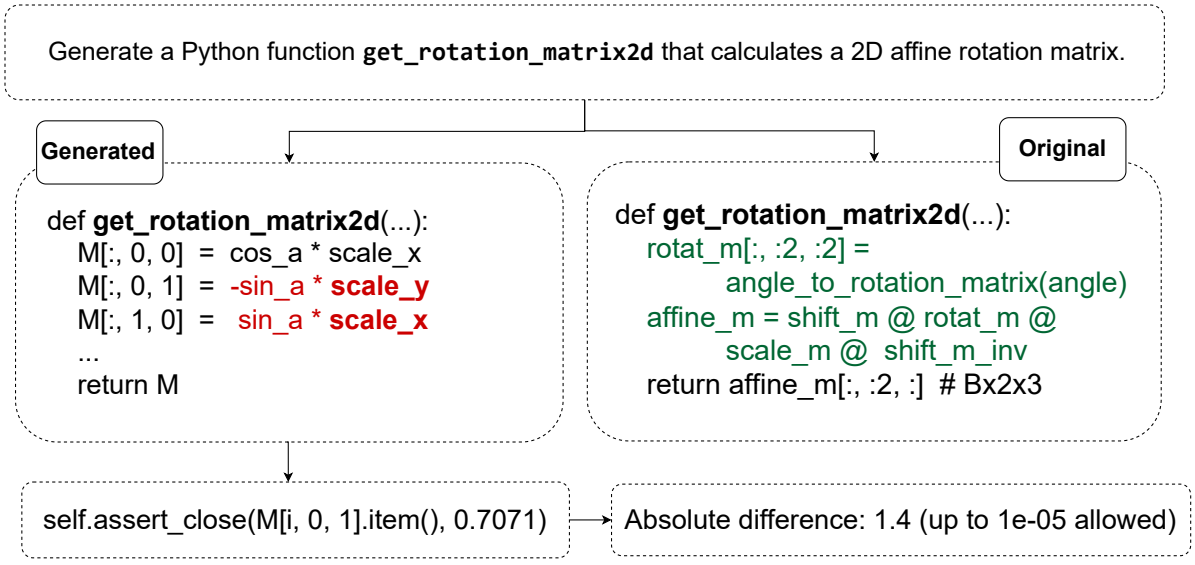


Figure 2.12: Incorrect processing of parameters: The axes scales need to be applied to both sin and cos

Example 2: An example of such logic-related bugs is shown in Figure 2.12, demonstrating how LLMs replicate logical reasoning errors that occur in human-written code. Here, GPT-4o applies *scale_x* only to the cosine, whereas the scaling factors *scale_x* and *scale_y* should be applied uniformly to both the sine and cosine components of the rotation matrix. This results in improper scaling along the axes and triggers a test failure.

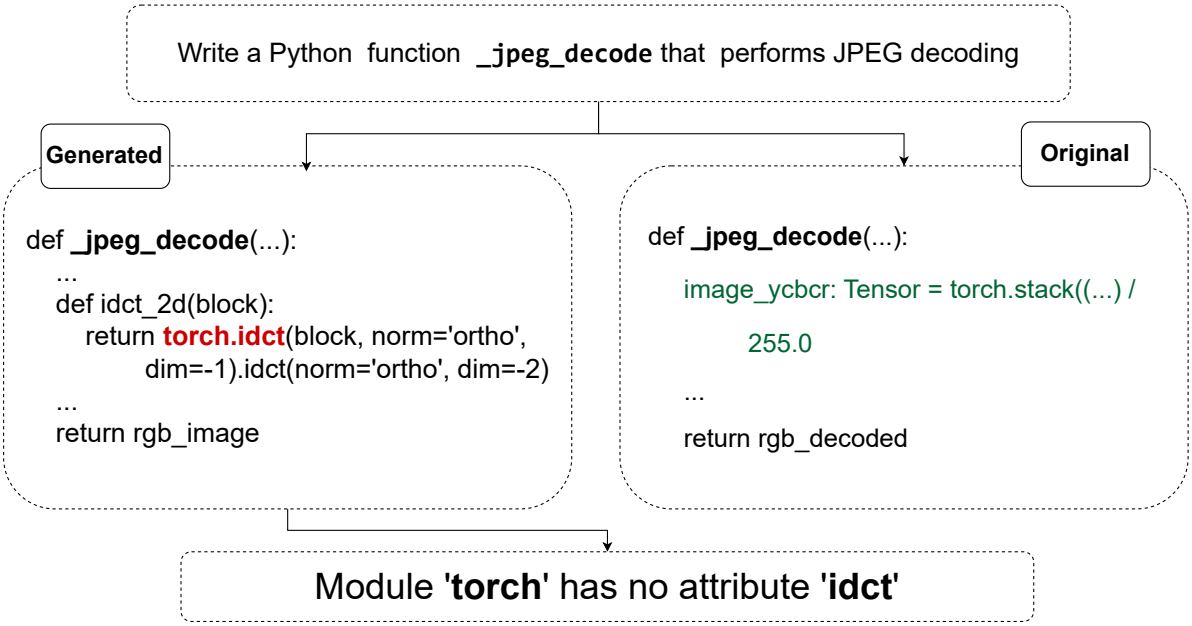


Figure 2.13: Wrong usage of a third-party library.

Example 3: Figure 2.13 provides an example of API misuse in LLM-generated code where GPT-4o attempts to call *torch.idct*, which is not implemented in PyTorch. One possible fix is to provide more context concerning third-party libraries. For example, one could hint to LLMs to use *scipy* instead, resulting in *scipy.fftpack.idct(x.numpy(), norm=norm)* instead.

2.9 Discussion: DL-Bench in practice

One usage of the categorized data in DL-Bench is to train classifiers that can provide DL-specific categorization for other unlabeled datasets(e.g., DS-1000) to improve their quality. To test this potential usage, we train a BERT classifier to predict the stage-in-pipeline for each input prompt. The classifier uses the BERT tokenizer, BERT encoder, and a linear classifier. The optimization is performed with AdamW ($\eta = 2\times10^{-5}$, $B = 8$, $E = 10$), and five-fold cross-validation confirms stable generalization (average weighted $F_1 = 0.56 \pm 0.06$).

Table 2.9: Distribution of stages in DL/ML pipeline for DS-1000 (predicted) and DL-Bench (actual).

Stage in pipeline	**Pre/Post Processing**	**Model Construction**	**Training**	**Inference**	**Evaluation**
DL-Bench (actual)	210(40.3%)	119(22.8%)	75(14.4%)	59(11.3%)	57(10.9%)
DS-1000 (predicted)	932(93.2%)	35(3.5%)	14(1.4%)	14(1.4%)	5(0.5%)

To verify the accuracy, we conducted manual labeling of 100 instances, which shows our classifier has a high accuracy of $95.0 \pm 5.3\%$ (with 99% confidence). This indicates a high level of generalization of DL-Bench categorization when applied to other DL-related benchmarks. Table 2.9 presents the predicted distribution for DS-1000 as well as the actual distribution for DL-Bench. The distribution differences further distinguish DL-Bench and DS-1000. Where DS-1000 mainly focuses on pre/post-processing, DL-Bench contains data from all stages of the DL/ML pipeline.

Finding 8: DL-Bench could complement other DL-related benchmarks by providing training data for categorization classification. Such a stage classifier can have a high accuracy ($95.0 \pm 5.3\%$ at 99% confidence when extending DS-1000).

Few-shot prompting emerged as a way to improve vanilla zero-shot prompting. However, guided shots from the same code category could potentially provide even more uplift in performance. To gauge the potential, we perform a preliminary experiment with three-shot

Table 2.10: Pass@1 rates for improved prompting techniques with DL-Bench's insight.

Dataset	Prompting Technique	O3-Mini	DeepSeek-V3	QwenCoder	GPT-4o	Claude 3.5 Sonnet	Llama 3.1 70b	Mixtral 8*22B	Avg.
DS-1000	Zero-Shot	61.0	61.7	57.3	51.1	61.9	40.9	39.3	53.3
	Three-Shot	50.2	62.6	57.5	54.2	64.8	51.4	42.0	54.6
	Stage-Predicted Three-Shot	54.3	64.1	58.9	57.7	66.4	54.0	47.6	57.5
DL-Bench	Zero-Shot	36.9	31.4	23.4	31.2	30.3	27.5	26.8	29.6
	Three-Shot	37.1	32.3	24.5	33.4	32.3	27.8	27.2	30.7
	Stage-Predicted Three-Shot	38.2	34.1	26.3	35.2	33.6	29.6	28.1	32.2

prompting where the shots are random reference samples, or samples in the same DL stage as the question. Since the stage the prompt belongs to is not available, we use the previously described classifier to predict the stage.

