Can Large Language Models accelerate the correction of student code ?

Auteur : Coco, Andreas
Promoteur(s) : Geurts, Pierre
Faculté : Faculté des Sciences appliquées
Diplôme : Master en science des données, à finalité spécialisée
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/24781

University of Liège
School of Engineering and Computer Science

Can Large Language Models accelerate the correction of student code?

Thesis supervisor
Prof. Pierre Geurts

Jury members
Prof. Gilles Louppe
Prof. Benoît Donnet
Prof. Christophe Debruyne

Master's thesis completed by Andreas Coco
in order to obtain the degree of Master of Science in Data Science

Academic year 2024 – 2025

Abstract

This thesis assesses to what extent large language models (LLMs) can accelerate the correction of student code in an introductory C programming course. The motivation is practical. Autograders are helpful grading tools but they miss many dimensions of code quality like clarity, efficiency and style. Hence, human review remains heavy and slow. LLMs, which can read code in context and provide natural language feedback, may fill part of this gap. The principal objective is to determine how they can be leveraged to accelerate code correction.

We conduct three sets of experiments on real coursework from the "Additional Information Theory" course at the University of Liège. First, we run preliminary code-generation tests to determine whether state-of-the-art LLMs can solve the course tasks. Second, we evaluate automated grading with Qwen2.5-Coder-7B on two datasets. These sets respectively consist of student submissions for a homework assignment and a project. We compare model-predicted grades and feedback to human grades. Third, we study error detection and code correction on the same homework by fine-tuning Qwen2.5-Coder-7B with LoRA using prompt-response pairs.

With respect to grading, the model's numeric predictions are not reliable. On both tasks, the mean errors often match or exceed those obtained by a constant baseline. However, when the task is reframed as a simpler classification problem where we ask the LLM whether each submission is fully correct, Qwen's performance is above chance. The best setting is the one in which we use a criteria-based prompt in French. This method consistently outperforms the baseline. Nevertheless, it remains insufficient for autonomous grading.

In error detection and correction, our initial fine-tuning with Qwen-generated data slightly improved correction rates. However, it often produced full code rewrites rather than genuine code corrections. A second fine-tuning attempt used more diverse, high-quality training data generated by OpenAI models, which encouraged targeted edits. However, this reduced correction performance on student submissions. These results indicate that improving a model's error detection and repair abilities is difficult with such limited datasets.

Overall, we find that LLMs are not powerful enough yet to replace human graders for either grading or error detection and correction. Their most promising use today is as a support tool alongside autograders and human review. Still, our findings are bounded by scope as we only used tasks in C from a specific course and minimal prompting. We recommend exploring more powerful models and considering fine-tuning on Python tasks with a larger, more comprehensive training set.

Résumé

Ce mémoire évalue dans quelle mesure les grands modèles de langage (LLMs) peuvent contribuer à accélérer la correction du code d'étudiants dans un cours d'introduction à la programmation en C. La problématique est d'ordre pratique. Les autograders constituent des outils précieux pour l'évaluation. Cependant, ils ne prennent pas en compte certains aspects essentiels de la qualité du code comme la clarté, l'efficacité ou le style. La relecture humaine reste donc indispensable mais elle est chronophage. Les LLMs, capables d'analyser du code dans son contexte et de donner un retour en langage naturel, pourraient partiellement combler cette lacune. L'objectif principal de ce travail est donc d'identifier comment ces modèles peuvent être mis à profit pour accélérer la correction.

Nous avons mené trois séries d'expériences sur des travaux réels issus du cours "Complément d'Informatique" de l'Université de Liège. Nous avons d'abord effectué des tests préliminaires de génération de code afin d'évaluer la capacité de LLMs à l'état de l'art à résoudre les tâches du cours. La seconde porte sur l'évaluation automatisée à l'aide du modèle Qwen2.5-Coder-7B. Elle utilise deux jeux de données composés respectivement de codes d'étudiants pour un devoir et pour un projet. Les notes et commentaires produits par le modèle ont été comparés à ceux attribués par des correcteurs humains. Enfin, la troisième évalue les capacités de détection et de correction d'erreurs sur le même devoir. Elle consiste à ré-entraîner Qwen2.5-Coder-7B avec LoRA à partir de paires "prompt-réponse".

En matière de notation, les prédictions numériques du modèle se révèlent peu fiables. Pour les deux tâches, l'erreur moyenne est souvent équivalente, voire supérieure, à celle obtenue avec un simple modèle de référence aux prédictions constantes. Toutefois, lorsque la tâche est reformulée en un exercice de classification binaire, consistant à déterminer si une soumission est entièrement correcte, Qwen obtient des résultats supérieurs au hasard. Les meilleures performances sont obtenues lorsque nous utilisons un prompt en français, avec des critères d'évaluation. Cette méthode surpasse quasiment toujours le modèle de référence. Les résultats restent cependant insuffisants pour envisager une correction totalement autonome.

Concernant la détection et la correction d'erreurs, un premier ré-entraînement avec des données générées par Qwen a permis une légère amélioration des taux de correction. Néanmoins, le modèle montrait une tendance à ré-écrire le code de façon extensive plutôt qu'à réellement le corriger de façon ciblée. Un second ajustement du modèle, cette fois reposant sur des données plus variées et de meilleure qualité produites par des modèles d'OpenAI, a incité le modèle à apporter des modifications plus ciblées. En contrepartie, il a diminué les performances sur les codes d'étudiants. Ces résultats indiquent qu'il est difficile d'améliorer les capacités de détection d'erreur et de correction de code d'un modèle avec des jeux de données aussi limités.

Dans l'ensemble, les résultats indiquent que les LLMs ne sont pas encore assez performants pour remplacer un correcteur humain. Cela vaut aussi bien pour la notation que pour la détection d'erreur. Leur usage le plus prometteur réside aujourd'hui dans un rôle d'assistant, en complément des autograders et de la relecture humaine. Les conclusions de ce travail restent toutefois limitées par sa portée. Nos expériences se concentrent uniquement sur des tâches en C issues d'un cours précis, avec une utilisation minimale des prompts. Nous recommandons l'expérimentation de modèles plus puissants et un ré-entraînement sur des tâches en Python avec un jeu de données plus grand et diversifié.

Acknowledgments

I am deeply grateful to Prof. Geurts, my thesis supervisor, for his guidance, availability and insightful advice throughout this work.

I would also like to thank my friend Guillaume for his invaluable suggestions on the writing of this thesis and for his constant help since my arrival at University of Liège.

My thanks go to my brother, Lucas, who has always been a role model to me and who offered thoughtful feedback whenever I needed it. I am equally grateful to my sister, Thelma, for her steady emotional support and for helping me put stressful situations into perspective.

Finally, I thank my parents for their unwavering support and for teaching me the value of hard work. I am especially grateful for their encouragement to pursue a second master's degree in a field I am passionate about, namely the field of data science.

Statement of Originality and Use of Generative AI

This thesis is the result of my own research and original ideas. Still, I acknowledge using OpenAI's ChatGPT solely as a writing aid to improve clarity, coherence and grammar. All suggestions produced by the tool were carefully inspected, verified and, when necessary, modified or rewritten to ensure accuracy and alignment with my intent. The final text, including its structure, arguments and conclusions, is entirely my work and I take full responsibility for its content.

Contents

Chapter 1

Introduction and Structure

1.1 Introduction

Large Language Models (LLMs) have become a central focus in artificial intelligence research and have experienced major advancements in recent years [1, 2]. Breakthrough models such as OpenAI's GPT-3 and GPT-4, introduced in 2020 and 2023 respectively, demonstrated unprecedented language understanding and generation capabilities [3, 4]. As a result, LLMs are now widely used across diverse domains, from healthcare and finance to creative writing [2]. They have also gained particular importance in software development and education [5–7]. Researchers and practitioners have been exploring LLM-driven applications that can assist in coding tasks as well as in teaching and learning contexts [5, 7–9].

In the programming domain, LLMs are increasingly employed for code generation, debugging and automated correction [2, 10, 11]. General-purpose models already learned basic coding skills from training data that included source code [5]. Still, models specifically trained or fine-tuned on code now often perform better [5, 12, 13]. These models allow users, even with little programming experience, to create executable code by describing a task in natural language [5, 12, 13]. Since the release of GitHub Copilot [14] in 2021, many LLM-based coding assistants have appeared, changing how developers approach programming tasks [15, 16]. Examples include proprietary tools like OpenAI's Codex [5], which powers GitHub Copilot, and open-source models such as Meta's CodeLlama [17]. These tools can not only generate code but also detect and fix errors, helping to streamline development and debugging. Their effectiveness has been shown in practice as tasks that once took considerable time can now be completed in seconds or minutes with LLM support [16].

LLMs have also begun to play an influential role in education, especially in computer science and programming [7–9]. In educational settings, these models have been used to augment both teaching and assessment. For instance, LLMs have been applied to generate study materials such as quizzes and summaries [8]. They can also act as on-demand tutors that explain concepts or code snippets and provide practice questions [8, 9].

The practical motivation for this thesis's research arises from the significant time and effort educators devote to correcting student code. In an introductory programming course like the

“Complément d’informatique” (“Additional Information Theory” in English) course taught by Prof. Geurts at the University of Liège, instructors must review a large number of code submissions. They also need to provide feedback on each of them. This process can be labor-intensive and time-consuming.

Existing automated grading tools like autograders help accelerating the grading of programming exercises by automatically checking outputs for correctness. However, these tools have clear limitations. They primarily evaluate whether the code produces the expected results. They cannot judge the clarity of a student’s solution, the efficiency of their logic or their adherence to good coding practices. Complex bugs or stylistic issues that do not affect the given test cases results may remain unaddressed by simple automated checks. Consequently, instructors still need to manually inspect code for logical correctness and code quality, which does not scale well with large classes.

In this context, modern LLMs present a promising opportunity to accelerate the code correction process. Leveraging their extensive knowledge, they could help instructors by analyzing student code, finding mistakes or inefficiencies and suggesting improvements. Such an approach could substantially reduce the grading workload and provide quicker feedback to students. Unlike traditional tools such as autograders and unit tests, LLMs are not limited to fixed test cases. They can understand code in context, comment on its logic and style and explain potential issues in natural language. This allows them to detect logical errors or poor style that might pass all tests, providing a more complete assessment of code quality.

Building on these insights, we aim to contribute to more efficient and effective code correction methods, ultimately improving the learning experience in programming education. This gives rise to the central research question guiding this thesis. *Can LLMs accelerate the code correction process in an introductory programming course?*

1.2 Thesis Structure

The thesis starts with a literature review (Chapter 2) that introduces LLMs and their applications in programming, code correction and education. It also covers the basics of prompt engineering and fine-tuning. Afterwards, we describe the data used in our experiments in Chapter 3. Specifically, we introduce the “Additional Information Technology” course and its coding tasks. Subsequently, Chapter 4 reports the results of preliminary code generation experiments that assess how well state-of-the-art models can solve the coding tasks used throughout this thesis. Chapter 5 then compares candidate models for our code grading and error detection experiments and motivates our choice of Qwen2.5-Coder-7B. The next chapter (Chapter 6) details our code grading experiments and their results. Following this, we present our error detection and fine-tuning experiments and their findings (Chapter 7). Lastly, we conclude the thesis, critically assess our work and make suggestions for further research. The source code for all experiments is available on GitLab.[1]

[1] https://gitlab.uliege.be/Andreas.Coco/master-thesis

Chapter 2

Literature Review

2.1 Large Language Models (SOTA)

Large Language Models (LLMs) are large neural networks trained on massive text corpora to understand and generate human-like language [2]. Most LLMs today are based on the transformer [18] architecture but other architecture types also exist. The transformer, introduced in 2017, is a type of neural network that has become the foundation for most large language models today [2]. It breaks text into tokens (small units like words or subwords) and converts each token into a vector of numbers called embedding [18]. The model then processes an input sequence of embeddings and generates output tokens one at a time in an autoregressive fashion [18]. Each predicted token is appended to the sequence and fed back in to inform the next prediction. At each step, the model employs multi-head self-attention to compute scores that indicate how strongly every previous token should influence the current position [18]. This mechanism contextualizes each embedding with information from the entire sequence [18]. By stacking multiple layers, each consisting of self-attention sublayers followed by simple feed-forward networks, transformers learn to capture complex, long-range patterns in language and generate fluent, human-like text [2]. In recent years, LLMs have experienced rapid advancements, through increasing model sizes and enhanced training techniques [1].

2.1.1 Scaling Laws and Model Optimization

Early research into scaling laws revealed that increasing the number of model parameters and quantity of training data leads to predictable boosts in performance [19]. This provided developers with a straightforward method to improve LLMs' abilities. For instance, OpenAI's GPT-3 model comprised 175 billion parameters [3] while the more recent version GPT-4 is estimated to contain approximately 1.8 trillion parameters [1].

However, subsequent studies have refined these insights. DeepMind's Chinchilla study found that, for a fixed computational budget, there exists an optimal balance between model size and the amount of training data, which is around 20 tokens per parameter [20, 21]. This finding shifted the focus from merely increasing model size to also considering the volume of training data. Consequently, smaller models trained on more data have outperformed larger, under-trained counterparts [21]. Building on this, some projects have experimented with over-training beyond the 20 to 1 token-to-parameter ratio. For instance, LLaMA 3 reportedly utilized a 200 to 1 ratio to further enhance accuracy [20].

More recently, new approaches have emerged that optimize computation during inference. OpenAI's recent o1 model, along with o3 and o4-mini released after it, exemplify models which perform well thanks to their longer reasoning processes rather than through an increased number of parameters [12, 22]. In the context of LLMs, reasoning is the model's capacity to generate and follow multi-step internal thought processes (chain-of-thought) to draw logical conclusions [23]. Overall, the state-of-the-art has evolved from an era of scaling at all costs to a more nuanced strategy that optimizes model size, training data and inference efficiency [20].

2.1.2 Recent Model Developments

The proliferation of LLMs continued in 2024 and 2025, with both industry leaders and open-source communities introducing new models. OpenAI launched GPT-4o in May 2024, featuring real-time audio-visual capabilities [24], followed by GPT-4.5 in February 2025 and GPT-4.1 alongside its mini and nano versions in May 2025 [25]. The latter models extended the context window to one million tokens while reducing serving costs [26]. The context window is the maximum number of tokens an LLM can process and keep in its "working memory" at one time [27].

Anthropic's Claude 3 series [28], released in March 2024, evolved into Claude 4 Opus and Sonnet by May 2025 [29]. These models can autonomously sustain coding tasks for several hours while preserving context and coherence and offer hybrid reasoning, switching from near-instant replies to extended deep thinking for more complex problems. Google DeepMind moved from Gemini 1.5 Pro in February 2024, offering a standard 128,000-token context and an experimental 1 million-token context [30], to Gemini 2.5 Pro in March 2025 [31]. The latter introduced a "Deep Think" mode and standardized 1 million-token prompts, making it highly ranked in many preference leaderboards.