Table 2.10 shows the pass@1 rate for SOTA LLMs using the three prompting approaches on DL-Bench and DS-1000. For three-shot prompting, we perform the experiment twice and present the average pass@1 rates. Zero-shot, without any examples, performs the worst with an average pass@1 rate of 53.3% on DS-1000 and 29.6% on DL-Bench. By including three examples, three-shot prompting has a better average pass@1 rate of 54.6% on DS-1000 and 30.7% on DL-Bench. When providing shots for each prompt, we made sure that the shots do not overlap with the prompt. When each query is paired with snippets that belong to the same predicted stage of the DL pipeline, the pass@1 rate improves significantly. Averaged across models, stage-predicted three-shot prompting yields a **4.2%** and **2.6%** boost over zero-shot in DS-1000 and DL-Bench. This indicates the value of having lower granularity categorization in a dataset, which can enable more sophisticated prompting and fine-tuning techniques, which in turn provide uplift in LLMs' performance.

Finding 9: Classifiers built using DL-Bench categorization data can provide targeted shots in few-shot prompting to improve code generation performance. Overall, stage-predicted three-shot yields up to **4.2%** and **2.6%** boost over zero-shot techniques in DS-1000 and DL-Bench respectively.

2.10 Discussion

2.10.1 Data Leakage Considerations

A critical threat to the validity of any benchmark for code generation is data leakage, where evaluation instances overlap—directly or indirectly—with the training data of the models under study. Such leakage can lead to inflated performance scores, since the model may simply memorize or reproduce previously seen solutions rather than demonstrating genuine reasoning or generalization abilities. Data leakage can arise through multiple channels, including the presence of identical or near-duplicate code snippets in public repositories, inadvertent inclusion of examples that were part of a model's pre-training corpus, or reliance on widely circulated tutorials and libraries that appear verbatim in the benchmark.

To mitigate these risks, the DL-Bench construction process explicitly incorporated safeguards against contamination. Candidate GitHub projects were selected and filtered so that only repositories updated after the training cut-off date of GPT-4o were considered, reducing the likelihood that benchmark code had been exposed to the models being evaluated. In addition, manual curation by multiple annotators ensured that each function under test represented non-trivial, domain-specific deep learning tasks, rather than common boilerplate or textbook examples. These measures strengthen the benchmark's ability to provide a fair assessment of LLM capabilities by ensuring that performance reflects true code synthesis rather than simple retrieval.

Despite these precautions, it remains important for future researchers to remain vigilant. As LLMs continue to be trained on ever-growing corpora and as open-source deep learning projects evolve, the risk of unintentional leakage persists. Periodic re-evaluation of the dataset's provenance and the maintenance of a "live" benchmark—where tasks are refreshed with code produced after each major model release—are recommended strategies to sustain the integrity and credibility of DL-Bench over time.

2.10.2 Benchmark Comparison and Evaluation Criteria

When introducing a new benchmark, it is essential to justify how it advances the state of the art relative to existing alternatives. A benchmark can be considered stronger or more informative than others when it satisfies several key criteria:

- **Domain Relevance and Novelty:** The tasks should capture the unique characteristics of the target domain. In the context of deep learning code generation, DL-Bench was explicitly designed to include function-level problems that require handling DL-specific constructs such as tensor shape manipulation, framework-specific APIs, and training or inference routines. Comparative analyses of prompt and code embeddings show that DL-Bench occupies a semantically distinct region compared to DS-1000 and AICoderEval, confirming its novelty and domain focus.

- **Challenge and Complexity:** A benchmark is more effective when it reveals weaknesses of current models. DL-Bench contains functions with higher lines of code and greater cyclomatic and cognitive complexity than related datasets, making it more demanding and better suited to stress-test the reasoning and synthesis capabilities of large language models.

- **Comprehensive and Fine-grained Coverage:** Beyond raw difficulty, DL-Bench categorizes each instance by pipeline stage, machine learning task, and input data type. This tri-axis labeling allows a more granular evaluation than DS-1000 or MLE-bench, enabling researchers to pinpoint model strengths and weaknesses across specific aspects of the DL workflow.

- **Robustness of Evaluation:** Finally, a superior benchmark provides reliable and reproducible metrics. DL-Bench includes multiple independent test cases per instance

and reports stable pass@k results with low variance, ensuring that model rankings and conclusions drawn from the benchmark are consistent across repeated runs.

By satisfying these criteria, DL-Bench offers a more rigorous and insightful evaluation of LLM performance in deep learning code generation than prior benchmarks. It not only highlights the absolute performance gap between general and DL-specific tasks, but also supports fine-grained analysis of model capabilities and weaknesses.

2.11 Related Works

Code Generation Benchmarks for Data Science and ML/DL: Several benchmarks have been developed to evaluate code generation models in the context of data science and ML/DL tasks. JuICe Agashe et al. [2019], PandasEval and NumPyEval Zan et al. [2022], and JuPyT5 Chandel et al. [2022] provide datasets from Jupyter notebooks or data science libraries, with a focus on realistic usage scenarios. However, most of these benchmarks rely on similarity-based metrics such as BLEU for evaluation, due to the lack of accompanying test cases. In contrast, DL-Bench includes multiple assert-based test cases for each entry, enabling more reliable evaluation via metrics like pass@1.

JuPyT5 introduces the DSP benchmark with 1119 pedagogically curated problems featuring markdown context, assert-based unit tests, and implicit data dependencies, making it suitable for evaluating notebook-based code generation. Similarly, CERT provides PandasEval and NumPyEval for structured, API-heavy data science tasks and shows performance gains by anonymizing user-defined elements. JuICe offers a large-scale dataset from Jupyter notebooks with manually curated test sets derived from nbgrader assignments, although its evaluation also depends on similarity metrics.

Shin et al. Shin et al. [2023] focus specifically on ML/DL tasks using JuICe samples but still evaluate with similarity scores, which do not reliably indicate functional correctness.

In contrast, DL-Bench offers task-level test cases for each function, allowing more precise evaluation of LLM performance in ML/DL scenarios.

DS-1000 Lai et al. [2023] collects 1000 data science problems from StackOverflow, primarily focusing on tasks like data preprocessing or transformation, with the support of test cases. However, the tasks are often limited to isolated code snippets rather than complete function-level implementations. MLE-Bench Chan et al. [2024] captures ML engineering workflows in the context of Kaggle competitions, focusing on end-to-end pipelines but lacking fine-grained test-based evaluation. AICoderEval Xia et al. [2024] further abstracts the evaluation to workflow-level code generation, treating setup and implementation as a black-box output, which can obscure the model's capabilities at the component level.

General Code Generation Benchmarks: Benchmarks like HumanEval Chen et al. [2021], MBPP Austin et al. [2021], APPS Hendrycks et al. [2021], AiXBench Hao et al. [2022], MultiPL-E Cassano et al. [2022], Spider Yu et al. [2018], CoderEval Yu et al. [2024], and RepoEval Zhang et al. [2023] have been widely used to evaluate LLMs on general-purpose programming. These benchmarks span various tasks such as competitive programming, repository-level generation, multi-language support, and SQL query generation from natural language. However, they are primarily focused on general programming capabilities and do not capture the domain-specific challenges of ML/DL code.

Distinctive Features of DL-Bench: DL-Bench differs from prior work in several key ways. First, it focuses exclusively on ML and DL software development tasks, offering function-level prompts that reflect real needs in the ML pipeline. Second, it categorizes each function based on the ML pipeline stage (e.g., preprocessing, model training), task type (e.g., classification, regression), and input data type (e.g., image, text, tabular), offering a richer annotation scheme. Third, unlike most benchmarks, DL-Bench is sourced from real GitHub repositories, ensuring practical relevance, and each entry includes assert-based test cases, enabling robust and reproducible evaluation using metrics like pass@1.

Overall, DL-Bench complements existing benchmarks by providing a granular, test-driven, and ML/DL-focused dataset that enables more realistic evaluation of LLMs in domain-specific development scenarios.

2.12 Conclusion

In this work, we introduce DL-Bench, a benchmark for deep learning tasks related to code generation. The dataset comprises 520 instances, gathered from the most starred and recently updated GitHub repositories. We categorize the data based on the pipeline stage, ML task, and input data type. Additionally, our quantitative analysis of the performance of four state-of-the-art LLMs on DL-Bench reveals that DL code generation is challenging and DL-Bench can provide more insight to help improve the generation process. Using our taxonomy of issues found in LLM-generated DL code, the qualitative analysis reveals the distinct challenges that LLMs face when generating DL code compared to general code as well as the similarities and differences between human-written and LLM-generated DL code. Our discussion shows potential usages of DL-Bench's categorization data outside of benchmarking usages. DL-Bench's data is available in our Kaggle repository.

Chapter 3

Mutation Based Code Generation

3.1 Introduction

Automated code generation for deep learning (DL) pipelines has emerged as a particularly demanding setting for large language models (LLMs). Unlike generic programming tasks, DL code involves specialized operations such as tensor manipulation, model construction, and training logic, where even minor specification ambiguities can result in invalid or non-functional implementations Xu et al. [2022]. Prior studies show that while LLMs achieve encouraging performance on algorithmic or repository-style benchmarks Chen et al. [2021], Nijkamp et al. [2022], their accuracy drops substantially on domain-specific benchmarks like DL-Bench, where tasks require chaining ML primitives with strict library and API conventions Liu and Zhang [2018], Du et al. [2024]. This gap reflects the broader challenge of producing a single correct solution: even models with high pass@k scores often fail at pass@1, leaving practitioners without a reliable output Lyu et al. [2025], Odeh et al. [2024].

To mitigate this issue, prior work has explored leveraging generated test cases (e.g., CodeT Chen et al. [2022]) and consensus-based algorithms Bonandin and Herty [2024], Derpanis [2010] to identify functionally consistent solutions without requiring human-written

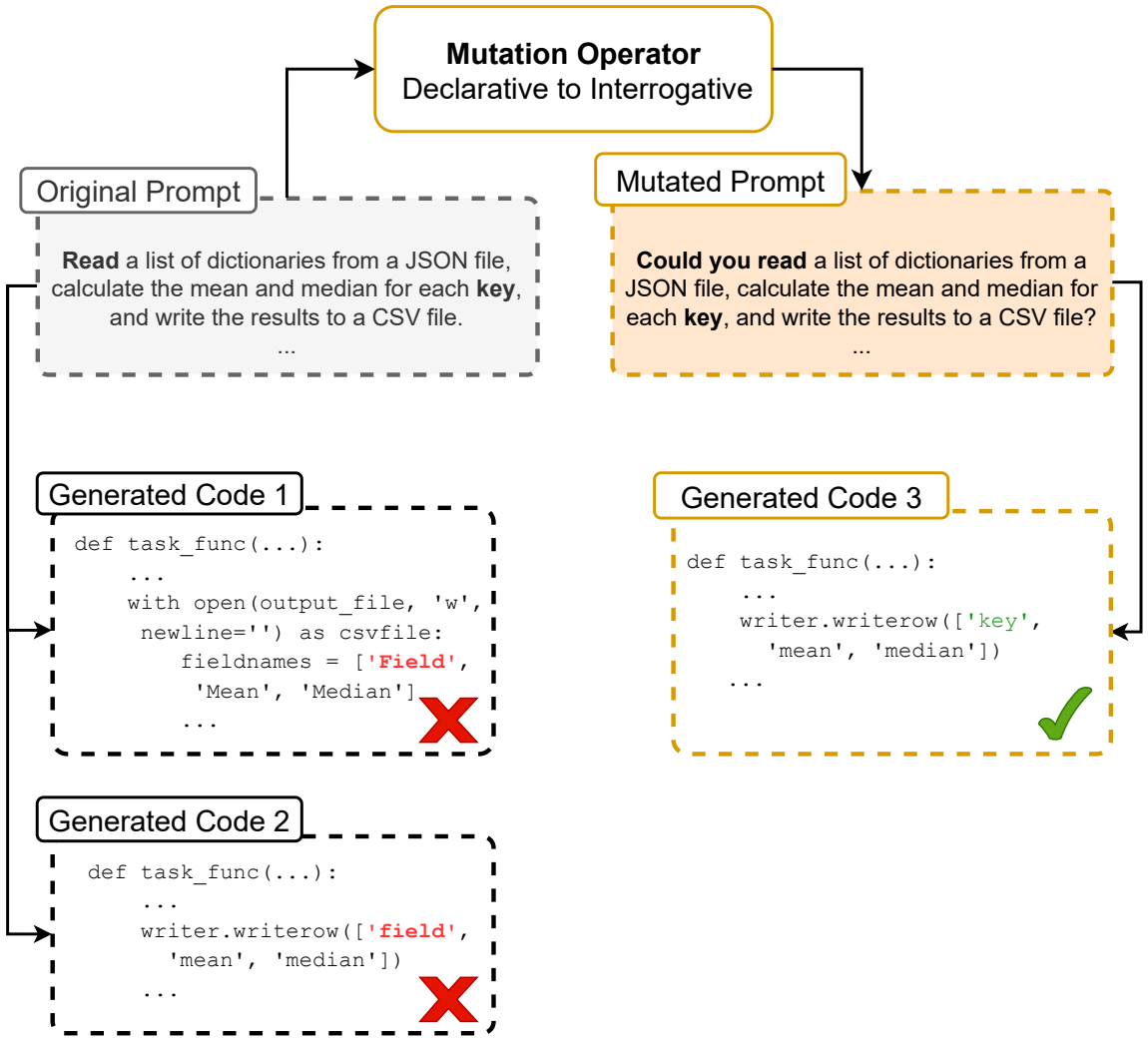


Figure 3.1: A code generation example on O4-Mini where multiple queries of the original prompt failed, but a change from declarative to interrogative voice triggered a better solution that successfully generated code with a corrected CSV header.

tests. However, these frameworks typically depend on a single natural language specification, meaning that any misinterpretation propagates through all generated code Becker et al. [2023], Ridnik et al. [2024]. Our analysis of DL-Bench confirms this weakness: single-prompt generation frequently fails due to subtle specification mismatches, such as incorrect parameter naming or output formatting, which prevent otherwise valid solutions from passing validation.

We propose Mad-Gen, which augments dual-execution agreement with metamorphic prompt mutations, systematic linguistic transformations that preserve semantics while diversifying phrasing Wang et al. [2024]. These include grammatical shifts (e.g., active to passive), structural reformulations (e.g., declarative to interrogative), and lexical substitutions (e.g., verb-to-similar-verb). Each mutation produces a new specification variant that elicits diverse code and test candidates. By clustering candidates based on their pass patterns, Mad-Gen identifies the most consistent group as the final output. This design builds on the intuition that correct implementations remain invariant under such controlled variations, whereas incorrect solutions often diverge.

Our DL-Bench experiments with three representative LLMs, O4-Mini, DeepSeek R1 Basic, and Gemini 2.5 Pro, demonstrate that prompt mutations consistently improve pass@1 performance. Individual transformations such as Active-to-Passive, Lowercase-to-AllCaps, and Verb-to-Similar-Verb each yield gains of up to +2.5, while combining all mutation types achieves the strongest overall improvements, with Gemini 2.5 Pro rising +2.9. These findings show that mutation strategies provide three key benefits for DL-specific code generation: (1) **syntactic diversity**, reducing overfitting to a single phrasing, (2) **semantic clarification**, disambiguating subtle task requirements, and (3) **robustness to specification noise**, enabling recovery from misleading or incomplete instructions.

Generating diverse code candidates is especially important for the dual-execution agreement framework, since its ability to identify truly correct implementations relies on comparing multiple independently generated solutions. Without sufficient diversity the agreement step cannot separate coincidental correctness from genuine functional consistency. A simple baseline—directly asking an LLM to produce eight candidate programs in a single run (see Table 3.3, **CodeT (1 run ask for 8)** column)—illustrates this challenge: pass@1 accuracy drops markedly (e.g., O4-Mini falls to 26.3%), showing that naively requesting many solutions at once does not provide the needed variety or reliability. This motivates the use of controlled prompt mutations in out approach, which generate structured and meaningful diversity that dual-execution agreement can exploit effectively.

3.2 Background

Selecting the best code solution from multiple candidates generated by LLMs remains a persistent challenge in automated code generation. Existing approaches such as CODET Chen et al. [2022] address this by using the same pre-trained model to automatically generate test cases for each problem. CODET reduces the need for costly, human-crafted tests by instructing the model to create plausible input-output examples tailored to the problem's specification. Each generated code solution is then executed against these test cases in a *dual execution agreement* process inspired by RANSAC. This process groups code solutions by shared functionality based on the tests they pass and selects the most consistent, high-quality candidates. By ranking consensus sets formed through this agreement, CODET improves the reliability of selecting correct solutions even without access to ground-truth test suites.