Meta expanded its open-source offerings from LLaMA 3 in April 2024 (with 8B and 70B versions) [32] to the multimodal LLaMA 4 Scout and Maverick in April 2025 [2]. These models were trained on sequences of up to 256,000 tokens. Mistral complemented its Mixtral 8×22B model (64,000 tokens window) [33] and Mistral Large 2 respectively introduced in April and July 2024 (128,000 tokens)[34] with the vision-enabled Mistral Medium 3 in May 2025, also supporting 128,000-token contexts [35].

Lastly, xAI advanced its Grok series from version 1.5 in March 2024 (128,000 tokens) [36] and the multimodal 1.5V in April 2024 [37] to Grok 3 Beta in February 2025, featuring a 1 million-token context window [38]. Collectively, the 2024–2025 state-of-the art LLMs have set new standards with million-token contexts and native multimodality. Furthermore, they demonstrate that open-weight releases like LLaMA 4 and Mistral Medium 3 are not far behind proprietary flagship models.

2.1.3 Alignment Techniques: RLHF and RLAIF

In addition to new model releases, several key research trends aimed at improving LLMs emerged in 2024–2025 [39, 40]. Notably, aligning LLMs with human intentions and values has become increasingly important as even the largest models can produce irrelevant or harmful outputs if not properly aligned [41, 42].

The predominant method for alignment is Reinforcement Learning from Human Feedback (RLHF) [43]. It has been popularized by models like InstructGPT and ChatGPT [44]. In this approach, human annotators rank model outputs and a reward model is trained to fine-tune the LLM accordingly [44]. This technique has resulted in models that adhere to instructions more reliably and safely than their pre-trained versions [44].

Nevertheless, as RLHF is labor-intensive, research also explored Reinforcement Learning from AI Feedback (RLAIF) [45]. It employs AI-generated critiques to supplement or replace human evaluations. For example, Anthropic's "Constitutional AI" utilized an AI model to assess outputs against a set of predefined principles, reducing the need for human labels [46, 47].

Open-source initiatives have also applied novel alignment strategies. Among others, DeepSeek-R1 underwent large-scale reinforcement learning to enhance reasoning capabilities [13]. Its initial version, DeepSeek-R1-Zero, was trained via reinforcement learning without any supervised fine-tuning. This resulted in the emergence of powerful reasoning behaviors even though some issues like mixed-language outputs were still observed. To address these, the team developed DeepSeek-R1 using a multi-stage training process that included a supervised cold-start phase before reinforcement learning. This approach yielded an aligned model whose performance on reasoning tasks was claimed to be comparable to OpenAI's o1–1217 model [13, 22].

These developments underscore the importance of both human- and AI-based alignment techniques in improving LLMs, ensuring models are not only larger but also more reliable and logically consistent.

2.1.4 Efficiency and Model Distillation

Another significant trend is model distillation and the focus on efficiency [48]. The substantial resource demands of state-of-the-art models have led to methods aimed at compressing or streamlining LLMs [49, 50]. One such method is knowledge distillation, where a large "teacher" model trains a smaller "student" model [48, 51].

For instance, DeepSeek-R1, a 671-billion-parameter model, has been distilled into six smaller, dense, checkpoints (1.5B, 7B, 8B, 14B, 32B, and 70B) based on the Qwen-2.5 and LLaMA-3 architectures [13]. The objective was to transfer the extensive reasoning capabilities of R1 into models of more manageable sizes, facilitating deployment.

Distillation is also widespread in the community. Meta's original LLaMA-7B and LlaMA-13B models, for example, respectively led to Alpaca [52] and Vicuna [53] through fine-tuning on conversational data generated by ChatGPT models. These distilled models maintain much of the performance of their larger counterparts while being sufficiently lightweight for academic or private use [52, 53].

Another efficiency technique is Mixture-of-Experts (MoE) modeling [54], as exemplified, once again, by DeepSeek's architecture [13]. The DeepSeek LLM activates only a subset (approximately 37B) of its 671B parameters per query. This significantly reduces the computation demands per task while leveraging a huge set of parameters.

In summary, the state-of-the-art has embraced strategies to enhance LLM efficiency. Through knowledge distillation or exploitation of sparsity via MoE, researchers can deploy advanced models with substantially reduced computational overhead compared to traditional scaling methods.

2.1.5 Advances in Reasoning Capabilities

There also have been significant breakthroughs in enhancing the reasoning abilities of LLMs, which stemmed from chain-of-thought (CoT) prompting [55]. Introduced in 2022, CoT prompting enables models to generate intermediate reasoning steps rather than directly providing an answer [23]. This led to substantial improvements in tasks such as arithmetic, logical reasoning, and code synthesis [56, 57].

By 2024–2025, CoT prompting has become a standard tool to induce reasoning in LLMs [55]. Many state-of-the-art models now exhibit enhanced multi-step reasoning capabilities [12, 13]. They are often combined with self-consistency methods, which consist in generating multiple reasoning paths and selecting the most coherent one [56] or tree-of-thought strategies [58].

DeepSeek-R1 was explicitly trained to "incentivize reasoning capability" through reinforcement learning[13]. After training, the model demonstrated spontaneous generation of more structured and in-depth reasoning chains. The developers also open-sourced both the model and its distilled variants, accelerating research into reasoning-focused LLMs.

Shortly before Deepseek-R1, OpenAI had released the o1 model [22], the first in a series designed to "spend more time thinking before they respond". Subsequent models such as Gemini 2.5 Pro [31] and Claude Sonnet 3.7 [59] followed this trend.

In essence, beyond raw power, the state-of-the-art LLMs are characterized by their ability to reason through problems step-by-step [55]. This is especially relevant to coding and debugging tasks, where explaining and justifying each step can lead to improved solutions [60]. This emphasis on reasoning is a direct response to previous LLMs' tendency to "guess" answers or hallucinate information [61]. Now, with CoT prompting and alignment, the best models approach problems in a human-like analytical manner [56]. This progress lays an important foundation for using LLMs in complex tasks like evaluating or correcting code, which require both understanding and reasoning [62].

It is important to acknowledge that the state of the art LLMs are continually evolving. By mid-2025, models like OpenAI's GPT-4.5 [63] have been released. However, my experiments began in late 2024 and ran through early 2025. Hence, they incorporate the most recent releases only to a limited extent. I therefore review them for context but limit direct evaluation to models available up to early 2025. This rapid model turnover also means findings in literature can become quickly outdated. Overall, LLMs have made astounding progress in a short time and the remainder of this section outlines how these advances specifically relate to coding applications relevant to this thesis.

2.2 Large Language Models for Code

2.2.1 Code-specialized and General LLMs

Across time, LLMs have developed code-related abilities, even without explicit coding training [5]. This phenomenon arose because of the presence of code among the huge quantities of text used to train the models [5]. Early general-purpose models such as GPT-3 already demonstrated basic code generation skills [5]. As architectures and training methods improved (see Section 2.1), the models' coding capacities continued to strengthen [12, 13].

However, it became clear early that specialized code-focused LLMs could outperform general-purpose models on programming tasks. One of the pioneering examples of this trend is OpenAI's Codex [5]. By fine-tuning GPT-3 on a large corpus of GitHub code, Codex achieved a 28.8% pass rate on the Python HumanEval benchmark, whereas the base GPT-3 model's rate was close to 0% [5]. This gap shows that targeted training on source code and programming problems can yield substantially better code generation. Following Codex, numerous code-focused LLMs have appeared, mostly open-source ones like CodeStral [64], Qwen2.5-Coder [65] or CodeLlama [17]. These models often incorporate adaptations tailored for programming, including tokenization techniques to respect indentation and syntax. They also generally use training objectives like fill-in-the-middle to enhance completion capabilities. Such specializations improve syntactic accuracy and make models less prone to producing nonsensical outputs [64, 65].

Thanks to these methods, code-specific models of a given size tend to excel at programming challenges, often matching or surpassing larger general models on code benchmarks [64, 65]. Still, this specialization, comes at the cost of broader versatility [66, 67]. Code-specialized models may underperform on open-ended natural language tasks or common-sense reasoning outside of coding contexts [67]. Additionally, specialized models may lack the extensive real-world knowledge base of general LLMs as their training typically prioritizes code over web text [68]. In practice, state-of-the-art proprietary models still perform better in complex coding tasks, probably thanks to their huge scale and training dataset size [69, 70]. Nonetheless, the rapid advancement of code-focused LLMs has significantly narrowed this gap [65, 71].

In summary, the community has found that while general LLMs can acquire code-related skills as a side effect of their training procedure, model specialization can yield improved coding performance despite moderate sacrifices in generality.

2.2.2 Code Generation Benchmarks

As LLMs' coding capabilities have expanded, they have become increasingly popular for programming tasks [11]. Consequently, coding benchmarks are now almost automatically included in the evaluation when a model is introduced [70, 72]. Evaluating code generation quality requires benchmarks that verify functional correctness, not just the text's fluency [5].

The most widespread benchmark is HumanEval, introduced by OpenAI alongside Codex [5, 10]. HumanEval consists of 164 hand-crafted Python programming problems. The problems all encapsulate a function signature, a natural language prompt (docstring) and a set of unit tests. All solutions and test cases were written from scratch rather than taken from existing competition problems. The objective was to mitigate training data leakage and ensure reliable functional correctness evaluation. The models' peformance on this benchmark is usually evaluated using the pass@1 metric [5, 10]. Pass@1 measures the probability that the model's first generated solution passes all unit tests for a given HumanEval problem, indicating the fraction of problems solved on the first try [5].

Another major benchmark is MBPP (Mostly Basic Programming Problems), which was released by Google [73]. MBPP contains 974 crowd-sourced Python tasks with prompt descriptions and example input-output pairs. MBPP tests an LLM's ability to generate simple programs. It was designed such that even entry-level programmers could solve these tasks, making it an attainable target for models. Together, HumanEval and MBPP have served as the primary benchmarks in nearly all code-generation model papers since 2021 [70, 72, 74]. However, both are limited to Python, revealing a common problem in the community: many code datasets and benchmarks focus on Python [75]. This arises because Python is omnipresent online, making it easier for models to generate compared to other languages [76].

To address this language bias and better assess global coding skills, researchers have developed multilingual and diverse coding benchmarks. MultiPL-E was released in 2022 [77]. It extends HumanEval and MBPP to 18 programming languages by automatically converting the Python problems into other languages like, Java, C/C++ or JavaScript. This allowed for extensive multilingual testing even though the automatic translation of prompts and tests can introduce imperfections.

A more rigorous approach to multilingual evaluation came with HumanEval-X [78]. Rather than translating the problems, HumanEval-X builds on HumanEval by providing human-written reference solutions in four additional languages (C++, Java, JavaScript, Go) for each problem. This ensures that solving the problems in those languages is truly feasible and not merely due to translation [78]. These multilingual evaluations have revealed that many models struggle outside Python [77, 78]. A model might score highly on Python tasks but fail on C or Java versions of the same problems, indicating specialization to Python in training [77, 78]. This is a pertinent consideration for this thesis, since the LLMs are used here in the context of introductory courses in C. Hence, LLMs that are not only proficient in Python must be prioritized.

2.2.3 LLMs for Code SOTA (Late 2024)

At the start of this thesis' experiments in late 2024, the landscape of code-capable LLMs included both massive proprietary and impressive open-source models. Among proprietary models, OpenAI's GPT-4o, introduced in May 2024, had quickly become renowned for its remarkable coding capabilities [69, 70]. It achieved approximately 90% pass@1 accuracy on the HumanEval benchmark [69, 70]. Anthropic's Claude series also advanced rapidly, with Claude 3.5 Sonnet being released in June 2024 [70]. This model reportedly surpassed GPT-4o in some coding and reasoning benchmarks, particularly after the model was updated in October 2024, achieving a 93.7% pass@1 score on HumanEval [79]. Additionally, Google's Gemini 1.5 Pro, a multimodal model released in May 2024, was also a key player [30]. However, it did not reach the same performance standards as Claude 3.5 and GPT-4o, exhibiting a HumanEval score of 84.1% [80].

Despite their superior performance, using these proprietary models in an automated student feedback pipeline presents some practical drawbacks. They require API access with associated costs and rate limits. Open-source models can be deployed on the university servers, ensuring data stays within the institution's control and incurring no per-query costs. Moreover, open models can be inspected and fine-tuned, allowing customization. For instance, a model could be better aligned with an assignment's grading criteria. Given our university context and limited budget, these factors motivate using open models despite potential performance gaps.

Among open-source offerings, Llama 3.1, released in July 2024, stood out [72]. Its 405B parameter version approached proprietary-level performance with an 89.0 pass@1 score on HumanEval [72]. The LLM's smaller variants also performed well given their sizes. The 70B and 8B versions achieved 80.5% and 72.6% on the same benchmark, respectively. Notably, Llama 3.1 surpassed Meta's CodeLlama model which, despite being tailored for code, was released earlier, in January 2024 [17]. CodeLlama only managed 67.8% on HumanEval [17].

Other notable open-source contenders included Codestral [64], Mistral's code-specialized model, whose 22B version, released in May 2024, scored 81.1% on HumanEval, surpassing the larger Llama 3.1 (70B) [72]. Similarly, Mistral introduced its Large 2 general-purpose LLM (123B parameters) in July 2024, reportedly nearly matching GPT-4o's coding performance with a HumanEval score close to 90% [74]. Additionally, Mistral released smaller Ministral general models (3B and 8B) in October 2024, both outperforming Llama variants of comparable size [81]. The Ministral 3B and Llama 3.2 3B models respectively obtained 77.4 and 61.0 HumanEval pass@1 scores and the Ministral 8B and Llama 3.1 8B models respectively obtained 76.8 and 72.6 [81]. It is worth noting that the smaller Ministral model outperforms its larger counterpart in coding even though it is inferior to it in other aspects like mathematical reasoning or function calling.

In the code-focused landscape, two families of open-source models emerged as top performers by late 2024, namely the Qwen [65] and DeepSeek [71] models. Alibaba introduced Qwen-2.5-Coder, a series specialized in coding, available initially in 32B, 7B, and 1.5B versions, followed by additional releases of 14B, 3B, and 0.5B scales [82]. Despite their relatively small sizes, these models demonstrated exceptional performance. The Qwen2.5-Coder-32B Instruct version notably reached 92.7 pass@1 on HumanEval while the smaller 7B variant impressively scored 88.4 on HumanEval. These results came from training on over 5.5 trillion tokens, predominantly code in various languages like Python, C/C++, Java or JavaScript, along with mathematical content and other text [65, 82]. This involved careful data mixing and cleaning to ensure high-quality code learning. The model also underwent instruction tuning on a curated dataset of coding prompts to make it user-friendly [65, 82].

DeepSeek models, developed by a collaborative group, leveraged an innovative Mixture-of-Experts (MoE) architecture to achieve remarkable efficiency [71, 83]. Their DeepSeek-Coder-V2 model, released in June 2024, is effectively an ensemble model with a total of 236B parameters divided into many "expert" subnetworks. At inference time, a routing mechanism selects only about 21B parameters to activate for each generated token. This architecture allows high-capacity performance while reducing the computational overhead compared to typical large dense models. DeepSeek-Coder-V2 notably reached 90.2% accuracy on HumanEval, coming close to several leading closed-source models like GPT-4o [69] and even outperforming some like Gemini 1.5 Pro [80]. Additionally, a compact version, DeepSeek-Coder-V2-Lite (16B), achieved 81.1% on HumanEval [71, 83]. These developments show that open-source efforts were rapidly closing the gap, with some specialized models even surpassing certain proprietary systems on coding tasks.