However, while CODET automates test generation and solution ranking, it still assumes the prompt describing the problem is fixed and well-formed. Real-world problem descriptions can vary widely in phrasing, grammar, and structure!Adlakha et al. [2024]. Inspired by ideas

explored in prompt mutation frameworks such as Promptbreeder Fernando et al. [2023], we recognize that LLM performance can be highly sensitive to subtle changes in prompt wording Lu et al. [2021], Reynolds and McDonell [2021]. Prompt mutation introduces controlled, systematic variations of the original problem description to assess and improve model robustness. Rather than relying on a single prompt, we generate a diverse set of semantically equivalent variants through transformations such as paraphrasing, changing grammatical voice, altering question form, and renaming function identifiers. These mutations simulate the variability encountered in real-world specifications, helping expose brittle behaviors and testing the model's ability to generalize across linguistic forms.

Our approach uses GPT-4o as the mutation engine to automatically produce these prompt variations in a consistent, reproducible manner. For each mutated prompt, we generate both code solutions and corresponding test cases using the same model, ensuring that test generation remains closely aligned with the variant's phrasing and intent. This process extends to the methodology used in CODET by introducing diversity at the prompt level. It encourages the model to explore a wider space of candidate solutions and better cover edge cases that a single phrasing might miss. By evaluating all generated code against every generated test in a dual execution agreement framework, we can identify solutions that are not only functionally correct but also robust to variations in how the problem is described.

Ultimately, this mutation-driven pipeline is designed to improve LLM-based code generation in scenarios where formal specifications are variable and human-written tests are scarce or unavailable. By combining automated test generation, prompt mutation, and agreement-based evaluation, our method aims to produce reliable, generalizable code solutions across diverse benchmarks without the need for hand-crafted validation resources.

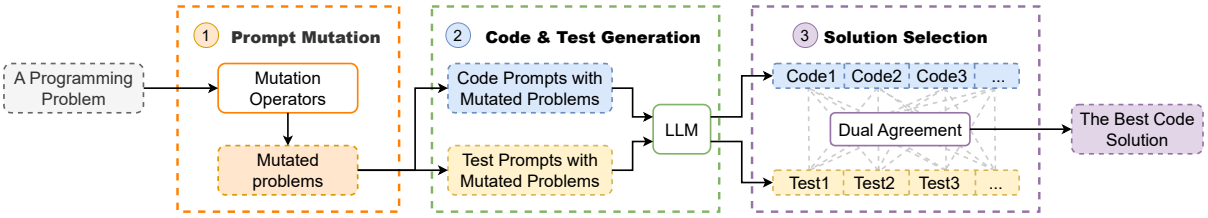


Figure 3.2: Overview of Mad-Gen

3.3 Approach

Figure 3.2 demonstrates MAD-Gen's process. First, given a programming problem, MAD-Gen performs (1) Prompt Mutation. Using the original and mutated problems, MAD-Gen then queries the LLM to execute (2) Code & Test Generation. Finally, by executing the generated test cases and applying dual agreement ranking, MAD-Gen identifies the best code solution in the (3) Solution Selection step.

Table 3.1: Examples of mutation types, their prompt instructions, and corresponding input–output transformations applied to modify code-generation prompts.

Mutation Type	Mutation Prompt	Example Input	Example Output
Active to Passive	Change the following sentence to passive voice	Write a Python function cot_laplacian that computes ...	A Python function cot_laplacian should be written to ...
Declarative to Interrogative	Change the following sentence to interrogative voice	Generate a Python function called top_k_accuracy ...	Is it possible to generate a Python function called top_k_accuracy ...
Lowercase to All_Caps	Change the following sentence to uppercase letters	Generate a Python function called hinton ...	GENERATE A PYTHON FUNCTION CALLED HINTON ...
Adversarial Function Name	Modify the following sentence to refer to a function with a misleading name that doesn't represent the actual task	Generate a Python function called get_perspective_transform ...	Generate a Python function called compute_angle_of_projection ...
Task Function Name	Modify the following sentence to refer to a function with a descriptive name that represents the task	Generate a Python function multinomial_sampling ...	Generate a Python function sample_ratings_multinomially ...
Rephrase Prompt	Rephrase the following sentence	Reads data from a CSV file and generates ...	The function ingests data from a CSV file and constructs ...
Verb to Similar Verb	Change the verbs of the following sentence to a similar verb	Write a python function for following task: Describe ...	Construct a Python function for the subsequent task: Illustrate ...

(1) Prompt Mutation. One of the advantages of the dual agreement selection process is that it can select the best solution by verifying the majority agreement of code solutions with generated test cases. In this way, if the generated code solutions are not diverse enough, the selection process may converge on incorrect outputs. As the example in Figure 3.1 demonstrates, a metamorphic mutation of the problem prompt can provide diversity in the generated code and help LLMs produce additional correct solutions that ultimately improve accuracy within DL-Bench.

In this step, MAD-Gen generates multiple diverse candidate solutions utilizing various metamorphic mutation operators. These mutation operators aim to vary the prompt styling,

phrasing, syntactic structure, or context. These variations can trigger changes in the generated code, which can improve the pool's overall correctness, efficiency, and readability. Specifically, MAD-Gen applies seven **prompt mutation** operators to the original problem description:

- **Lowercase to All_Caps**: Formatting variations such as converting all text to uppercase are introduced to test sensitivity to casing.
- **Verbs to similar verbs**: Key verbs are replaced with synonyms or near-synonyms, introducing lexical variation without changing the underlying task.
- **Active verb to passive**: Sentences are rewritten from active to passive voice, altering phrasing while preserving meaning.
- **Declarative to interrogative**: Statements are rephrased as questions (e.g., "Implement a function" $\rightarrow$ "Can you implement a function?"), introducing subtle changes in instruction style.
- **Rephrase the original prompt**: Paraphrasing is used to restate the problem with different wording while preserving intent.
- **Adversarial function name**: Misleading or confusing function names are used to assess model robustness to naming variation.
- **Change the task function name**: Neutral function names are systematically introduced to test naming sensitivity.

(2) **Labeling.** For creating these mutations, we employ **GPT-4o**, an LLM capable of generating high-quality text variations. Once the mutated prompts are generated, it is crucial to ensure they remain valid representations of the original problem. To verify this,

three co-authors performed **manual labeling** of a sample of 360 randomly selected mutated prompts. They evaluated each mutation according to two criteria:

1. **Semantic equivalence**: Whether the mutated prompt expresses the same meaning as the original, ensuring the intended task is preserved.

2. **Correct application of the mutation**: Whether the intended mutation type was properly and clearly applied to the prompt.

Through this process, we curated a set of verified prompt variations for each DL-Bench problem description. These multiple versions of the prompt are then used to generate different code solutions, promoting diversity in the candidate pool and increasing the likelihood of finding correct and robust implementations.

③ **Code and Test Generation.** The challenge then becomes selecting the most effective solution from this set, the one most likely to work correctly. To tackle this, we adopt the insight from prior work which demonstrates the value of using the pre-trained model itself to generate test cases that align with the problem description. We extend this idea to DL-Bench by using the same set of mutated prompts not only to generate diverse candidate solutions but also to generate corresponding test cases. This ensures that the tests are relevant, tailored to the intended functionality, and systematically evaluate correctness.

With both the candidate solutions and the generated test cases in hand, we evaluate each solution by running it against these tests. We then choose the one that performs best using a **dual execution agreement** method. This ensures that the selected solution is not only executable but also successfully passes the generated tests, improving the chances that it is both correct and reliable within DL-Bench.

3.3.1 TEST CASE GENERATION

In addition to generating code solutions, it is crucial to create test cases that can verify correctness. A **test case** includes a specific input along with the expected output for the function defined in the problem context. For example, for a DL-Bench task that asks whether there are elements in a list that are closer than a given threshold, a test case would provide an input list and threshold value, as well as the correct Boolean result.

To generate these test cases, we use the same pre-trained language model employed for producing code solutions. Instead of asking it to write code that solves the problem, we adapt the task by reusing our collection of **mutated prompts** to request test cases instead. By applying the same mutations we used for generating varied code solutions, we ensure that the test case prompts stay consistent with the original problem descriptions while introducing enough variation to cover different scenarios and phrasings.

By supporting multiple formats, we ensure that DL-Bench can interoperate with other frameworks while retaining its own `pytest`-based design. This enhances DL-Bench's role as both a rigorous evaluation benchmark and a flexible testbed for robustness-oriented methods.

④ **Dual Execution Agreement.** We rely on the dual execution agreement strategy described in prior research, which provides a principled way to evaluate code solutions against automatically generated test cases. The central insight is that correct code should consistently pass the relevant test cases, while incorrect code tends to fail or behave inconsistently.

The process involves executing all candidate code solutions against all generated test cases, grouping solutions by the set of tests they pass, and then ranking groups based on breadth (number of tests passed) and consistency (agreement with other solutions). This consensus-based ranking allows us to identify robust solutions more effectively than relying on a single prompt–solution pair.

For our experiments, we implement this approach directly in DL-Bench. All generated solutions are systematically executed against DL-Bench's test cases, results are logged, and agreement scores are calculated to select the most reliable candidate. This ensures that the final solution is not only syntactically valid but also functionally correct within DL-Bench's evaluation framework.

3.4 Experimental Setup

3.4.1 Models

We evaluate three state-of-the-art code generation models to ensure a robust comparison across different architectures and design philosophies. The first model, **o4-mini**, is a compact variant from the OpenAI code-generation family, selected for its balance of generation quality and inference speed. The second model, **DeepSeek R1 Basic**, represents the baseline edition of the DeepSeek R1 series and is known for its aggressive decoding strategy tailored specifically for code tasks. The third model, **Gemini 2.5 Pro**, is Google's latest large language model designed with extended context window support and specialized optimizations for code generation. For fair evaluation, all models are queried in zero-shot mode using identical decoding parameters, with a temperature of 0.7 and top-k sampling set to 50, ensuring that any observed performance differences reflect the underlying model capabilities rather than tuning biases.

3.4.2 Benchmarks

Our experiments are centered on **DL-Bench**, a deep-learning–specific benchmark that covers model definitions, training loops, tensor manipulation, and data pre/post-processing tasks.

DL-Bench provides the most direct insight into LLM capabilities in real-world DL workflows, where correctness and framework-specific details are critical.

3.4.3 Prompt Mutation Pipeline

To better simulate the variability found in real-world problem descriptions and to reduce the risk of models overfitting to specific prompt templates, we employ a systematic prompt mutation pipeline. For each original DL-Bench problem statement, we generate eight distinct variations using GPT-4o as our mutation engine. GPT-4o is prompted with carefully designed instructions to automatically produce meaningful variations that maintain the original problem's intent while introducing controlled differences in phrasing and structure.

These mutations include transformations such as converting active voice to passive voice and vice versa, rephrasing declarative statements as interrogatives, toggling between lowercase and uppercase formatting, replacing verbs with semantically similar alternatives, and paraphrasing sentences to restructure them while preserving meaning. Additional mutations involve substituting function names with adversarial or misleading alternatives to test naming robustness and uniformly changing function names across prompts and tests to simulate API changes.

By introducing these systematic variations, we generate a diverse set of prompts for each DL-Bench problem. This diversity exposes brittleness in model reasoning, improves evaluation realism, and increases the likelihood of identifying correct solutions.

For evaluating the quality of generated code, we adopt the **pass@k** metric, which measures the probability that at least one of the top-k generated solutions correctly solves the given problem. This metric is widely used in code generation research because it reflects practical use cases where developers review multiple suggestions from the model to select a correct or preferred solution. By combining GPT-4o-based prompt mutation with pass@k-based

evaluation, our experimental setup rigorously tests model robustness to prompt variability while fairly assessing solution quality, with DL-Bench serving as the primary benchmark.

3.4.4 Research Questions

Our study is guided by two central research questions that aim to investigate the role of prompt mutation in DL-specific code generation with large language models (LLMs) and its practical impact on model performance.

RQ1: How well do LLMs handle variations in DL-Bench problem descriptions generated through prompt mutations? This question examines the robustness of code generation models when faced with systematically mutated versions of the same DL-Bench prompt. Since real-world DL specifications are rarely phrased identically and may vary in wording, grammatical form, naming conventions, or instruction style, understanding how models respond to these variations is critical. We analyze the extent to which LLM-generated DL code remains correct and consistent across these diverse prompt formulations.

RQ2: How can prompt mutations improve the overall quality and reliability of DL-specific code solutions? This question explores whether introducing prompt diversity within DL-Bench can help models produce better code by encouraging varied candidate solutions and uncovering edge cases that a single prompt might miss. By generating multiple mutated prompts for each DL-Bench problem and aggregating their outputs, we evaluate whether this strategy increases the likelihood of finding correct solutions, especially when combined with the pass@k metric. This approach also allows us to test whether mutations can help mitigate model over-reliance on specific templates or phrasing, ultimately leading to more robust and generalizable DL code generation.

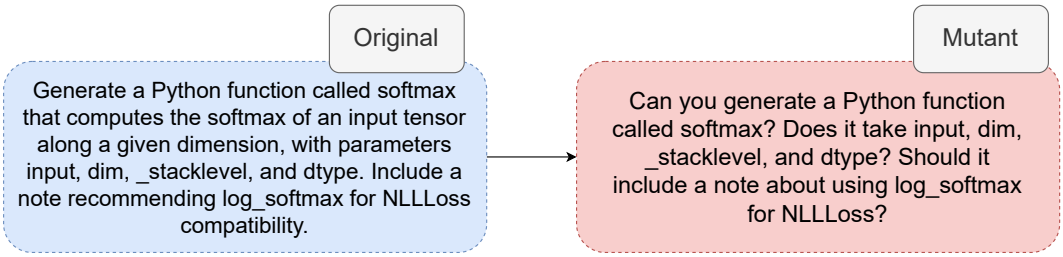


Figure 3.3: Example of prompt mutation that alters intent by turning clear instructions into questions.

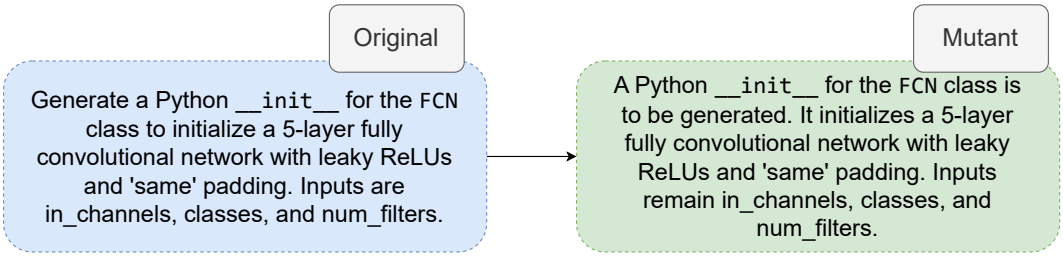


Figure 3.4: Example of prompt mutation that preserves meaning despite stylistic rephrasing.

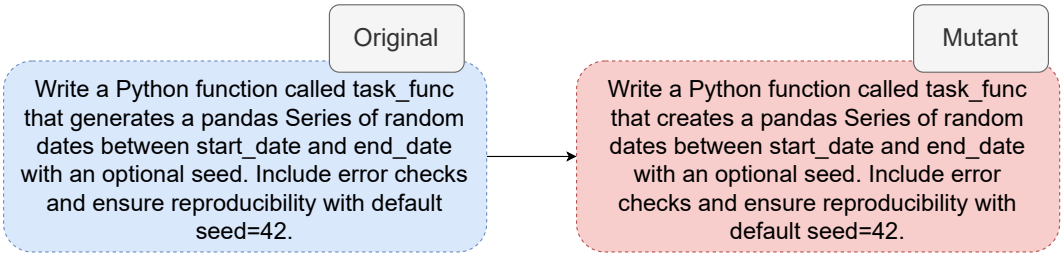


Figure 3.5: Example of adversarial function name mutation that failed to apply the intended renaming, leaving the function name unchanged and thus not introducing any adversarial variation.

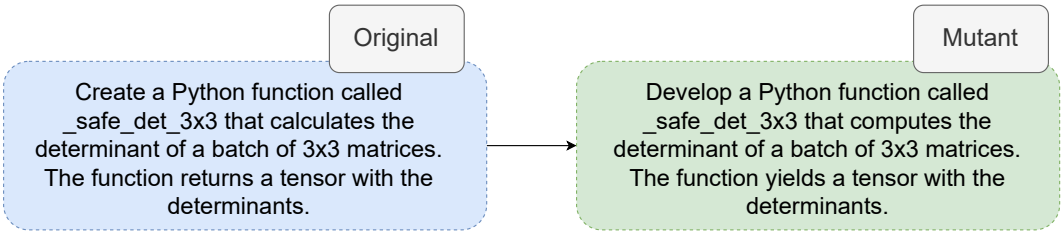


Figure 3.6: Example of a verb-to-similar-verb mutation successfully applied by substituting synonyms while preserving the original prompt's meaning and intent.