Given the resource constraints inherent to academic settings, particularly limited GPU availability, model size is another critical factor to account for in this thesis. Extremely large models with hundreds of billions of parameters cannot be deployed or fine-tuned within university infrastructure. Consequently, this thesis prioritizes smaller, open-source models like the Llama variants (Llama 3.1 8B or Llama 3.2 1B/3B) [72, 84], Codestral (22B) [64], Ministral (3B/8B) [81], Qwen-2.5-Coder (7B) [65] and DeepSeek-Coder-V2-Lite (16B) [71]. These smaller models may not match the leading proprietary models' coding proficiency. Still, recent results demonstrate their rapid improvement, making the performance trade-off increasingly negligible for our purposes.

It is important to note that this section focuses on HumanEval benchmark scores, which reflect Python programming ability. However, the thesis specifically aims at using these models to help grade and correct student submissions in introductory C programming courses. This reliance on HumanEval scores is justified due to inconsistent reporting across different models, as evaluations rarely use the same benchmarks for multilingual coding, making direct comparison difficult [65, 71, 79]. This section's main goal is to provide a high-level survey of the leading LLMs for coding tasks. More targeted analyses comparing these models' performance specifically for C-language code generation and error detection are provided in Chapter 5.

2.2.4 Emerging 2025 Models and Outlook

The models reviewed so far represented the state of the art in late 2024. Nonetheless, the progress in LLMs' coding skills did not slow down in 2025. New models that pushed the frontier further were released. As noted in Section 2.1, OpenAI introduced the o3 and o4-mini (and o4-mini-high) models [12]. These are optimized for complex reasoning and OpenAI claim they deliver leading performance on coding tasks. In the same period, DeepSeek released DeepSeek-R1 and Qwen launched QwQ-32B, both reporting strong coding results [13, 85]. These reasoning-oriented models illustrate the growing emphasis on reasoning capabilities to enhance performance in code-related tasks. However, comparing these new models' performance with earlier models is difficult. Evaluations have shifted to more difficult, varied and contamination-resistant benchmarks [86]. For example, suites based on Codeforces problems and LiveCodeBench are now commonly used, replacing simpler tests like HumanEval [12, 13, 85–87]. Notably, CodeElo [87] is a standardized benchmark that compiles six months of Codeforces contest problems and evaluates LLMs by submitting their generated solutions directly to the platform to compute human-comparable Elo ratings. The results show that OpenAI's o1-mini and Qwen's QwQ-32B-Preview achieved Elo ratings of 1578 (close to the 90th percentile) and 1261 (above the 60th percentile). Still, reputable models such as Claude 3.5 Sonnet and Mistral Large Instruct fall below the 25th percentile, struggling even on the easiest problems.

In the product domain, GitHub Copilot [14] is worth mentioning due to its popularity, as 41.2% of Stack Overflow's 2024 developer survey respondents have indicated working with it [15]. This is the second highest rate in the "AI Search and Developer tools" category behind ChatGPT. By 2025, Copilot has evolved well beyond its initial Codex-based form [14, 88]. It integrates advanced LLM backends, including offerings from OpenAI, Anthropic and Google for certain features [14]. It also provides capabilities beyond simple code completion, such as context-aware bug detection, automated test generation and natural language explanations of code. This evolution reflects a general trend in which coding assistants are expected not only to generate code but also to support understanding and debugging [89].

Overall, the literature gives a promising but complex view. LLMs can already significantly accelerate coding tasks, but their effectiveness depends on the model. Newer models like OpenAI o3 [12] or DeepSeek-R1 [13] further improve coding reliability and may handle more complex student queries. However, they were released as this study was being conducted, so they are only discussed briefly in this thesis. We focus on models up to early 2025, acknowledging that this may lead to weaker performance.

2.3 LLMs for Bug Detection and Code Correction

Code generation, which consists in writing new code from a set of instructions is not the only code-related task in which Large Language Models can be useful. More specifically, bug detection and code correction are more relevant to this thesis. LLMs have demonstrated impressive skill in code generation. However, translating this success to bug detection and correction is not trivial [6]. Generating a correct solution from scratch and diagnosing or fixing an error in a given piece of code are distinct challenges [6]. LLMs can often produce correct code given well-detailed instructions but still struggle to analyze a buggy program and pinpoint the problems [90]. Recent studies confirm that strong generation performance does not necessarily imply competence in debugging, though benchmark performance on the two tasks correlate positively [91]. This was notably evidenced by Majdoub et al. [91] who benchmark five open-source LLMs on more than 4,000 buggy Python, Java and C++ programs to assess zero-shot debugging performance. Tyen et al. [92] systematically evaluate LLMs' ability to locate logical mistakes across several reasoning benchmarks and measure their capacity to correct those mistakes when given the precise error locations via a backtracking setup. They also train a lightweight out-of-domain classifier to predict mistake positions and enable effective correction. They show that LLMs rarely identify errors in their own reasoning steps but if the location of a mistake is provided, they can correct it [92]. In other words, the main challenge is localizing the error. Once told what to fix, the model's code synthesis ability is often sufficient to repair it [92].

2.3.1 Benchmarks (for debugging)

The benchmarks introduced in Section 2.1 like HumanEval and MBPP evaluate the LLMs' code generation abilities [5, 73]. However, since bug correction is a distinct capability, researchers have designed benchmarks specifically for error detection and repair. CodeXGLUE, released in early 2021, was one of the first benchmarks for this task [93]. CodeXGLUE gathered separate datasets, retrieved from open sources, for evaluation across several coding tasks. In addition to code generation, these tasks include code completion, code summarization, clone detection and bug fixing. Although it does not provide a single aggregated metric, CodeXGLUE highlighted the multiple aspects of code intelligence, going further than the ability of writing a function from scratch. At the time CodeXGLUE was introduced, strong models achieved moderate success on these relatively simple benchmarks, indicating that LLMs could fix certain classes of bugs [93]. However, these tasks often involve isolated code pieces or synthetic errors that may not reflect real-world complexity. Hence, modern state-of-the-art models are generally not evaluated on CodeXGLUE due to its limited difficulty [12, 13, 94].

Another commonly used benchmark in this category is Aider [12, 65, 71, 95]. Aider's polyglot benchmark evaluates LLMs on editing real code exercises in multiple languages, gathering tasks from various sources. It requires models to edit provided code, ensuring changes integrate correctly rather than working on isolated snippets. It falls between simpler benchmarks like CodeXGLUE and larger-scale suites like SWE-Bench, introduced below. It uses realistic single-exercise contexts that still require significant code edits [95].

The more recent SWE-Bench suite introduced a framework of 2,294 real GitHub issues and their fixes [96]. It tasks LLMs with reading a codebase and making changes across multiple files to solve all issues. This represents a more realistic bug-fixing scenario and is significantly more complex than writing an isolated function. It evaluates long-context understanding, the ability to follow a bug report and make a set of related changes across a project. SWE-Bench results show LLMs fall short on these complex tasks. When the benchmark was released, the best model (Claude 2) solved only about 2% of issues [96]. More recent models achieve better results. For instance, Deepseek-Coder-V2 [71] gets 12.7% and Gemini 1.5 Pro [30] obtains 18.7% while reasoning models DeepSeek-R1 [13] and OpenAI o1 [97] come close to 50% with respective scores 49.2% and 48.9% [13, 71]. The newest models of OpenAI, o3 and o4-mini perform noticeably better with respective scores of 69.1% and 68.1% [12]. Still, these numbers show there is plenty of room for improvement.

Such results reduce the optimism resulting from particularly high benchmark scores, like those obtained on HumanEval. They reveal how far LLMs are from autonomously handling realistic software engineering tasks [98]. Nevertheless, these diverse benchmarks provide a more comprehensive overview of code capabilities and inform this thesis. Accelerating student code correction involves understanding student code, not just toy problems, and debugging and directly modifying code. This remains a challenging task for LLMs.

2.3.2 Improving LLM Debugging with Tools and Feedback

Given these challenges, recent research has explored how external feedback and interactive tools can enhance debugging with LLMs. Jiang et al. [99] propose a self-debugging approach in which the LLM executes the code it generated and analyzes the results to decide how to fix potential errors. In their framework, the model acts as a rubber-duck debugger. It runs the code, observes errors or test failures and explains the code's logic to itself to detect mistakes. This approach was found to significantly improve accuracy on several coding tasks. For instance, CodeLlama-7B increased its pass@1 on the combined MBPP & HumanEval benchmarks from about 42.8% to over 50% after supervised fine-tuning [99]. It even reached around 53% when the model was further trained to explain its own code and then refine it based on failures. The main conclusion is that external feedback signals like runtime errors or assertion failures are helpful for correction. LLMs on their own have difficulty recognizing that their output is wrong. However, being provided a failed test or exception helps them identifying the problematic part of the code and fix it. Jiang et al. [99] also found that even without explicit test feedback, asking the model to explain its code step-by-step leads to small improvements. This suggests that encouraging the LLM to reason about the code's behavior can make it notice contradictions or mistakes. Still, the need for test outcomes or error messages is evident. Self-reflection by itself is rarely sufficient to catch logical bugs in code. This supports Tyen et al. [92]'s finding that the models can correct known errors but struggle finding them without help.

Another line of work allow LLMs to use traditional debugging tools in their debugging process. Yuan et al. [90] introduced debug-gym, a text-based environment that allows an LLM agent to perform interactive debugging actions. It allows the LLM to use the Python debugger, set breakpoints, inspect variables, read multiple files and in an isolated code environment [90]. The idea behind this technology is to mimic how human programmers debug. That is, start by forming a hypothesis, run the program or go through it step by step, collect information and then refine the code. By allowing the model to do more than one-shot code edits, debug-gym agents can engage in iterative trial-and-error, while being guided by the program's actual behaviour [90].

Their results are somewhat promising but also reveal limitations. For instance, with a powerful backbone model (Claude 3.7 Sonnet), an agent using the debugger tools solved about half of the issues in SWE-Bench-Lite, a 300-problem subset of SWE-Bench [90]. By contrast, the same model without tool use solved only around 37%. This suggests that giving LLMs a way to actively gather information beyond what's in the prompt can significantly boost their bug-fixing performance. Nonetheless, smaller-scale models, like the 70B version of Llama 3.3, showed only small improvement as they often failed to use the debugger effectively. Yuan et al. [90] argue that these results could probably be improved as the LLMs used in their experiments have not been trained or fine-tuned on debugging trajectories. Thus, only the largest models exhibit some emergent ability to leverage the tools provided by the environment. Moreover, even the agent with the best backbone still could not solve half of the SWE-Bench-Lite problems. Also, the success rate on the full SWE-Bench remains particularly low. Hence, even though debug-gym brings a promising avenue to improve LLMs' code-fixing abilities, it still exhibits significant limitations. Another drawback is that the technology is very Python-focused [90]. Notably, the only debugger it gives access to is the Python debugger although the environment is made to be easily extended [90].

2.3.3 Critical Outlook

In summary, LLMs can already accelerate the code correction process, especially when errors are known or easily detected [92]. Still, they fall short of reliably finding bugs in more complex settings [13, 71]. Benchmarks like CodeXGLUE showed LLMs success on simple bug fixing tasks [93]. More rigorous evaluations like SWE-Bench reveal that models still struggle to solve non-trivial bugs in practice [96]. The literature indicates that LLMs are good at correcting errors once they know what and where to fix but identifying those errors remains hard. Using feedback from running code, such as test failures, debug tools, or user hints was found to greatly help. As Tyen et al. [92] and Jiang et al. [99] both suggest, an LLM on its own des not notice its mistakes by itself but external signals can help it to find a right fix. Research like debug-gym [90] represents a promising avenue for building more autonomous debuggers, allowing models to replicate the human debugging approach.

2.4 LLMs in Education

Improvements in large language models and their growing popularity quickly let them to being used for multiple purposes in education. These models can generate educational materials such as quizzes and summaries and serve as virtual tutors by offering explanations and interactive learning opportunities [8, 9]. For instance, Kasneci et al. [8] highlight that LLMs can generate quiz questions and even explain code snippets to help students learn. This reduces the teachers' workload and customizes learning for each student.

In addition to generating study material and tutoring, LLMs are being explored for assessment and feedback [7]. The literature suggests they can analyze student responses or code, provide feedback and accelerate grading [100]. However, challenges such as hallucinations, alignment with grading criteria and fairness make human oversight and AI understanding essential [8, 9].

2.4.1 Automated Grading and Feedback with LLMs

Techniques to automate the grading of written assignments and open-ended responses have been a subject of research for multiple years [100]. Traditional automated scoring methods used machine learning (ML) models trained based on grading criteria to evaluate essays and short answers [101]. Recent advances in LLMs have significantly expanded these capabilities. State-of-the-art LLMs can evaluate free-form text answers like a human grader would thanks to extensive training on human writing and their ability to understand context and semantics [102]. For example, Pan et al. [102] compare LLM scoring of short scientific explanations to a dedicated ML system called EvoGrader. They report that with minimal prompt tuning, GPT-4o, the best-peforming LLM on this task, achieved solid agreement with human-attributed scores despite making significantly more errors than the specialized model. GPT-4o obtained approximately 500 more incorrect answers than the machine learning model over a dataset comprising 9000 student answers. This suggests that LLM-based graders are reasonably fairly reliable but less precise than dedicated solutions in certain domains. Still, a major advantage of LLMs is that they do not require task-specific training data [102]. The traditional system needed a large human-graded dataset and manually designed features to reach its accuracy [102]. Thus, LLMs offer a grading tool that is faster to deploy though slightly less accurate.

Another benefit of LLMs is that they can provide qualitative feedback alongside the grades [103]. This capability can transform assessment into a learning opportunity rather than a simple evaluation. Recent work by Chiang et al. [104] deployed GPT-4 as an automatic evaluator for assignments in a large university course (more than 1,000 students). They revealed that students generally found LLM-based grading acceptable as long as they have free access to the LLM evaluators. The automated grader significantly shortened feedback return times. However, they observed some important limitations. The LLM sometimes failed to strictly follow the grading rubrics or instructions. This sometimes led to inconsistent or unfair evaluations. These findings show that while LLMs can greatly reduce grading time, their use in important assessment must be handled carefully. Issues of fairness, transparency and reliability remain [104].

Consequently, improving LLM's instruction-following for grading and developing safeguards or complementary checks to mitigate these issues is a priority for the future [104]. This field is evolving rapidly since as discussed in Section 2.1, research continues to refine LLMs and make them more reliable. Nevertheless, given their current capabilities, LLMs cannot function as independent graders yet. Human oversight remains essential to support critical thinking and ensure academic honesty [105]. In other words, a human must stay involved in the grading process [105].