Table 3.2: Manual evaluation of GPT-4o's prompt mutation quality across seven mutation types. Each mutation was rated for semantic equivalence to the original prompt and correctness in applying the intended transformation. Results show strong performance overall but highlight challenges for transformations that shift sentence modality or naming consistency.

Mutation Type	**Semantically Equivalent**	**Mutation Applied Correctly**
Active to Passive	97.9	95.9
Lowercase to All_Cap	97.6	97.6
Declarative to Interrogative	79.6	79.6
Verb to Similar Verb	98.9	96.7
Rephrase Prompt	97.1	97.1
Adversarial Function Name	100.0	88.4
Task Function Name	100.0	100.0

Table 3.3: Pass@1 (%) results on **DL-Bench** across mutation types for three LLMs. Column short forms: Orig (Original baseline), CodeT, +A2P (Active to Passive), +D2I (Declarative to Interrogative), +LC2AC (Lowercase to AllCaps), +AFN (Adversarial Function Name), +TFN (Task Function Name), +RP (Rephrase Prompt), +V2SV (Verb to Similar Verb), All (all mutations combined).

Dataset	LLM	Orig	CodeT	CodeT (1 run ask for 8)	+A2P	+D2I	+LC2AC	+AFN	+TFN	+RP	+V2SV	All
DL-Bench	O4-Mini	35.4	37.4 (+2.0↑)	29.4	37.4 (+2.0↑)	36.3 (+0.9↑)	37.9 (+2.5↑)	36.9 (+1.5↑)	36.9 (+1.5↑)	36.1 (+0.7↑)	37.4 (+2.0↑)	37.9 (+2.5↑)
	DeepSeek R1 Basic	34.1	35.4 (+1.3↑)	28.3	36.1 (+2.0↑)	34.8 (+0.7↑)	35.4 (+1.3↑)	34.6 (+0.5↑)	34.1 (—)	33.8 (-0.3↓)	34.6 (+0.5↑)	35.4 (+1.3↑)
	Gemini 2.5 Pro	35.4	36.2 (+0.8↑)	25.2	36.6 (+1.2↑)	37.9 (+2.5↑)	35.6 (+0.2↑)	35.0 (-0.4↓)	34.4 (-1.0↓)	36.2 (+0.8↑)	37.9 (+2.5↑)	38.3 (+2.9↑)

3.5 Result

3.5.1 How is the Performance of GPT-4o on Doing Mutation?

To evaluate the quality and reliability of our prompt mutation pipeline, we conducted a detailed manual analysis of 360 mutated prompts generated by GPT-4o across seven distinct mutation types. Each mutation was evaluated along two dimensions: whether the mutated prompt remained semantically equivalent to the original prompt—preserving its meaning and task intent—and whether the mutation was applied correctly according to the specified transformation type. Table 3.2 reports the proportion of instances meeting each criterion for every mutation class.

We assessed inter-rater reliability among three reviewers using observed agreement percentages Viera et al. [2005]. For the satisfaction mutation-type task, reviewers achieved 93.2% observed agreement, and for semantics preservation, they achieved 88.9% observed agreement. These high levels of agreement indicate substantial consistency among reviewers on both evaluation criteria. Because of the prevalence effect—where one response category dominates the ratings—chance-corrected reliability measures are less meaningful for our data [Feinstein and Cicchetti, 1990, Byrt et al., 1993]. With 92.9% and 85.5% positive ratings for the two tasks, respectively, simple observed agreement offers a more interpretable measure of reviewer consensus than chance-corrected statistics, which can be artificially lowered when category distributions are highly skewed [Gwet, 2008].

Overall, GPT-4o demonstrates strong performance across most mutation types, with semantic equivalence rates exceeding 95% for six of the seven categories. For example, Lowercase to Uppercase and Rephrase Prompt mutations achieved near-perfect semantic equivalence (97.6 and 97.1, respectively) and 100% correctness in applying the intended transformation. Similarly, Active to Passive and Verb to Similar Verb mutations each achieved 98.9 and 96.7 in both semantic equivalence and correctness. These high scores suggest that GPT-4o handles both surface-level changes and fine-grained lexical or grammatical edits exceptionally well. Figure 3.4 illustrates a success case of the Active to Passive mutation, where the instruction to implement an `FCN` class constructor retained all task-relevant details despite the shift in grammatical voice. Another success is shown in Figure 3.6, where the Verb to Similar Verb mutation replaced verbs like "create" with "develop" while preserving the original task's meaning and intent.

On the other hand, Declarative to Interrogative exhibited the lowest semantic equivalence rate at 79.6, despite achieving perfect correctness for those mutants that did maintain equivalence. This reflects the inherent difficulty of converting clear imperative instructions into interrogatives without changing their directive nature. As shown in Figure 3.3, our

failed `softmax` mutation illustrates this issue: converting a precise generation instruction into a question introduced ambiguity and weakened the instructional force necessary for code generation.

Other mutation types show nuanced behavior. Adversarial Function Name achieved perfect semantic equivalence (100%) but a lower correctness score (88.4), indicating GPT-4o sometimes failed to introduce actual adversarial variation. An example is shown in Figure 3.5, where the mutation left the function name unchanged, offering no robustness benefit. In contrast, Task Function Name mutations performed flawlessly in both metrics (1.00 equivalence and correctness), showing that GPT-4o can handle explicit renaming when task descriptions are clear.

Using a 95% confidence level, the semantics preservation task achieved 95.1% with a margin of error of ±2.1%, and the mutation type satisfaction task achieved 92.8% with a margin of error of ±2.6%. These narrow intervals confirm that our sample size provides sufficient statistical power for reliable conclusions.

Finding1: GPT-4o demonstrates strong overall ability to perform prompt mutations, achieving over **95%** semantic equivalence and correctness in most cases. However, modality-changing transformations—especially Declarative to Interrogative—remain challenging, highlighting the risk of semantic drift.

3.5.2 Impact of Individual Prompt Mutations in Dual Agreement

Table 3.3 reports Pass@1 results on **DL-Bench** when applying each mutation individually, with dual execution agreement selecting between the original and mutated variants. Overall, controlled prompt mutations provide consistent gains, though the magnitude varies by model and mutation type.

For **O4-Mini**, all mutation types lead to improvements over the baseline. The largest gain comes from Lowercase-to-AllCaps (+2.5), followed closely by Active-to-Passive and Verb-to-Similar-Verb (+2.0 each). **DeepSeek R1 Basic** also benefits from Active-to-Passive (+2.0) and Lowercase-to-AllCaps (+1.3), though Rephrase Prompt slightly reduces performance (–0.3). For **Gemini 2.5 Pro**, the strongest gains come from Declarative-to-Interrogative and Verb-to-Similar-Verb (+2.5 each). However, mutations like Adversarial Function Name and Task Function Name reduce accuracy (–0.4 and –1.0).

Finding 2: On **DL-Bench**, individual prompt mutations uncover additional correct solutions. Mutations such as Active-to-Passive, Lowercase-to-AllCaps, and Verb-to-Similar-Verb are consistently beneficial, though adversarial naming can hurt performance.

3.5.3 Combining All Mutations with Broader Dual Agreement

The "All Together" setting combines all mutations, generating diverse variants per task and applying dual execution agreement across them.

On **DL-Bench**, this strategy produces the strongest improvements overall. O4-Mini gains +2.5, DeepSeek R1 Basic improves +1.3, and Gemini 2.5 Pro shows the largest boost (+2.9). These results highlight that pooling multiple controlled variations increases the probability of recovering correct solutions, even in the challenging context of DL-specific code generation.

Finding 3: On **DL-Bench**, aggregating all prompt mutations with dual execution agreement yields the most robust improvements (up to +2.9), demonstrating that diversity in prompt phrasing expands the solution space effectively.

3.6 Limitation & Threats to validity

One threat to validity in our approach arises from the reliance on automatically generated test cases for evaluating code correctness. While using the same language model to produce both solutions and tests reduces the need for human-written suites, it also risks circularity: if the model generates flawed or overly simplistic tests, incorrect solutions may pass evaluation. Although our dual execution agreement method seeks to mitigate this by rewarding solutions that consistently pass many diverse tests, it cannot guarantee absolute coverage of all edge cases or complex requirements found in real-world software. This limitation suggests that while our framework improves over naive generation, it should be seen as complementary to, rather than a full replacement for, expert-designed test cases in critical settings.

A second threat stems from the use of prompt mutation to simulate real-world variation in problem descriptions. While we carefully design a diverse set of mutation types—including paraphrasing, grammatical voice shifts, question transformations, and function renaming—these transformations may not capture the full range or nuance of naturally occurring specification changes. Additionally, GPT-4o, which we use to generate these mutations, may introduce unintended biases or fail to preserve the original task semantics perfectly in all cases. Although we manually validate a sample of mutations for semantic equivalence and correctness of transformation, there remains a risk of introducing subtle changes in meaning that could distort evaluation results or obscure model weaknesses.

Finally, our evaluation relies on benchmarks such as LiveCodeBench, BigCodeBench, and DLBench, each of which has its own assumptions, domain biases, and limitations. While these benchmarks offer complementary coverage—from algorithmic puzzles to real-world API usage and deep learning code—they may not fully represent all software engineering contexts or programming languages. Our use of pass@k as the main evaluation metric, while standard in code generation research, also prioritizes finding at least one correct solution

in multiple samples rather than ensuring consistent high-quality output. Consequently, our results should be interpreted within the scope of these experimental conditions, recognizing that further work is needed to validate prompt mutation and dual execution agreement strategies in broader, more heterogeneous development environments.

3.7 Related work

3.7.1 Code Generation

Code generation has become a prominent area of research, particularly with advancements in LLMs. Dehaerne et al. Dehaerne et al. [2022] provide a systematic review categorizing code generation tasks into three paradigms: description-to-code, code-to-description, and code-to-code transformations. Their review emphasizes the importance of selecting suitable neural architectures like recurrent neural networks (RNNs)Sherstinsky [2020], transformers, and convolutional neural networks (CNNs)Alzubaidi et al. [2021], alongside quality datasets and robust evaluation metrics, to enhance the effectiveness of generated code.

Recent transformer-based models, such as Codex and AlphaCodeLi et al. [2022], have demonstrated remarkable capabilities, particularly in competitive programming. Li et al. Li et al. [2022] introduce AlphaCode, a transformer-based system capable of generating multiple diverse code candidates, which are subsequently filtered and clustered to select optimal solutions. AlphaCode notably achieves human-level performance in competitive programming, illustrating the advanced capabilities of transformer models in solving complex programming challenges.

Fried et al. Fried et al. [2022] propose InCoder, an innovative model capable of both traditional program synthesis and sophisticated code editing through infilling. By conditioning generation on bidirectional context, InCoder significantly improves zero-shot performance

across tasks such as type inference, comment generation, and variable renaming, effectively bridging pure synthesis and code editing tasks. Additionally, Chen et al. Chen et al. [2022] address practical evaluation challenges by introducing CODET, which leverages pre-trained language models for generating both candidate solutions and comprehensive test cases. Using a dual execution agreement approach, CODET substantially enhances solution selection accuracy across benchmarks such as HumanEval and APPS.

Empirical investigations by Yu et al. into the real-world adoption of models like ChatGPT and Copilot on GitHub further highlight their practical utility and current limitations. Their study reveals these tools are predominantly utilized in smaller, individually maintained projects involving simpler, low-complexity code, requiring minimal subsequent modifications. Collectively, these studies underscore significant progress in code generation through advanced machine learning models, while also emphasizing ongoing challenges related to practical applicability, comprehensive evaluation, and quality assurance.

3.7.2 Prompt Mutation

Several works have effectively highlighted the importance of prompt mutation within LLM-assisted test and code generation. The works of Yang et alYang et al. [2024] and Lops et alLops et al. [2024] highlighted the positive impact of including relevant code features within test generation prompts. Lops expanded upon this insight by using skeleton prompts within their testing framework to enforce uniformityLops et al. [2024]. In terms of code generation, Yang et al explore prompt mutation through an iterative, user-involved, requirement refinement methodologyYang et al. [2024].

Although each previous work presents a unique approach to prompt mutation, the work of Chen et alChen et al. [2025] most effectively explores the concept from the perspective of human-induced variance in semantically similar prompts. While this work includes two

paraphrasing mutation classes, it places a greater emphasis upon minor prompt perturbations such as one-character typos and extra spaces. Our work builds upon the notion of real-world variance by placing a greater emphasis on the semantic variation of a set of prompts.

3.8 Conclusion

This chapter explored mutation-based strategies for improving the robustness and reliability of LLM-generated deep learning code. We first introduced a prompt mutation pipeline that systematically transforms input prompts across multiple dimensions (e.g., active-to-passive, declarative-to-interrogative, rephrasing, adversarial renaming). These mutations provide syntactic and semantic diversity while preserving the intended functionality of the task, enabling a broader and more comprehensive evaluation of model performance.

Through extensive experiments, we showed that individual mutation types can substantially influence generation outcomes. Certain mutations, such as declarative-to-interrogative or verb-to-similar-verb transformations, often led to better solutions, while others highlighted model vulnerabilities. When integrated into the dual execution framework, combining multiple mutations consistently improved pass@1 accuracy across diverse benchmarks. This demonstrates that leveraging linguistic and structural variation in prompts can mitigate brittleness and uncover alternative reasoning pathways for LLMs.

Our analysis also revealed limitations. Some mutation types introduced ambiguity or failed to preserve semantic equivalence, leading to inconsistent or degraded outputs. Moreover, while dual execution reduces variance and improves robustness, it does not fully resolve systematic weaknesses in DL-specific code generation, such as shape mismatches or incorrect module usage. These findings underline the need for more principled approaches to mutation design and integration with other robustness-enhancing techniques.

In summary, mutation-based prompting and dual agreement offer a promising complementary strategy to DL-Bench for evaluating and strengthening LLM code generation. By highlighting the interaction between prompt formulation and model performance, this chapter provides both methodological insights and empirical evidence that can guide future research on making LLMs more reliable for deep learning applications.

Chapter 4

Conclusions and Future work

4.1 Thesis Findings and Contributions

This thesis investigated the challenges and opportunities of large language models (LLMs) for deep learning (DL) code generation. The key contributions can be summarized as follows:

- **DL-Bench: A Comprehensive DL-specific Benchmark.** We introduced DL-Bench, the first large-scale function-level benchmark dedicated to DL code generation. It consists of 520 curated instances extracted from 30 GitHub repositories and is systematically categorized along three dimensions: pipeline stage, task type, and input data type. This benchmark provides a semantically distinct and more challenging evaluation space compared to existing datasets such as DS-1000 and AICoderEval, exposing significant performance gaps between general code generation and DL-specific code generation.

- **Fine-grained Categorization and Analysis.** DL-Bench uniquely categorizes each code instance by (i) stage in the ML pipeline (pre/post-processing, model construction, training, inference, evaluation), (ii) DL/ML task type (classification, regression, object

detection, segmentation, recommendation, etc.), and (iii) input data type (text, image, structured arrays, others). This categorization enabled detailed analysis of model strengths and weaknesses, revealing systematic failure modes (e.g., model construction and segmentation tasks are particularly challenging).

- **Empirical Evaluation of State-of-the-Art LLMs.** We conducted extensive experiments across seven representative LLMs, including GPT-4o, Claude 3.5 Sonnet, O3-Mini, LLaMA 3.1 70B, and DeepSeek V3. The results demonstrated that DL-specific problems are significantly harder, with average pass@1 scores dropping from above 50% (on DS-1000) to below 30% (on DL-Bench). Even the strongest open-weight and commercial LLMs failed to achieve acceptable accuracy, underscoring the difficulty of DL code generation.

- **Bug Taxonomy for DL Code Generation.** Through a systematic qualitative analysis, we developed a taxonomy of common bug types in LLM-generated DL code. Beyond general programming issues (e.g., misinterpretation, syntax errors), we identified DL-specific categories such as tensor shape mismatches, incorrect DL module usage, and performance inefficiencies. These findings not only characterize failure patterns but also provide guidance for debugging and improving model design.

- **Mutation-based Prompting and Dual Agreement.** We explored prompt mutation strategies (active $\rightarrow$ passive, declarative $\rightarrow$ interrogative, adversarial renaming, etc.) within a dual execution framework. This approach demonstrated measurable improvements in robustness and accuracy, highlighting the promise of mutation-based evaluation pipelines as complementary tools to DL-Bench.

Taken together, these contributions establish DL-Bench as a challenging and diagnostic benchmark for DL code generation, provide novel insights into LLM failure modes, and propose robustness-enhancing strategies for future LLM evaluation.