2.4.2 LLM-Assisted Programming Assignment Correction and Grading

The automated assessment of programming tasks presents unique challenges compared to grading natural-language answers [100]. Recent studies have started to evaluate how well LLMs perform in code correction and grading scenarios. Lagakis et al. [105] conducted one of the first evaluations. They used GPT-3.5 and GPT-4 to grade introductory programming assignments in a Python course. They used a zero-shot approach, asking the models to grade student code and deliver feedback. The results were mixed. With well-designed prompts, the LLMs could assess simple programs reasonably well and make relevant comments. Nonetheless, they struggled to predict precise grades for more complex or rare tasks[105]. As expected, GPT-4 outperformed GPT-3.5. Furthermore, it performed even better when given minimal instructions like few-shot examples or grading criteria in the prompt. Still, even GPT-4 showed inconsistencies and errors when faced with complicated code or atypical student solutions [105]. The authors concluded that LLMs could be useful for automated code grading. However, they lack the reliability and consistency to be used by themselves [105]. Still, the models they conducted their experiments with are no longer representative of the state of the art [12, 24].

Traditionally, autograders for coding assignments rely on methods like unit tests, code inspections or static analysis and sometimes symbolic execution to evaluate correctness and quality [106]. These methods can catch many bugs but often require considerable manual setup like writing test cases or specifications [106]. LLMs exhibit some abilities that traditional test-based graders do not [106]. They can analyze the code semantically and explain errors or inefficiencies [107]. They can address aspects like coding style, readability or logical approach [108].

A growing body of work integrates LLMs directly into programming assignment pipelines to deliver detailed, personalised feedback [107, 108]. For example, Pankiewicz et al. [107] embedded GPT-3.5 in an automated assessment platform and found the model could highlight the exact compiler-error line and suggest a remedy. This helped students solve tasks more quickly. Azaiz et al. [108] integrated GPT-4 Turbo into an online evaluation system and found that the model correctly detected buggy lines in 73% of 165 feedback messages. The LLM explained the faults in plain English and gave a corrected version of the code. Such feedback helps students to understand and fix their mistakes.

However, there are some limitations. LLMs can make mistakes in their feedback or be misleading if they misinterpret the code [109]. Contrarily to compilers or test suites, LLMs might hallucinate a problem that does not exist or overlook subtle errors, especially if the error involves complex logic or domain-specific knowledge [110]. Moreover, ensuring that the LLM's feedback aligns with the instructor's expectations regarding aspects like coding style or allowed solving methods is difficult [111]. When using GPT-4 as teaching assistant in their "Introduction to Generative AI" course, Chiang et al. [104] observed that the model's strict view led to unfair penalization. Similar issues could arise in code grading.

In order to address these challenges, recent work has explored hybrid approaches that combine LLMs with traditional autograders [112]. For example, an LLM can be used to analyze a student's code when it fails certain autograder tests and produce a human-readable report [107]. A recent example is given by Nagakalyani et al. [113], which pair a fine-tuned version of Code Llama with their autograder. The model was fine-tuned using a dataset comprising programs and their associated grades. Given a problem statement, student code submissions and a grading rubric, TA Buddy, the tool they designed, gives rubric-based scores and explanations. They report this method cut their grading time by approximately 45% while matching human marks in almost 90% of cases. Another type of approach consists in using LLMs to automatically repair code, as discussed in Section 2.3. Given a buggy program, the LLM is tasked to write a corrected version [114]. For example, Zhao et al. [114] applied their Peer-aided Repairer (PaR) framework in an advanced systems-programming course. They combined GPT-4 with carefully selected former students correct solutions to generate patches for student submissions. PaR was able to repair about 45% of 1300 programs from their datasets. Nonetheless, these systems are not flawless. Some bugs require deep reasoning or domain context that LLMs may lack [115]. There is also a risk that the model introduces new errors [110].

In summary, LLMs can accelerate grading by detecting common mistakes, suggesting fixes and giving feedback [107],[108]. They are also promising for evaluating code quality aspects that traditional tests miss [106],[107]. However, their accuracy on complex tasks is limited and ensuring their judgments match with instructors' standards requires care [104]. Resultingly, human oversight remains essential to review AI outputs and avoid mistakes and misuse [105]. It is also worth noting that LLMs are still evolving rapidly and research on LLMs in educations is still in its early days. This means that as techniques and models advance, LLMs are likely to play a larger role.

2.5 Prompt Engineering and LLM Fine-tuning

Efficient use of large language models (LLMs) relies on two main approaches, namely prompt engineering to guide the model's outputs and fine-tuning to adapt it to a specific domain or task [116]. This section reviews key techniques in both areas, focusing on those that were proven to be effective for code generation and correction.

2.5.1 Prompt Engineering

Prompt engineering consists in designing the input given to a LLM in order to obtain the best response possible from the model [117]. It involves choosing the right words, examples and structure to guide the model towards the desired output [117]. Even without any fine-tuning, well-designed prompts can significantly improve performance on a task [118]. A major example is Chain-of-Thought (CoT) prompting [23], mentioned in Section 2.1. Instead of asking the model to simply give its answer, we prompt it to reason step by step. For instance, a CoT prompt to debug code could say "Analyze the code step by step to find any errors." By requesting reasoning, we push the model to explicitly state the logical steps it follows to reach a conclusion. This method both increases accuracy and makes the output easier to interpret [23].

An extension of CoT is Self-Consistency (SCoT) decoding [56]. This technique involves sampling multiple reasoning paths by letting the model generate different chains of thought using temperature or randomization. The next step is to take a vote or use a scoring heuristic to select the best answer. This approach reduces the probability of following an incorrect reasoning [56]. If most chains reach the same answer, it is probably correct [119]. In coding tasks, self-consistency can be applied by generating multiple potential corrections or explanations of a bug then choosing which one is most likely to be right or is returned the most frequently [120].

Few-shot prompting is another general technique. It simply means providing examples in the prompt [118]. For grading or feedback tasks, one could simply include some examples like providing erroneous student code, pointing out the error it contains and giving feedback example [121]. Few-shot learning often improves reliability by showing the model the format and level of detail expected [118]. However, context length limits can make including examples infeasible, depending on the length of the prompt and student code. Still, recent models typically support very long contexts, usually at least 32,000 tokens [65, 71]. Thus, including examples or longer instructions is generally feasible.

There also exist coding-specialized prompting methods [122]. One of them asks the model to plan first and implement second. In the literature, this is referred to as Plan-and-Solve or self-planning [123, 124]. The model is first asked to describe a solution or identify potential difficulties before writing the actual code and sometimes to run tests [124]. This step by step prompting can be done in a single interaction with the model or through a multi-turn conversation [124]. Breaking the task down in multiple phases can prevent the model from jumping straight to an answer without reflection [125].

2.5.2 Fine-tuning

While prompting can already be helpful, fine-tuning allows to directly specialize an LLM to a given task or dataset [126]. Fine-tuning the entire model is often impractical for large models due to computational cost and the risk of overfitting [127, 128]. Consequently, lightweight fine-tuning methods have become popular [127, 129].

The most widespread lightweight specialization technique is Low-Rank Adaptation (LoRA) [127]. LoRA consists in freezing the pretrained weights and inject small, trainable low-rank matrices into the model's layers. Thanks to this, the adaptation is learned in a low-dimensional subspace rather than updating all the model parameters. The fine-tuning remains highly effective since most task-specific adjustments lie in that subspace, allowing it to match or nearly match full fine-tuning performance [130]. Furthermore, since only these adapters are optimised, the number of trainable parameters strongly decreases [131]. For instance, fine-tuning GPT-3 (175 billion parameters) with LoRA using rank 8 requires updating roughly 38 million parameters, representing only around 0.02% of the model [127]. Setting the rank parameter to 1 reduces this further to around 4.7 million. This massive reduction allows a large model to be adapted to a new dataset using a relatively small amount of GPU memory [130].

QloRA makes LoRA even more memory efficient by combining 4-bit quantization of model weights with LoRA fine-tuning [129, 130]. Dettmers et al. [129] showed that even a 65B model can be fine-tuned on a single 48GB GPU with this approach. Moreover, the authors showed that the 4-bit-quantized LoRA fine-tuned model achieves task performance comparable to the full 16-bit fine-tuned version.

Another common branch of LLM fine-tuning is instruction tuning [132]. This refers to fine-tuning on a dataset of instruction-response pairs to make the model better at following instructions [132]. Put simply, by fine-tuning a weaker model on a broad set of instruction-response examples, we can give it conversational, instruction-following abilities similar to ChatGPT [52]. The Alpaca project applied this to LLaMA-7B using 50,000 synthetic instruction-following examples [52]. Instruction tuning is a common practice in the LLM field. For example, Qwen-2.5-Coder [65] and DeepSeek-Coder-V2 [71] underwent instruction tuning as part of their training, yielding their "Instruct" versions. A possible way to apply this to code evaluation would be to tune a model on prompt-answer pairs following specific grading criteria or a particular feedback format.

RLHF and RLAIF are more complex fine-tuning techniques. They were discussed in Section 2.1 due to their popularity in modern models [13, 47]. Both methods involve training a model using a learned reward signal to make the model favour certain outputs. The model generates multiple candidate responses for a given input. In RLHF, human annotators then review these candidates and select or rank the best outputs. In RLAIF, it is an off-the-shelf or stronger AI model that provides the feedback. These preferences are subsequently used to train a reward model. This model is in turn used for policy optimization via reinforcement learning. In an educational context, one might implement RLHF with teachers rating the feedback given by an LLM to a student. The model would then be optimized to maximize those ratings. However, this is likely beyond the scope of a master's thesis due to resource and time constraints. RLHF and RLAIF are more common in industry [13, 22].

It has been shown in the past that fine-tuning pretrained language models on code data can yield important advances in coding ability. For instance, OpenAI's Codex model was essentially GPT-3 fine-tuned on GitHub code [5]. Similarly, CodeLlama was LLaMA-2 fine-tuned on a large set of code files [17]. These resulted in big jumps in coding ability [5, 17]. Nevertheless, fine-tuning on a narrow distribution like a single course's assignments requires special attention to avoid overfitting [133]. Techniques like LoRA can help [134]. By keeping the base model intact and only injecting the adapters, the model retains general knowledge and only adjusts slightly to the new domain [134].

In conclusion, the literature indicates that carefully designing the prompts and fine-tuning significantly improve the models performance [23, 117, 127]. Fine-tuning challenges include building high quality datasets and avoiding overfitting [135].

Chapter 3

Data

3.1 Course Presentation

All coding tasks and student assignments in this thesis are drawn from the “Compléments d’informatique” course, which translates to “Additional Information Technology”. This course, taught in French, is the second programming class in the Civil Engineering bachelor’s program at the University of Liège. Students must first complete the “Introduction to Computer Science” course, which covers fundamental computer science concepts, basic algorithm design and analysis, an introduction to C and elementary data structures.

The “Additional Information Technology” course focuses on fundamental algorithmic concepts and common data structures. It also emphasizes teaching students to analyze programs and plan their implementation. It begins with a review of C, then covers algorithm construction and analysis, core algorithmic principles, program organization and data structures. The final chapter introduces Python and other programming languages. All coding tasks in the course are in C, except for the last chapter.

3.2 Course Content

In this course, students complete two types of graded tasks. In both, emphasis is placed on evaluating the performance and structure of the code written by the students. The first type, called “Travaux Pratiques” (TP), consists of four assignments. The second type comprises three annual projects that are more extensive and assess a wider range of skills and methods. Both TPs and projects test students’ mastery of the course material, each applying different concepts covered in class. The course has been taught for seven years. TP0 has remained unchanged for the last three years. All other assignments and projects are updated each year while evaluating the same core competencies.

TP0 is an introductory assignment designed to familiarize students with the tools used throughout the year, such as the submission platform. It requires implementing simple iterative functions. TP1 assesses algorithmic skills by having students implement divide-and-conquer sorting and search algorithms. TP2 evaluates program organization, while

TP3 focuses on working with data structures such as trees and dictionaries. TP4 tests students' ability to write code in Python. Similarly, the three projects cover complementary abilities. Project 1 involves simple algorithms and array manipulation. Project 2 emphasizes modular programming with separate, interacting modules. Lastly, Project 3 examines string manipulation and data structure use. All tasks must be completed individually, except for TP4 and Projects 2 and 3, which may be completed in pairs.

3.3 Tasks Used in Experiments

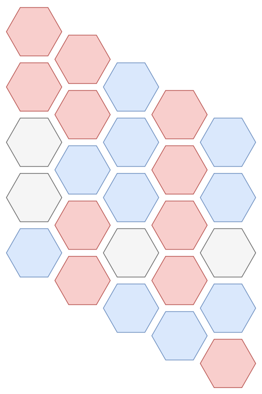
For my experiments, I mostly relied on the first homework and project of 2023–2024. In the TP1 from 2023, students work with an $n \times n$ integer array M. A peak in this array is a position (i, j) that satisfies the inequalities in Equation 3.1. All entries outside the array are defined as -∞ so that every matrix has at least one peak. Students must write a function, `int *findPeak2D(int n, int M[n][n])`, that returns a two-element array containing the elements (i, j) corresponding to any peak's coordinates. The function must be as efficient as possible in terms of time complexity. Additionally, the auxiliary function `int isPeak(int n, int M[n][n], int i, int j)` should return 1 if position (i, j) is a peak and 0 otherwise. Finally, students must comment the time complexity of `findPeak2D`, which must be strictly better than $O(n^2)$.

$$\begin{aligned} M[i][j] &\geq M[i+1][j], \\ M[i][j] &\geq M[i-1][j], \\ M[i][j] &\geq M[i][j+1], \\ M[i][j] &\geq M[i][j-1]. \end{aligned} \tag{3.1}$$

In Project 1 from 2023, students implement HexGame, a game based on an $N \times N$ grid of hexagons for two players, red and blue. Taking turns, each player selects an unclaimed hexagon and assigns a colour to it. The aim is to finish with the most islands. An island is a group of adjacent hexagons sharing the same colour. Two hexagons belong to the same island if they are the same colour and can be reached by moving only through neighbouring hexagons of that color. Figure 3.1 shows two examples of grids. The game ends when no unclaimed hexagons remain or when a single island connects either the left and right borders or the top and bottom borders of the grid. This project was strongly inspired by the "Islands of Hex" Nifty assignment [136, 137]. Nifty Assignments is a repository of exemplary computer-science exercises that educators freely share to enrich their teaching.

The file students have to submit is `hexgame.c`, in which students must implement the game. It defines the HexGame structure and all functions declared in `hexgame.h`. In total, there are eleven functions to define that enable gameplay. These functions notably include `int hexgameGetNumberOfIslands(HexGame *game, Player player)`, which returns a player's current number of islands, and `void hexgamePlayMove(HexGame *game, Move move)`, which updates the game state for a move unless it is invalid or the game has ended. Students must also comment the time complexity of their `hexgameGetNumberOfIslands` implementation in terms of N, the grid size.

Finally, the instructions guide students through implementation. They are told to begin by


(a) 4 red and 4 blue islands

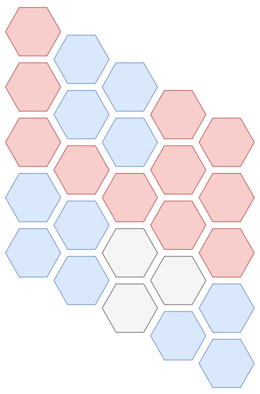

(b) 1 red and 3 blue islands

Figure 3.1. HexGame grid examples. An island is a group of adjacent hexagons sharing the same colour.

identifying the fields for the `HexGame_t` structure, then follow a recommended sequence for writing each function and apply the supplied tips. A full description of the project, containing the complete list of functions, is available in Section A.1 of the Appendix.