4.2 Future Work

While this thesis has made progress in benchmarking and analyzing LLMs for DL code generation, several promising avenues remain for future exploration:

- **Extending DL-Bench.** Increasing the dataset size by incorporating more repositories, programming languages (e.g., Java, Julia), and broader DL frameworks (e.g., TensorFlow, JAX) would improve representativeness. Moreover, integrating multimodal tasks where text, vision, and audio are jointly processed could better reflect real-world DL pipelines.

- **Live Benchmarking and Recency-aware Evaluation.** Code recency strongly influences LLM success rates, as observed in our live DL-Bench evaluation. A future direction is to maintain DL-Bench as a continuously updated "live" benchmark, enabling longitudinal evaluation of models and tracking adaptation to evolving DL libraries.

- **Enhanced Mutation Pipelines.** While initial experiments with prompt mutation and dual execution improved robustness, there is scope for developing systematic mutation grammars, adversarial test generation, and hybrid agreement protocols that combine syntactic and semantic diversity for more resilient evaluation.

- **Beyond Accuracy: Code Quality and Maintainability.** Future studies should evaluate LLM outputs not only for correctness but also for readability, maintainability,

efficiency, and adherence to DL best practices. Incorporating metrics such as maintainability index, memory profiling, and runtime efficiency could yield a more holistic evaluation framework.

- **Human-in-the-Loop Evaluation.** Incorporating expert DL developers into the evaluation pipeline could validate correctness in cases where automated test suites are incomplete and guide improvements to dataset labeling and categorization.

- **Integration with Robustness-Oriented Techniques.** Finally, integrating DL-Bench with other robustness-enhancing strategies such as self-consistency, retrieval-augmented prompting, and program repair could yield stronger pipelines for LLM-based DL code generation.

In summary, this thesis provides a foundation for rigorous and fine-grained evaluation of LLMs in DL code generation. Extending DL-Bench, enriching evaluation criteria, and integrating robustness strategies hold the potential to drive meaningful advances in the next generation of code-capable language models.

Bibliography

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

Vaibhav Adlakha, Parishad BehnamGhader, Xing Han Lu, Nicholas Meade, and Siva Reddy. Evaluating correctness and faithfulness of instruction-following models for question answering. Transactions of the Association for Computational Linguistics, 12:681–699, 2024.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. arXiv preprint arXiv:1910.02216, 2019.

Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. Journal of big Data, 8(1):53, 2021.

Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In 2018 44th euromicro conference on software engineering and advanced applications (SEAA), pages 50–59. IEEE, 2018.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. ArXiv, abs/2108.07732, 2021. URL https://api.semanticscholar.org/CorpusID:237142385.

Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pages 500–506, 2023.

Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934, 2020.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258, 2021.

Sabrina Bonandin and Michael Herty. Consensus-based algorithms for stochastic optimization problems. arXiv preprint arXiv:2404.10372, 2024.

Ted Byrt, Janet Bishop, and John B Carlin. Bias, prevalence and kappa. Journal of clinical epidemiology, 46(5):423–429, 1993.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. arXiv preprint arXiv:2208.08227, 2022.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench:

Evaluating machine learning agents on machine learning engineering. arXiv preprint arXiv:2410.07095, 2024.

Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Training and evaluating a jupyter notebook data science assistant. arXiv preprint arXiv:2201.12901, 2022.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. 07 2022. doi: 10.48550/arXiv.2207.10397.

Junkai Chen, Li Zhenhao, Hu Xing, and Xia Xin. Nlperturbator: Studying the robustness of code llms to natural language variations. ACM Trans. Softw. Eng. Methodol., July 2025. ISSN 1049-331X. doi: 10.1145/3745764. URL https://doi.org/10.1145/3745764. Just Accepted.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. ArXiv, abs/2107.03374, 2021. URL https://api.semanticscholar.org/CorpusID:235755472.

Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. Ieee Access, 10:82434–82455, 2022.

Konstantinos G Derpanis. Overview of the ransac algorithm. Image Rochester NY, 4(1): 2–3, 2010.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers), pages 4171–4186, 2019.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. arXiv preprint arXiv:2308.01861, 2023.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13, 2024.

Alvan R Feinstein and Domenic V Cicchetti. High agreement but low kappa: I. the problems of two paradoxes. Journal of clinical epidemiology, 43(6):543–549, 1990.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155, 2020.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. arXiv preprint arXiv:2309.16797, 2023.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999, 2022.

John Cristian Borges Gamboa. Deep learning for time-series analysis. arXiv preprint arXiv:1701.01887, 2017.

Kilem Li Gwet. Computing inter-rater reliability and its variance in the presence of high agreement. British Journal of Mathematical and Statistical Psychology, 61(1):29–48, 2008.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. Aixbench: A code generation benchmark dataset. arXiv preprint arXiv:2206.13179, 2022.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021.

Nur Farhana Hordri, Siti Sophiayati Yuhaniz, and Siti Mariyam Shamsuddin. Deep learning and its applications: A review. In Conference on Postgraduate Annual Research on Informatics Seminar, pages 1–5, 2016.

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In Proceedings of the 2019 27th ACM joint meeting on

european software engineering conference and symposium on the foundations of software engineering, pages 510–520, 2019.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. arXiv preprint arXiv:2403.07974, 2024.

Andreas Kamilaris and Francesc X Prenafeta-Boldú. Deep learning in agriculture: A survey. Computers and electronics in agriculture, 147:70–90, 2018.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In International Conference on Machine Learning, pages 18319–18345. PMLR, 2023.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. arXiv preprint arXiv:2404.00599, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. Science, 378(6624):1092–1097, 2022.

Yang Liu and Meng Zhang. Neural network methods for natural language processing, 2018.

Andrea Lops, Fedelucio Narducci, Azzurra Ragone, and Michelantonio Trizio. Agonetest: Automated creation and assessment of unit tests leveraging large language models. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, page 2440–2441, New York, NY, USA, 2024. Association for

Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695318. URL https://doi.org/10.1145/3691620.3695318.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. arXiv preprint arXiv:2104.08786, 2021.

Zhi-Cun Lyu, Xin-Ye Li, Zheng Xie, and Ming Li. Top pass: Improve code generation by pass@ k-maximized code ranking. arXiv preprint arXiv:2408.05715, 2024.

Zhicun Lyu, Xinye Li, Zheng Xie, and Ming Li. Top pass: improve code generation by pass@ k-maximized code ranking. Frontiers of Computer Science, 19(8):198341, 2025.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474, 2022.

Ayman Odeh, Nada Odeh, and Abdul Salam Mohammed. A comparative review of ai techniques for automated code generation in software development: advancements, challenges, and future directions. TEM Journal, 13(1):726, 2024.

Soya Park, April Yi Wang, Ban Kawas, Q Vera Liao, David Piorkowski, and Marina Danilevsky. Facilitating knowledge sharing from domain experts to data scientists for building nlp models. In Proceedings of the 26th International Conference on Intelligent User Interfaces, pages 585–596, 2021.

Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In Extended abstracts of the 2021 CHI conference on human factors in computing systems, pages 1–7, 2021.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. arXiv preprint arXiv:2401.08500, 2024.

Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. Physica D: Nonlinear Phenomena, 404:132306, 2020.

Jiho Shin, Moshi Wei, Junjie Wang, Lin Shi, and Song Wang. The good, the bad, and the missing: Neural code generation for machine learning tasks. ACM Transactions on Software Engineering and Methodology, 33(2):1–24, 2023.

Sundaravelpandian Singaravel, Johan Suykens, Hans Janssen, and Philipp Geyer. Explainable deep convolutional learning for intuitive model development by non–machine learning domain experts. Design Science, 6:e23, 2020. doi: 10.1017/dsj.2020.22.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. Bugs in large language models generated code. arXiv preprint arXiv:2403.08937, 2024.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. Journal of machine learning research, 9(11), 2008.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

Anthony J Viera, Joanne M Garrett, et al. Understanding interobserver agreement: the kappa statistic. Fam med, 37(5):360–363, 2005.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation. arXiv preprint arXiv:2406.04531, 2024.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859, 2021.

Yinghui Xia, Yuyan Chen, Tianyu Shi, Jun Wang, and Jinsong Yang. Aicodereval: Improving ai domain code generation of large language models. arXiv preprint arXiv:2406.04712, 2024.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, pages 1–10, 2022.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the evaluation of large language models in unit test generation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, page 1607–1619, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695529. URL https://doi.org/10.1145/3691620.3695529.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pages 1–12, 2024.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled

dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887, 2018.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: continual pre-training on sketches for library-oriented code generation. arXiv preprint arXiv:2206.06888, 2022.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. arXiv preprint arXiv:2406.15877, 2024.