This thesis also uses tasks TP0, TP1, TP2 and TP4 from the 2022–2023 academic year, TP2 and TP3 from 2023–2024 and TP1 and Project 1 from 2024–2025.

TP0 (2022) requires implementing a binary sorting algorithm that orders an array of N zeros and ones in $O(N)$ time using only element swaps. TP1 from the same year asks for a function that computes the total length of the union of intervals stored in an $L \times 2$ array in $O(L \log(L))$ time. TP2 requires writing a generic `map` function which takes an array, its size and a function and returns a new array by applying that function to each element. It is then used to implement a `cartesian_to_polar` function that converts a list of Cartesian points to polar coordinates using the `map` function.

In TP2 (2023), students must implement a generic `mapImage` function that applies a given function to every pixel in one or more images. They must then use `mapImage` to create two additional functions. The first one is `encrypt`, which hides one image inside another. The second one is `decrypt`, which retrieves an image hidden in another. TP3 from 2023–2024 requires students to represent simple arithmetic expressions, built from the operators $+$, $-$, $*$, and $/$, numeric values and variables, as binary trees. They need to write three functions. The first one is `exprPrint`, which converts a tree into its string form. The second one is `exprEval`, which computes the value of a given expression. The last one is `exprDerivate`, which returns an expression's derivative with respect to a specified variable. In TP1 (2024–2025), students are tasked with coding a `tile` function that covers an $n \times n$ grid, where n is a power of 2. The function must use L-shaped tiles to fill the grid so that exactly one cell remains uncovered. In Project 1 of the same year, students must develop a set of functions for a word-search game. Given an $h \times w$ grid of letters and a dictionary, the goal is to find as many valid words as possible within the grid.

3.4 Autograder and Final Assessment

For all these graded tasks, students have access to an autograder on the Gradescope platform, which runs their code through automated tests. This process lets them iterate their solutions, identify and fix issues and improve their final submission within a limited number of attempts. A human grader then assigns the final grade using additional criteria that the autograder cannot measure. This ensures that tests are not passed by sheer luck and allows assessment of aspects beyond automated tests.

3.5 Confidentiality

It is essential to note that throughout all experiments, no student names or IDs were ever used or provided to the language models. In particular, this applied to models accessed online to preserve confidentiality. Instead, each submission was identified only by its unique Gradescope submission ID, ensuring that the author of any given piece of code could not be determined.

Chapter 4

Code Generation Experiments

The primary objective of this thesis is to evaluate the potential of Large Language Models (LLMs) to facilitate and accelerate the correction of student code submissions. However, we designed additional preliminary experiments to assess whether LLMs could be used to complete the programming tasks from the “Additional Information Technology” course.Students are strictly prohibited from using LLMs to generate solutions but we investigate this for experimental purposes. Previous research indicates that generating code and detecting errors within code and correct them represent distinct yet correlated skills [6]. Hence, before delving into code correction, experiments were conducted to examine the proficiency of LLMs in code generation. These experiments focused on the model’s ability to tackle introductory C programming course assignments and projects.

4.1 Experimental Design

Concretely, we evaluated large language models on all graded assignments from the 2022–2023 academic year (TP0 through TP4), except TP3 for which we used the 2023–2024 version. We selected every assignment from a single year because they span the course topics and exhibit various levels of difficulty. In addition, we included TP2 from 2023–2024 and TP1 from 2024–2025, as these exercises were judged to be particularly challenging. Finally, we added Project 1 from 2024–2025 to our tests. Unlike the assignments, this project requires writing multiple interdependent functions, making it more complex to complete. This allowed us to assess the model’s ability to manage a more involved task.

The selected model for these experiments was GPT-4o [24], developed by OpenAI. OpenAI’s models are the most widely used large language models and are particularly popular among code learners [15]. Thus, we assumed most first-year students would choose it for completing assignments and projects. When these experiments were conducted, GPT-4o was OpenAI’s most capable model and was available for free on their website. Still, the free plan imposed a usage limit that varied with demand but was reset every five hours [138]. Consequently, we expected that students seeking to generate code would select the strongest available model and, if necessary, wait for the limit to reset in order to continue using it.

```
Je dois implémenter une fonction en C
pour un exercice de programmation. Voici
l’énoncé:
[Enoncé]

Exemple d’entrée: [description de
l’exemple d’entrée]
Exemple de sortie: [description de
l’exemple de sortie]

Voici le contenu des fichiers que j’ai à
ma disposition :

Le fichier main.c contient:
‘‘‘[contenu du fichier main.c]‘‘‘

Le fichier [nom_de_fichier].h contient:
‘‘‘[contenu du fichier
[nom_de_fichier].h]‘‘‘

Le fichier [nom_de_fichier].c (à
compléter) contient:
‘‘‘[contenu du fichier .c avec la
signature de la fonction]‘‘‘

Je dois compléter la fonction
[nom_de_fonction] dans le fichier
[nom_de_fichier].c pour que le programme
fonctionne correctement. Peux-tu m’aider à
 implémenter cette fonction ? Il faut que
le code compile et respecte l’énoncé.
```

(a) Original prompt

```
I must implement a function in C for a
programming exercise. Here are the
instructions:
[Instructions]

Example input: [description of the
example input]
Example output: [description of the
example output]

Here is the content of the files I have
at my disposal:

The main.c file contains:
‘‘‘[content of the main.c file]‘‘‘

The [filename].h file contains:
‘‘‘[content of the [filename].h file]‘‘‘

The [filename].c file (to be completed)
contains:
‘‘‘[content of the .c file with the
function signature]‘‘‘

I must implement the function named
[function_name] in the [filename].c file
so that the program works correctly. Can
you help me implement this function? The
code must compile and respect the
instructions.
```

(b) Translated prompt

Figure 4.1. Code generation prompt template for the assignments. The prompt on the left is the original prompt, in French. The prompt on the right is its English translation. The prompt was designed to mimic a student’s approach. It is thus simple but complete, containing all information provided in the instructions.

4.2 Prompt Engineering

The prompts supplied to the LLM were designed to reflect a student’s natural approach. They were straightforward and did not employ advanced prompt-engineering methods despite including every detail a student might provide. Specifically, we provided the model with all the information from the assignment instructions. Figure 4.1 contains the template used for all tasks, which we customized according to each assignment’s specific instructions and available materials. The prompt is in French because the course is taught in French and we assume students are unlikely to translate the instructions. An English translation of the prompt is shown alongside the French version. Additionally, our prompt includes a single example of input paired with its output, making it a one-shot prompt. Although one-shot prompting typically yields lower performance than few-shot prompting, it generally outperforms zero-shot prompting [118].

Slight adjustments were made to the prompt for certain tasks, still with the intent of mirroring student behavior. For instance, TP0 and TP1 of 2022 are in fact the only tasks for which we included a proper input-output example because these were the only ones whose instructions contained one. We assumed students would not take the time to create their own examples. For TP1 of 2024, we created two prompt versions because the instruction file contains an input-output example but as an illustrative figure, not in textual form. Thus, the first version omits the example. The second one references it with "(see the image for an example)" and we provided the image alongside the prompt to GPT-4o. This is permitted because GPT-4o is a multimodal model [24]. This means its vision encoder turns uploaded images into tokens that are then integrated into its transformer pipeline [24, 139]. This allowed us to assess whether including an illustrated example affects the generated code. Finally, the 2022 TP4 prompt was slightly modified to use Python instead of C.

The project employed a custom prompt, developed in three versions. Multiple files interact in the project although all functions that need to be implemented lie in a single file. In version 1, the prompt was accompanied by a ZIP archive containing the PDF instruction file which also contained advice and visual support and all project files. GPT-4o could read this archive thanks to its Advanced Data Analysis tool [140]. This version offered the model a complete view of the materials at its disposal. In version 2, the prompt was self-contained. It omitted the PDF's illustrations and included only the code snippets necessary to implement the required functions. The templates for these prompts appear in English in Figure 4.2.

The third prompt version was created by asking GPT-4o itself to refine the no-ZIP prompt. We provided the original prompt and asked the model to produce the best possible version for an LLM. Although our main goal was to test the model's ability to complete course assignments and projects, we also explored whether a refined prompt would improve outcomes. GPT-4o returned a more structured prompt, shown in Figure 4.3 and recommended including the visual example from the PDF instructions in text form. While further prompt engineering was possible, we limited changes to keep the prompt practical for students and because code generation is not the main focus of this thesis.

4.3 Evaluation Procedure

Even though code generation is often measured by the pass@1 score, we applied a simpler approach. We ran tests manually on the ChatGPT website to avoid API fees, so estimating pass@1 would have been too time-consuming. Instead, we issued each prompt five times in separate conversations, producing five distinct code versions per task. This method also allowed us to manually verify the autograder's results for each generated output.

We then evaluated each LLM-generated solution using the appropriate assignment or project autograder. If the initial submission failed any autograder test, we continued the conversation with the LLM, asking for corrections. The motivation behind this is that students are allowed to make multiple submissions to the autograder. Hence, they would certainly ask the LLM to fix errors in their erroneous code. Furthermore, as Zheng et al. [141] observe, well-structured multi-turn interactions often produce higher-quality code than single-turn prompts. We used two basic methods. The first one involves reporting the specific autograder failure with a message like "`The autograder reports an issue with [problem]. Here is the error message: [error message]`". If there were multiple problems, we would list them accordingly. The second

```
I am a first-year university student
taking a C programming course. I have a
project to complete that involves
implementing a small game based on word
search in a grid. To do this, I need to
fill in the wordgrid.c file by
implementing the functions it contains.
The projects detailed instructions,
including each function’s role are in a
file named enonce.pdf, in the ZIP archive
I provide you.

The functions to complete in wordgrid.c
are:
[Function signatures]

The ZIP archive also contains other files
which cannot be modified. Their purpose
is explained in the instructions.

I must implement the functions listed
above and define the Wordgrid_t structure
in the wordgrid.c file so that the
program works correctly. Can you help me
implement all of this? The code must
compile and respect the instructions.
```

(a) Prompt using ZIP archive

```
I am a first-year university student
taking a C programming course. I have a
project to complete that involves
implementing a small game based on word
search in a grid. To do this, I need to
fill in the wordgrid.c file by
implementing the functions it contains.

Here is the principle of the game:
[Game principle]

I have a wordgrid.h file at my disposal,
the content of which is below. The
wordgrid.h file must not be modified.
[Short WordGrid description]
wordgrid.h: ‘‘‘[wordgrid.h content]‘‘‘

Here is the list of functions to
implement along with their explanations:
[Function name]: [Explanation]

The wordgrid.c file (to be completed)
contains:
‘‘‘[wordgrid.c file content, containing
the function signatures]‘‘‘

I must implement the functions listed
above and define the Wordgrid_t structure
in the wordgrid.c file so that the
program works correctly. Can you help me
implement all of this? The code must
compile and respect the instructions.
```

(b) Self-contained Prompt

Figure 4.2. Code generation prompt template for Project 1 of 2024–2025. The prompt on the left was accompanied by a ZIP archive containing the PDF instruction file which also contained advice and visual support and all project files. This version offered the model a complete view of the materials at its disposal. In version 2, the prompt was self-contained. It omitted the PDF's illustrations and included only the code snippets necessary to implement the required functions.

method simply consists in telling the LLM "`It does not work. Try again`". The first method served as the default strategy, while the second was applied when the entire code was incorrect and, in some cases, to compare the two approaches. These minimal strategies were crafted to reflect typical student behavior.

4.4 Supplementary Experiment with GitHub Copilot

For the first two assignments (TP0 and TP1 of 2022–2023), we also used GitHub Copilot [14], focusing specifically on its code completion feature. Our process was simple. We created a code file, placed the assignment instructions at the top as a comment and added the provided function signature. We then followed Copilot's suggestions to complete the code without any manual edits. We applied this method only to TP0 and TP1 because those exercises required

```
I am a first-year university student taking a C programming course. I have a project to
complete that involves implementing a small game based on word search in a grid. To do
this, I need to fill in the wordgrid.c file by implementing the functions it contains.

### Game principle
[Game principle, containing the example in text form]

### Files at my disposal
- **wordgrid.h**: This file must not be modified. [Short WordGrid description]
- **wordgrid.c**: This file contains the function signatures and the Wordgrid_t
structure definition, which must all be implemented.

### wordgrid.h content
‘‘‘[wordgrid.h content]‘‘‘

### Functions to implement
[Function name]: [Explanation]

### wordgrid.c content
‘‘‘[wordgrid.c file content, containing the function signatures]‘‘‘

### Your task
I must implement the functions listed above and define the Wordgrid_t structure in the
wordgrid.c file so that the program works correctly. Can you help me implement all of
this? The code must compile and respect the instructions.
```

Figure 4.3. GPT-4o-improved code generation prompt template for Project 1 of 2024–2025. We provided the original self-contained prompt and asked the model to produce the best possible version for an LLM. GPT-4o returned this more structured prompt and recommended including the visual example from the PDF instructions in text form.

writing a single function and had brief instructions. We did not apply it to the other tasks because integrating Copilot's chat completion proved too complex for them. Our goal was to determine whether Copilot alone could solve the problems. Despite it not being a typical LLM, we deemed Copilot worth testing because of its impressive popularity [15] and the fact the Pro version is freely accessible to verified students [142].

4.5 Results

4.5.1 Assignments

The LLM handled TP0, the introductory warm-up task, with ease.This is shown in Table 4.1, which displays the grades obtained by the LLM before and after making corrections on each task. On TP0, GPT-4o achieved a perfect 20/20 score on its first answer without any follow-up in all five attempts. Copilot also completed the assignment with a 20/20 score. Similarly, TP2 from 2022 presented minimal difficulty as the model again obtained 20/20 on its first generation in each of five attempts.

TP1 from 2022–2023 was also solvable by the LLM but its performance was lower than on TP0 and TP2. In all five runs, the model scored 17 out of 18 because it used an auxiliary function without providing its prototype first. This omission caused a compilation warning and cost one point. To solve this, we provided the model with the autograder's feedback,

Table 4.1. Code generation experiments summary table. The table shows the autograder grades of the five codes generated by the LLM, before and after correction. Code often fails to compile at first but these errors are easily fixed by asking for corrections. As task difficulty grows, the LLM struggles more as observed with TP1 and Project 1 from 2024–2025. Still, requesting corrections consistently improves results. Additionally, using a well-structured prompt can help.
*For the project, the numbers are those obtained after a first round of compilation. For all other tasks, they are counted straight from the raw model output.

Task	**Prompt**	**Correction**	**Grade for Attempt #**					**Max Grade**	**Correct Attempts (/5)**
			1	**2**	**3**	**4**	**5**		
TP0 2022 (Binary sorting)	Standard	Before	20	20	20	20	20	/20	5
		After	20	20	20	20	20		5
TP1 2022 (Interval union)		Before	17	17	17	17	17	/18	0
		After	18	18	18	18	18		5
TP2 2022 (Cartesian to polar)		Before	20	20	20	20	20	/20	5
		After	20	20	20	20	20		5
TP3 2023 (Arithmetic expressions)		Before	6	5.5	6	5.5	6	/8	0
		After	8	8	8	8	8		5
TP4 2022 (Python)		Before	18	18	18	18	5	/20	0
		After	20	20	20	18	20		4
TP2 2023 (mapImage)		Before	8	20	20	20	20	/8	1
		After	8	8	8	8	8		5
TP1 2024 (Tiles)	No Example	Before	3.5	3.5	3.5	3.5	7.5	/8	0
		After	4	4	3	8	8		2
	Example	Before	3.5	3.5	3	4	3		0
		After	8	4	3.5	4	3		1
Project 1* (HexGame)	ZIP	Before	9	9	9	11	12	/14	0
		After	9	9	10	13	9		5
	Self-contained	Before	9	10	12	12	12		0
		After	10	14	14	14	13		3
	GPT-Improved	Before	12	12	14	14	14		3
		After	14	14	14	14	14		5

continuing the previous conversation rather than starting a new prompt. The model was then able to identiy the missing prototype, correct the code, and receive full marks in each of the five conversations. GitHub Copilot's results also showed that TP1 was more challenging than TP0. Its generated code only achieved a 9/18 grade. Like GPT-4o, it missed the auxiliary function prototype but lost eight additional points due to logic errors.

The third assignment of the 23–24 academic year proved more challenging for GPT-4o. It did not earn full marks in any of its five attempts. In every case, its `exprPrint` function was wrong, always printing numbers as integers instead of the required float format. Two of those attempts also produced compilation warnings. One was caused by a case label that was not followed by a break statement and another for placing a label inside a declaration, while labels can only be part of a statement. After informing the model of these autograder errors, it directly fixed its implementation on its first revision and achieved an 8/8 score in all five conversations.

TP4 differs slightly because it is implemented in Python. GPT-4o scored 18/20 on four of its five attempts, losing two points each time due to slower performance than the reference solution. We then asked the LLM for correction by providing the autograder output. For three attempts, the model produced faster code and obtained 20/20. On the fourth, the feedback did not help. In the fifth initial attempt, the model again lost two points for efficiency but only achieved a score of 5/20 because its logic was incorrect. Since the autograder's output simply indicated that most of the solution was wrong, we just prompted "This is wrong. Try again". The revised code scored 18/20, once again lacking efficiency. Finally, after we noted the speed issue to the model, it generated faster code and earned 20/20.

For the 2023–2024 TP2, only one of the LLM's five submissions received a full score on the first try. The other four exhibited compilation issues once again because they included nested functions and unused variables. Nested functions are forbidden by ISO C [143] and unused variables trigger compilation warnings. We then provided the autograder's error messages and asked the LLM to correct them. Although it fixed the flagged issues, new errors appeared, namely missing function prototypes. After sharing these new messages and asking for another correction, the model finally obtained 8/8 on each remaining submission.

GPT-4o had difficulty with the first 2024 assignment, as expected. Using the initial prompt without the example, it scored 3.5/8 in four of five attempts because of compilation and reasoning errors. In the fifth attempt, it obtained 7.5, missing only a function prototype. After seeing the autograder output, it fixed this and received a full score. For two of the four low-scoring generations, corrections fixed compilation errors but not logic errors and further attempts to correct the logic failed. For the remaining one, asking for a correction even worsened compilation, decreasing the score from 0.5 to 0, and additional fixes did not help. For the last 3.5/8 attempt, instead of giving the model the autograder's feedback, we simply stated that the code did not work and asked it to try again. Unexpectedly, GPT-4o produced fully correct code.

When using the second prompt version with an example, the model's performance declined further. The generated code had compilation issues in two of five attempts and showed reasoning errors in all five. We asked for corrections by providing the autograder's output in three of these runs and only one then received a full grade. In the other two, it merely fixed compilation without solving the other problems. We applied the more direct correction approach in the remaining two conversations but it had no effect on the grades. Overall, the model performed less effectively on this assignment, earning full grades only three times out of ten for this assignment where two of those were only after correction.

4.5.2 Project

Lastly, we conducted tests with Project 1, which was chosen to challenge the LLM with complex function interactions and longer instructions. Note that each code submission initially led to compilation errors so we first requested to correct these. All reported grades are those obtained after the compilation corrections, which were all successful.

When we provided the prompt with a ZIP archive, the model yielded three solutions at 9/14, one at 11/14 and one at 12/14. All errors resulted from incorrect function logic. After requesting corrections, two of the 9/14 solutions' grade remained unchanged and one improved to 10/14. The 11/14 code rose to 13/14 but further correction attempts never exceeded that score. Surprisingly, the solution that first scored 12/14 dropped to 9/14 after correction.

With the self-contained prompt, the grades improved to one 9, one 10 and three 12s. After correcting the original code, GPT-4o obtained 14/14 for three of them (two originally scored 12 and one scored 10). One of the remaining 12s reached 13/14, and the 9/14 case improved to 10/14.

Finally, the GPT-improved self-contained prompt produced even better results. Two of the five attempts earned a perfect 14 immediately while the other three obtained 12. After requesting corrections for those three, each of them reached 14 on the first revision. This indicates that a clearer prompt structure and the inclusion of a text example help the LLM generate higher-quality code, confirming findings in the literature [118].

4.5.3 Conclusion

Overall, our results and Table 4.1 indicate that code often fails to compile at first but these errors are easily fixed by asking for corrections. As task difficulty grows, the LLM struggles more, as observed with TP1 and Project 1 from 2024–2025, despite GPT-4o being the best model available at the time. Still, requesting corrections consistently improves results. Additionally, using a well-structured prompt can help. In practice, students can obtain high autograder scores on these tasks with minimal effort.

However, our study makes a couple of simplifying assumptions, notably about students' behaviour. First, we assumed they would not translate instructions nor create their own examples. We did not survey or conduct experiments directly involving students to analyze their LLM usage patterns. Second, our application of the two correction approaches was somewhat ad hoc rather than systematically designed or rigorously motivated. Nevertheless, code generation and solving the TPs and projects with an LLM are not the central focus of this thesis. In addition, even with these assumptions and simple methods, we demonstrated that LLMs can meaningfully aid in solving most of the course tasks.

Chapter 5

Model Selection

5.1 Model Comparison and Selection

Before diving into the code grading and correction experiments, we selected the LLM we would use. As noted in Section 2.2.3, we restricted our choice to open source models. Furthermore, since our resources were limited, we focused on smaller-scale models. Among these, six stood out as the top performers at the outset of our experiments.

The first two models were both from Meta's Llama series, namely the 8B-parameter version of Llama 3.1 [72] and the 3B-parameter version of Llama 3.2 [84]. Llama 3.1 also exists in 70 billion and 405 billion versions [72]. However, these sizes were deemed excessive for our computational resources, especially given our intention to fine-tune and consistently use the same model across all experiments. Similarly, Llama 3.2 offers larger multi-modal versions of 11B and 90B parameters [84]. Nevertheless, these are restricted for on-premises use in the EU under GDPR and emerging AI regulations [84, 144]. Our third choice was Mistral's 8B Ministral model [81]. We also considered their code-focused model, Codestral [64], with 22 billion parameters. DeepSeek-Coder-V2-Lite, a 16B-parameter Mixture-of-Experts model was also assessed [71]. Finally, Alibaba's Qwen-2.5-Coder [65], for which 1.5B, 7B and 32B-parameter versions were available at the time, was also part of the list. Note that its 14B version had not yet been released and like Llama 3.1 70B, the 32B variant was judged too large for our needs.

Table 5.1 summarizes each model's key attributes, namely whether it is specialized for code, its total number of parameters and its maximum context window. In addition, it presents their results on four code-related benchmarks. The first one is HumanEval [5], OpenAI's Python benchmark comprising 164 hand-crafted programming problems. It measures a model's ability to produce functionally correct code solutions [5]. BigCodeBench [145, 146] is a more extensive suite of 1,140 Python tasks drawn from 139 libraries across seven application domains. It evaluates a model's capacity to follow complex instructions and compose multiple function calls and offers two splits. The "complete" split uses full docstrings and the "instruct" split provides concise natural-language prompts. We report the average score over both variants [146]. The third benchmark we use is Aider [95], comprising 133 exercises from

Table 5.1. Model comparison. The table describes each model, indicating whether it is specialized for code, its total number of parameters and its maximum context window. In addition, the table presents the LLMs' results on four benchmarks [5, 86, 95, 145]. For all benchmarks, we report pass@1, except for Aider, for which we report the percentage of tasks completed in at most two attempts. **Bolded values** represent the best score for each benchmark. Qwen2.5-Coder-7B achieves the highest scores on three of the four benchmarks.

Model	**Code specialized**	**# Params (B)**	**Context window (k)**	**Benchmark results**			
				HumanEval	**BigCodeBench**	**Aider**	**LiveCodeBench**
Llama 3.1	No	8	128	72.6	36.6	37.6	–
Llama 3.2	No	3	128	61.0	25.9	26.3	–
Ministral	No	8	128	76.8	–	–	–
Codestral	Yes	22	32.8	81.1	**47.1**	51.1	32.9
DeepSeek-Coder-V2-Lite	Yes	16	128	81.1	42.2	55.6	24.3
Qwen2.5-Coder	Yes	7	131	**88.4**	44.6	**57.1**	**35.9**

Exercism [147], a community-driven coding-exercise platform. It does not evaluate code generation but code editing [95]. It also only covers Python. An updated version of the benchmark was released in December 2024, which includes 225 exercises in six languages (including C++ but not C) [95]. However, we only report the results on the original version as the new version was released after the start of our experiments and the models we consider were not evaluated on it [95]. Lastly, we include LiveCodeBench [86], a benchmark constructed from several hundred competitive programming problems drawn from LeetCode, AtCoder and Codeforces. It is contamination-free, meaning none of its test cases overlap with or appear in any model's training data, ensuring evaluation measures true generalization rather than memorization. LiveCodeBench assesses models in multiple languages including C [86], which is central to this thesis. Moreover, like Aider, it does not only cover code generation. It also evaluates code execution, test output prediction and most importantly, self-repair [86], matching our interest in error detection and correction. This benchmark is regularly updated. We report model scores on LiveCodeBench problems dated from May 2023 to September 2024 [148, 149].

Note that for all models, we use their instruct versions (omitting "Instruct" in the names for brevity), except Codestral [64] which does not have an instruct release. An LLM's Instruct version is obtained by fine-tuning it on a dataset of instruction-prompt-response pairs using supervised learning and often reinforcement learning [132]. This process enables the model to follow explicit commands more accurately [132]. We use these versions because they better interpret our prompts and produce outputs better aligned with our experimental requirements.

In Table 5.1, there are missing values for the two Llama versions and Ministral. This arises because they are not code-specialized models. Moreover, as could have been expected, their performance is below that of the models designed for code. Notably, despite its small size, Qwen2.5 Coder 7B achieves the highest scores on three of the four benchmarks. It was only outperformed by Codestral on BigCodeBench. In particular, it is the best model on the multi-lingual LiveCodeBench and Aider benchmarks, which also evaluate code editing skills. It also offers the largest context window, although even 32k tokens is more than enough for our work. Based on these results and informal manual tests to find bugs in student code, we selected Qwen2.5-Coder-7B for our experiments. Its smaller size also suited our limited computing resources and our plan to fine-tune the model.

5.2 Qwen2.5 Coder

Qwen2.5 Coder [65] is a code-focused member of Alibaba's Qwen family. It was released in late 2024 under the Qwen Research License Agreement, Alibaba Cloud's custom research-oriented license. The Qwen2.5 Coder variant is available in five sizes ranging from 0.5B to 32B parameters [65]. It quickly established itself as one of the top-performing open models in its class. Its strong results stem from pretraining on over 5.5 trillion tokens, drawn primarily from public code repositories like those from GitHub supplemented by extensive web scraped collections of code-related texts [65]. The final data mixture comprised 70% code, 20% general text and 10% math [65]. During data preparation, any examples overlapping with standard benchmarks such as HumanEval, MBPP, GSM8K and MATH were removed. Also, additional synthetic code samples generated by its predecessor, CodeQwen1.5, were added to strengthen performance on less common languages and complex algorithmic tasks [65].

Built on the refined Qwen2.5 Transformer architecture, Qwen2.5-Coder benefits from enhanced training stability and a multilingual tokenizer that supports Python, C/C++, Java, JavaScript, Go and other languages [65]. After base pretraining, the model underwent instruction tuning. This tuning combined supervised fine tuning and direct preference optimization on a curated set of coding prompts, covering tasks like bug fixes, code reviews and conversational programming assistance [65]. These steps improved its responsiveness to real world queries such as "Help me debug this function" or "Write a docstring for the code above".

By combining extensive, high-quality data with targeted fine-tuning, Qwen2.5-Coder delivers not only accurate code generation but also reliable code reasoning and repair capabilities [65]. Its balance of performance, openness and efficiency makes it especially well suited for our research scenario where both capability and resource constraints are important.

5.3 Choice Limitations

Even though Qwen2.5-Coder exhibits impressive performance, particularly for its size, in hindsight, our model choice was maybe not optimal. First, as Table 5.1 shows, it is not the best model on the BigCodeBench benchmark. In addition, time constraints prevented us from testing every model exhaustively or including multiple models in our subsequent experiments. Instead, we trusted the reported benchmark performances. However, finding reliable benchmarks that cover both C programming and code editing, with results for our small-scale models, proved difficult. Benchmark results also vary between sources and change over time as benchmarks are updated. The figures in Table 5.1 reflect the state of benchmarks when our experiments began. In particular, the LiveCodeBench scores were retrieved from the Qwen2.5 release paper [149] and may be inflated. We cannot verify this since the LiveCodeBench website [150] no longer provides them. Still, scores for other benchmarks were obtained from various external sources [95, 146, 151], making them more reliable.

Chapter 6

Code Grading Experiments

The first code correction experiments we ran were automated grading experiments. More specifically, we asked the LLM to assign grades and feedback to past student submissions from the "Additional Information Technology" course. We then compared the LLM-predicted grades and comments with those given by the human graders.

6.1 Coding Tasks

We conducted these experiments using TP1 and Project 1, both from the 2023–2024 academic year. TP1 was chosen because its grade distribution was more varied than those of the other TPs in C. For instance, only about 40% of students earned the maximum grade, compared to 79%, 84% and 56% for TP0, TP2 and TP3 of the same year. Asking the LLM to grade almost only correct submissions would not have been informative. Project 1 was chosen to evaluate grading on a task that is more complex than TPs but still more manageable than other projects. We used the 2023–2024 version because the 2024–2025 project grades were not yet available, preventing comparison with actual student results. Both tasks are described in Section 3.3.

There were 171 graded submissions for the TP and 170 for the Project. The TP was graded out of 20 points and the Project out of 22. The Project assessment used eight criteria. The first one, "Warnings" (/1) checks compilation and errors. The "hexgameCreate/Free" (/2), "Core Functions" (/5), "hexgameGetNumberOfIslands" (/5), "hexgameGetWinner" (/1) and "hexgameIsOver" (/5) criteria each verify that the specified functions work correctly and meet requirements. The "hexgameGetNumberOfIslands" criterion also assesses how efficiently the student implemented this function. "Complexity" (/2) confirms that the reported complexity for `hexgameGetNumberOfIslands` matches the implementation and "Memory Leaks" (/1) ensures there are no leaks. The TP had four criteria. Like for the project, "Warnings" (/1) again checks compilation. The "isPeak" (/5) and "findPeak2D" (/10) criteria verify correct function implementation. Finally, the "Complexity" criterion (/4) checks whether the reported complexity matches the student's implementation. It also ensures that the implementation runs in strictly better than $O(n^2)$ time.

6.2 Grading Approaches

We tested various grading approaches in these experiments. First, we asked the LLM to give a binary response. It simply had to predict whether the student's code was correct or incorrect, with no explanation requested. We defined correct as fully meeting the specification and exhibiting no detectable error. An alternative of this approach also consisted in asking for a correct/incorrect decision but also asked the LLM to explain its decision. These explanations could help identify why the model's judgments may be wrong and allow us to evaluate whether requiring explanations affects the model's grading.

The next method asked the LLM to assign a score out of ten to the code and provide comments to justify it. Lastly, our main approach consisted in requiring the model to score the project and the TP out of their real maximums (22 and 20, respectively). Moreover, the LLM was requested to assign a grade and provide an analysis for each evaluation criterion before providing the total score, like a human grader would. Note that there did not exist any document describing the evaluation criteria in detail or explaining when to assign what grade. Thus, in our prompt, we only supplied the names of the evaluation criteria and a short description. This absence of detailed explanations is a clear limitation. More detailed instructions on how to assign a grade for each criterion could have improved the LLM's evaluations. However, it would have required more effort from the professor's side.

We adopted these various approaches to examine how the model performs on progressively complex grading tasks. We progressed from a simple binary classification to an intermediate ten-point rating of code quality and finally to a detailed, criteria-based approach similar to human grading. This progression let us first verify the LLM's ability to identify correct code. This could be a valuable preliminary phase in a grading process, so that only incorrect submissions would require manual inspection. Next, we evaluated Qwen's ability to predict students' actual grades as a substitute for human graders.

6.3 Prompts

We created distinct prompt templates for each grading method. All prompts were first written in French, matching the language of the task instructions. The final, criterion-based method was our principal focus and the most comprehensive approach so it deserved further experimentation. Hence, we also translated this grading approach's prompt and used it in English. This allowed us to assess whether language affected the model's performance. Figure 6.1 displays the prompt template which we adapted based on the grading approach to be used and to the project and TP contents. As discussed in Section 4.5.2, a well-structured prompt can improve LLM performance so we carefully organized this prompt. It includes all relevant information such as instructions, student code and header files, while excluding details that do not aid the model like submission requirements. We also omitted the example figures from the Project 1 2023 instructions because our LLM is text-only and these images were deemed unnecessary and difficult to convey in text.

The "Task" section of the prompt template is the part that varied by grading method. Figure 6.2 shows the "Task" for the rubric-based approach. The English translations of the

~~~
I am a university professor teaching an introductory programming course in C. My
students have a project/homework where they must implement various functions and define a
structure/two functions in the hexgame.c/peak.c file, following specific instructions and
using the support file hexgame.h/peak.h. I will provide these elements along with the
solution submitted by one of my students.
**Important** : hexgame.h/peak.h is provided to students as a reference file and must not
be modified under any circumstances.

### Project/Homework Information:
1. **Instructions**:
[Project/Homework instructions]
2. **Elements to implement**:
[- ```function signature```: function description]
[+HexGame structure for the Project]
3. **hexgame.h/peak.h file**
[Short file description]
```[File content]```

### Student Code
```[Student Code]```

### Task
[Explain the grading approach]
~~~

Figure 6.1. Code grading prompt template. This is the prompt template for all the code grading experiments. This template is adapted to the project and TP contents and to the grading approaches by filling in the placeholders (between square brackets). We carefully structured this prompt, aiming to get the best LLM performance. It includes all relevant information while excluding details that do not aid the model.
Note: The colours are used to distinguish between Project and TP instructions. Also, we show the English version of the template although the binary and out-of-ten versions on the prompt are only used in French.

"Task" sections for the approaches where the LLM simply judges correctness or assigns a score out of ten appear in Figures B.1 and B.2 in the Appendix.

Additionally, we created enhanced versions of the criterion-based prompt in two ways. First, we provided a perfect solution as an example. As shown in Figure B.3b in the Appendix, we made clear that the example solution is not the only valid approach and asked the LLM to allow alternative solutions. For TP1, the reference solution includes two implementations. The first is a simpler, less efficient searching method that splits the matrix in half and the second is a more efficient method that divides it into quadrants. Hence, we created two versions of the prompt. In one of them, we show only the simpler solution so that the LLM would not expect maximum-grade answers to match the improved version's efficiency and would clearly understand the example. In the other version, we presented both solutions as alternative implementations to illustrate multiple possible approaches. Project 1 had only one example solution, which we provided. These enhanced prompts were also translated in English.

The second way we improved the prompt was by providing the LLM with a grading example. As literature [118] and Section 4.5.2 findings highlight, including a task example helps the model perform better. We therefore supplied a real student's code along with its grade and

```
### Task
Analyze the student code according to the following evaluation criteria and provide your
feedback for each criterion. For each criterion, provide a complete analysis and assign
a grade in the specified format. At the end, compute and display the total score out of
22/20 following precisely the format given below.

### Evaluation Criteria
[i. **Grading criterion** (/x): short description.
- Analysis: describe what to analyse in the code.
- Score: [y/x]]

### Expected Format:
For each criterion, provide an analysis followed by a score in the following format:
**Analysis of [criterion]**
(Score : [x/y], where x is the assigned score and y is the maximum score for that
category).
Finally, conclude your evaluation with the following standardised format:
Total Score : [x/22/x/20]
where x is the sum of points obtained across all criteria.

**Important:**
- Strictly adhere to the required format for each score (e.g., Score: [x/y]) and for the
total score (e.g., Total Score: [x/22/x/20]).
- Each criterion's analysis must precede its corresponding score.
- Do not add any text after the total score.

**Example Format:**
Analysis of Warnings
[Detailed analysis here.]
Score : [1/1]

Analysis of hexgameCreate/free/isPeak
[Detailed analysis here.]
Score : [2/2/5/5]

...
Total Score : [22/22/20/20]
```

Figure 6.2. Explanation of the criteria-based grading approach. This piece of text is inserted in the "Task" section of the prompt for the code grading experiments.
Note: The colours are used to distinguish between Project and TP instructions. These instructions are also used in French.

comments given for each criterion, as displayed in Figure B.3a in the Appendix. It would have been more realistic to use LLM-generated code and autograder feedback as grading example. In practice, we would not have any prior student submissions before grading the student codes. However, because the final grades (out of 20 for the TP and 22 for the Project) are given by human graders, they include criteria that the autograder does not cover. This mismatch would cause the example feedback to differ from the grades the model is expected to predict. Consequently, we chose to use actual student submissions and feedback instead. Every time we asked the LLM to grade a student code with this prompt, we filled the template with a randomly selected submission from a filtered set of student codes. We built this set in three possible ways. The first way consists in including all submissions with a grade below the maximum to show examples of errors to notice and how to write feedback about them. In the second way, the set includes all student submissions. The last method consists in including only submissions scoring between a specified minimum and maximum grade. For the TP, we chose the grade interval [9, 16] while we chose [10, 19] for the project. These enhanced prompts were only used in French as the comments from the real student submissions are all in French.

6.4 Response Generation and Processing

We asked the LLM to grade all student submissions using each prompt we introduced. We ran these experiments with Qwen2.5-Coder-7B, based on Section 5.1 results, using its default generation parameters. As the prompt templates illustrate, we asked the LLM to follow specific response formats, allowing us to extract and analyze answers automatically. However, the LLM generations sometimes did not match the required format (missing markers or using a score out of 40 instead of 20) so the extraction failed. I first adjusted the extraction code but also had to review and correct some answers by hand. For TP1, format errors occurred in 35 responses for the ten-point grading, 2 for the French rubric-based grading and 72 for the English rubric-based grading. For the project, there were 9 errors in the ten-point, 3 in the French rubric-based, and 64 in the English rubric-based grading.

The LLM responses obtained from the criteria-based prompt and its derivatives displayed several other inconsistencies. We notably noticed that the reported total grades often did not equal the sum of the subgrades. Hence, we recalculated each total grade from its subgrades for our analyses. A similar issue arose with the TP1 grade predictions. The model sometimes assigned total grades above the maximum of 20, and on several occasions even out of 40, which was inexplicable. Furthermore, also with the TP, two responses were graded out of 28 when using the prompt that included a perfect solution. We thus used our recomputed totals for these cases too. Another issue we encountered in the responses to the prompt containing a grading example is that the LLM gave two negative subgrades. We replaced them with zeros before recomputing the totals. Finally, the model occasionally returned more grades than expected, for example by writing the total score twice, making automated grade extraction more difficult. Thus, the first drawback of using LLMs for grading is that you cannot simply submit a prompt and trust the results. The outputs must be verified and sometimes inspected manually.

6.5 Results

6.5.1 Performance Criteria and Baselines

We assess grade predictions using four principal metrics. First, we calculate the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) between predicted and actual student grades. These apply only when predictions use the same scale as the true grades. In addition, we measure the correlation between predicted and actual grades for all methods. Finally, we report the Area Under the Curve (AUC), treating submissions with the maximum true grade as correct and varying the threshold at which the model labels code as correct. For binary grading methods, the AUC reduces to a single operating point because the model simply labels each submission as correct or incorrect.

As a reference, we include a constant baseline that always predicts the overall median grade for MAE and the overall mean grade for RMSE since these values minimize the respective error measures. The results appear in Table 6.2.

6.5.2 Binary Grading

We start by examining the LLM's binary grading results. Figure C.1 in the Appendix plots predicted against true student grades for the binary approaches, where a prediction of 1 indicates fully correct code and 0 indicates otherwise. Figure C.1a shows that, when grading the project without giving explanations, the model almost always labels code as correct. It occurs 163 times out of 170 (95.88%), as revealed by the many circles at the top of the plot. When the LLM must justify its decision, this bias decreases. It marks code as correct only 91 times (53.53%), which is still above the true rate of full marks (52 cases, corresponding to 30.59%). A similar, though smaller, overestimation appears for the TP. Without explanations, the LLM predicts 113 of 171 codes (66.08%) as correct and with explanations, 101 codes (59.06%). Still, in reality students obtained full marks only 69 times (40.35%).

We built the confusion matrices in Table 6.1 to evaluate whether the model's judgments about each code being correct or incorrect are accurate. The matrices show that the model is more often right than wrong on both the project and the homework, whether or not we ask for explanations. However, its project accuracy is worse than luck with explanations (33.5%) and close to luck without (51.7%). On the TP task, the LLM's accuracy rises to 61.4% with explanations and 55.6% without. This outcome is unexpected because the model predicts codes as correct more often than incorrect, even though most students actually received below-maximum grades. Consequently, in every case except when explanations are requested on the TP, the model performs worse than a simple majority-class strategy of always predicting "incorrect". This simple strategy would achieve 69.41% accuracy on the project and 59.25% on the TP.

Still, Table 6.2, which provides summary metrics for all approaches, shows a positive correlation between LLM predictions and student grades. The model labels submissions as "correct" more often when the student's grade is higher. Figure C.1 also reflects this trend, though less clearly. This pattern is strongest for the TP cases and for the Project task without explanations, where the model almost never predicts "incorrect". It is less pronounced for the Project task with explanations.

Overall, the model performs slightly better in terms of prediction quality when it does not provide explanations. It outerforms the version with explanations in both correlation and AUC. The only exception is on the project where the version with explanations' AUC is only 1% more accurate. However, both models rarely exceed the constant baseline in classification accuracy. Only the version with explanations surpasses the baseline on the TP, with a 2% higher accuracy.

Table 6.1. Confusion matrices for the binary grading approaches. 1 designates a code that is fully correct and 0 designates a code that is not. The matrices show that the model is more often right than wrong except on the project when we ask for explanations.

Project

(a) With explanations

	Predicted 0	Predicted 1
True 0	6	112
True 1	1	51

(b) Without explanations

	Predicted 0	Predicted 1
True 0	57	61
True 1	22	30

TP

(c) With explanations

	Predicted 0	Predicted 1
True 0	47	55
True 1	11	58

(d) Without explanations

	Predicted 0	Predicted 1
True 0	48	54
True 1	22	47

6.5.3 Out-of-ten Grading

As a next step, the model's predictions out of ten, assessing global code quality, are evaluated. In Figure 6.3, it can be observed that, unlike with the binary method, the model generally underestimates code quality for both tasks, as most circles lie below the red line. Despite this bias, it still gives greater scores to higher-grade submissions, as evidenced by the positive correlations in Table 6.2. For identifying fully correct code, the out-of-ten method outperforms the baseline. The ROC curves in Figure 6.4 and the AUC values in Table 6.2 reveal that the out-of-ten method achieves an AUC of 0.585 on the project. This result places it among the top three models and above the binary approach. It is just below the French rubric-based method, which stands at 0.674. On the TP, it displays an AUC of 0.67, which is clearly better than a dummy model but does not rank among the top-performing models. Like the AUC values, the ROC curves in Figure 6.4 are obtained by considering submissions whose true grade is the maximum as correct and varying the score threshold at which the LLM predicts a code to be correct.

6.5.4 Criteria-based Grading

Our main focus is on the results obtained with the prompt using grading criteria. Figures 6.5a-b and 6.6a-b's scatter plots show that for both the project and the TP, the model predicts higher grades than the true ones when the prompt is in French but lower grades when it is in English. Hence, the first notable difference between the two languages is that English prompts lead to lower predicted grades. Nevertheless, in both languages, the model still gives higher predicted grades to student codes with higher true scores, as the correlations in Table 6.2 demonstrate. These correlations are stronger when the prompt is in French.

Table 6.2 shows that, on the project task with the French prompt, the LLM only slightly outperforms the constant baseline, while it falls short with the English prompt. On the TP,

Table 6.2. Performance metrics by prompt, separately for Project and TP. **Bolded values** represent the best score in each column. The basic criteria-based approach in French is consistently the best except in RMSE on the TP.

(a) Project

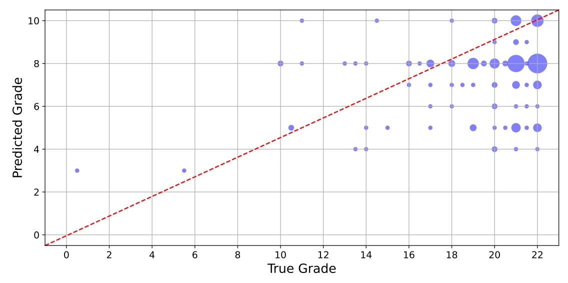
Prompt	**Language**	**RMSE**	**MAE**	**Correlation**	**AUC**
Binary (no explanations)	FR	–	–	0.27	0.52
Binary (with explanations)	FR	–	–	0.12	0.53
Out-of-ten	FR	–	–	0.29	0.59
Criteria-based	FR	**3.17**	**1.90**	**0.44**	**0.67**
Criteria-based	EN	4.02	3.02	0.27	0.55
Reference solution	FR	3.55	2.26	0.34	0.60
Reference solution	EN	3.70	2.61	0.29	0.57
Grading example (non-max grades)	FR	3.57	2.46	0.31	0.51
Grading example (any grade)	FR	3.29	2.31	0.38	0.56
Grading example (grades in [10,19])	FR	3.89	2.92	0.27	0.52
Constant baseline	–	3.29	1.92	–	–

(b) TP

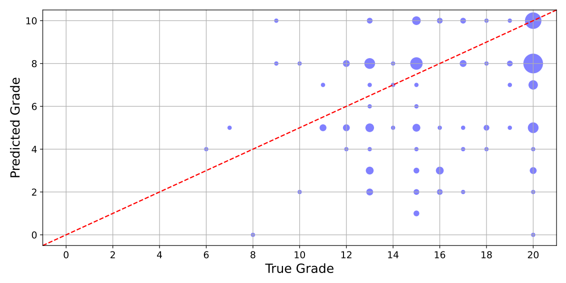
Prompt	**Language**	**RMSE**	**MAE**	**Correlation**	**AUC**
Binary (no explanations)	FR	–	–	**0.44**	0.65
Binary (with explanations)	FR	–	–	0.19	0.58
Out-of-ten	FR	–	–	0.31	0.67
Criteria-based	FR	4.13	**2.97**	**0.44**	**0.70**
Criteria-based	EN	4.68	3.58	0.31	0.69
Reference solution, simple approach	FR	6.33	5.11	0.36	0.68
Reference solution, simple approach	EN	6.68	5.47	0.16	0.60
Reference solution, both approaches	FR	6.26	4.99	0.34	0.69
Reference solution, both approaches	EN	6.85	5.71	0.16	0.61
Grading example (non-max grades)	FR	4.26	3.19	0.37	0.66
Grading example (any grade)	FR	4.88	3.81	0.34	0.67
Grading example (grades in [9,16])	FR	4.14	3.17	0.39	0.69
Constant baseline	–	**3.43**	3.02	–	–

Qwen2.5-Coder-7B with the French prompt beats the baseline in MAE but not in RMSE and is again outperformed by the baseline with the English prompt. Nevertheless, overall, the model's grade predictions are unsatisfactory.

The model struggles to predict exact grades but performs better when grading is framed as a classification task, determining only whether the code is fully correct. As shown by the

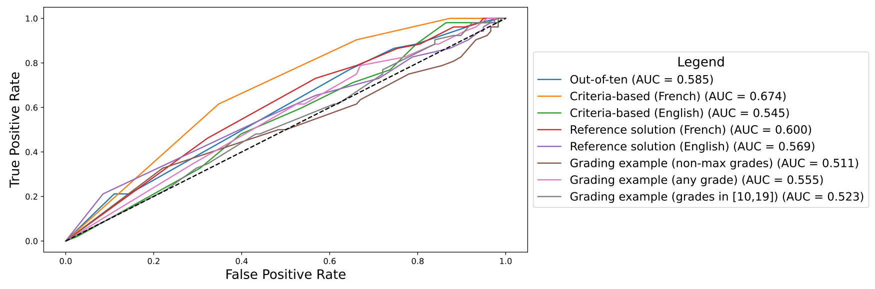


(a) Project

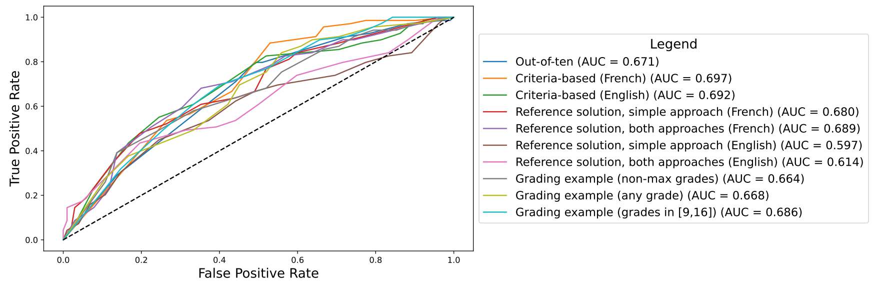


(b) TP

Figure 6.3. Scatter plots for the out-of-ten grading approach. They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot. The plots show that the model generally underestimates code quality for both tasks, as most circles lie below the red line.



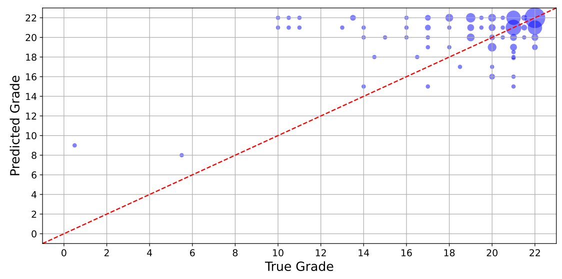
(a) Project



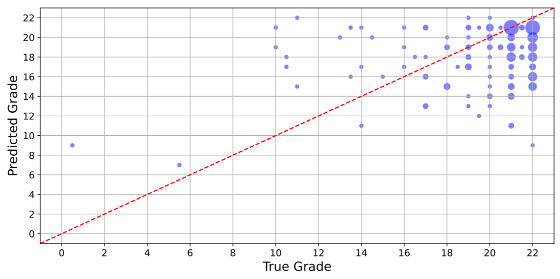
(b) TP

Figure 6.4. ROC curves, separately for Project and TP. The ROC curves are obtained by considering submissions whose true grade is the maximum as correct and varying the score threshold at which the LLM predicts a code to be correct. On both tasks, the model performs better than by chance, with the vanilla criteria-based approach performing the best on both tasks.

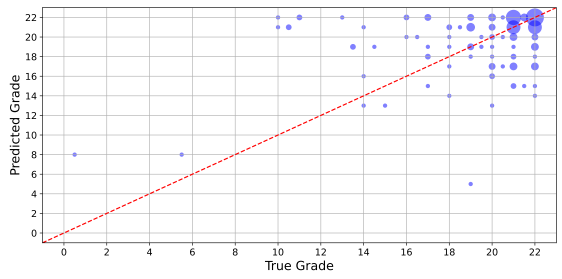
ROC curves and AUC values in Figure 6.4, each AUC exceeds 0.5, indicating performance above chance, which is encouraging. The vanilla criteria-based approach is the most effective at identifying fully correct TP codes. The French variant achieves the highest AUC and the English variant ranks second with respective values of 0.697 and 0.692. On the project, the French approach again leads with an AUC of 0.674, while the English approach only attains a modest AUC of 0.545.



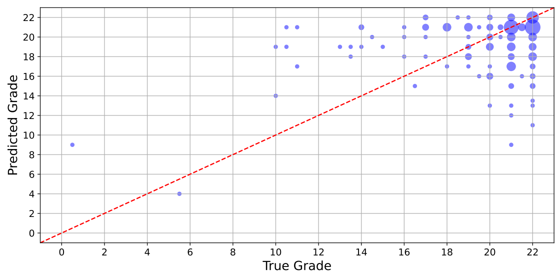
(a) Criteria-based (French)



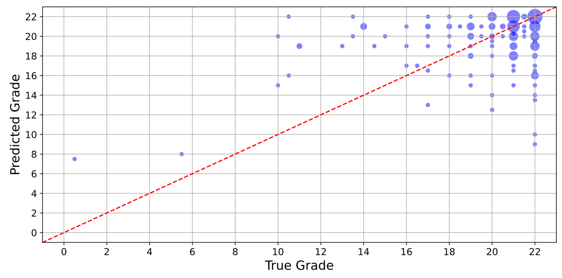
(b) Criteria-based (English)



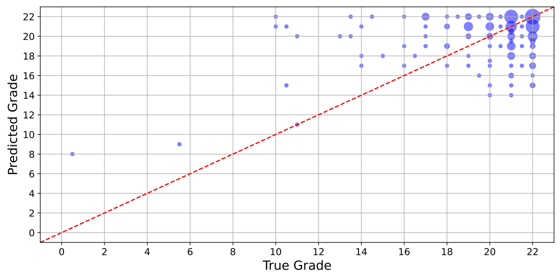
(c) Reference solution (French)



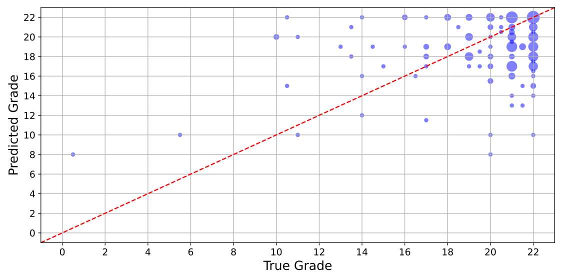
(d) Reference solution (English)



(e) Grading example (non-max grades)



(f) Grading example (any grade)



(g) Grading example (grades in [10,19])

Figure 6.5. Scatter plots for the criteria-based grading approaches on the Project. They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot. The plots show that the model predicts higher grades than the true ones when the prompt is in French but lower grades when it is in English. They also reveal that including a grading example consistently lowers predicted grades.

Manual Inspection

In addition to these quantitative tests, we reviewed the LLM's criterion-based responses by hand for a few TP and project submissions. The aim was to understand what causes the model's predictions to be inaccurate.

For the project, we inspected three submissions in total. Among these were two submissions that the model overestimated in both French and English. The first one was predicted at 22/22 in French and 21/22 in English but was actually graded 10/22. The second one was predicted at 22/22 in both languages while the student received 11/22. We also selected a submission that the English prompt strongly underestimated. The LLM gave it 9/22, although the student obtained 22/22. There was no such underestimation with the French prompt, the most severe one being a 15/22 prediction for a 21/22 true score. In the overestimated cases, the LLM typically failed to detect errors and awarded full marks for criteria the submissions did not actually satisfy. As a concrete illustration, in the second example, the model judged the `hexgameIsOver` implementation to be correct and to meet the requirements, suggesting only minor optimizations. However, it failed to detect that the function simply did not work. The human grader assigned it a 1/5. Also, it is noteworthy that the model sometimes noted efficiency issues but still gave maximum grades. In the underestimated case, the LLM notably gave 0/5 for "hexgameGetNumberOfIslands" despite its actual score of 5/5, citing missing edge-case checks and inefficiencies. Similar incorrect assessments occurred in other criteria.

For the TP, we manually reviewed only two submissions. The first one's grade was underestimated by the model. In both languages, the LLM graded it 6 out of 22 when its true score was 17. With the English prompt, it marked the `isPeak` function as 2/5 despite it correctly handled all cases. With the French prompt, it gave 0/10 for "findPeak2D" while the student actually earned 7/10. These mistakes arose from incorrect claims of redundancy and inefficiency and a misunderstanding of edge-case handling. The second submissions we inspected was selected because it exhibited the largest overestimation. The model estimated its score to be 20/22 while the student in fact obtained 9. The LLM attributed it a 10/10 for "findPeak2D", even though the function's implementation was flawed. It also awarded 4/4 for the "Complexity" criterion, though it actually deserved 0/4. The claimed complexity did not match the code and running the code could lead to infinite recursion.

6.5.5 Grading criteria

The results show that the LLM has difficulty predicting a student's overall grade. However, its poor performance on the total grade may stem from weaknesses in specific grading criteria rather than a general inability. Thus, we also evaluated the model's accuracy for each individual grading criterion.

Figures C.2 and C.3 in the Appendix shows the model's performance on the each criterion. On the project, the model assigned 1/1 to every student's code for the "Warning" criterion even though nine students actually received 0/1. Similarly, on the TP, the model gave 1/1 for "Warning" in every case when given the English prompt. Still, only three students earned 0 on that task. In both tasks and both languages, the model failed to detect any instance where a student scored 0 on this criterion. This failure was predictable since warnings depend on subtle differences in C standards and can be detected more reliably by a dedicated autograder tool than by an LLM.

On the project, for the "core functions" criterion, the lowest grades assigned by the LLM were 4/5 with the French prompt and 2/5 with the English prompt, even though one student actually scored 0/5 and one 1/5. For the “hexgameGetNumberOfIslands” criterion, the LLM using the English prompt gave much lower grades than the true scores. This helps explain why it tended to underestimate overall grades in English. Nearly 90% of students earned 5/5 but the LLM awarded this full score to only half of them in French and to fewer than 5% in English. Lastly, in the human-given grades, almost half of the students got 0/1 for the “memory leaks” criterion but the LLM gave a score of 1 almost every time, for both languages. Detecting memory leaks depends on running the code under specific tools and settings, which LLMs cannot simulate. Simple automated tests catch leaks more reliably. In hindsight, we should probably not have asked the LLM to grade either warnings or memory leaks, since these checks are better handled by autograders.

On the TP, using the English prompt caused grades for “isPeak” and “findPeak2D” to be underestimated, just as was observed for “hexgameGetNumberOfIslands” on the project. With the prompt in French, the model does not consistently overestimate or underestimate. It slightly overestimates “Warning”, “isPeak” and “Complexity”, but underestimates “findPeak2D”.

Tables D.1 and D.2 in the Appendix report the MAE and RMSE obtained in both languages on the TP and the project. On the project, the LLM with the French prompt performs worse than the baseline on six of eight criteria. These criteria are “Warning”, “core functions”, “hexgameGetNbrOfIslands”, “hexgameIsOver” (only on RMSE), “complexity” (only on RMSE) and “memory leaks”. Nonetheless, it outperforms the baseline on the remaining ones. With the English prompt, the model improves on the Warning criterion but simply predicts all code as correct. Otherwise, the French prompt still yields better results than English except for memory leaks, where both perform suboptimally. Similarly, on the TP, the French prompt approach falls below the baseline on three of four criteria. This time though, the English predictions surpass the French on two criteria. Still, overall, its performance remains unsatisfactory, as it exceeds the baseline for only those two criteria.

Although the model slightly outperforms the baseline on a few individual criteria like “Complexity” in the project, its gains are marginal and inconsistent. Overall, these results indicate that the LLM cannot be relied upon to predict student grades with confidence.

6.5.6 Enhanced prompts

As a next step, we assess how prompt enhancements affect the LLM’s grade predictions. We first discuss the effect of including a reference solution in the prompt. On the TP, providing a reference solution causes the LLM to predict lower grades, regardless of the prompt language. This results in a strong underestimation tendency, as can be observed in Figures 6.6c-f. The reason for this could be that showing a “perfect” answer makes the model stricter and penalize any deviation from this solution. Moreover, whether we supply the two reference solutions or only the simpler one does not change this outcome. In fact, as Table 6.2 shows, both MAE and RMSE worsen when a reference solution is included, performing even worse than using prompts without one. The AUC values, also in the table, drop under the same condition, confirming the negative effect of the reference solution. Although we do not provide a thorough analysis of subgrades, we discover the same pattern. Subgrades are lower on average and prediction accuracy declines when a reference solution is given. On the project, the effect is different. Grades are not always lower with a reference solution and the decline in prediction quality is smaller. Nonetheless, adding a reference solution never improves results and often makes them slightly worse, as reflected by Figures 6.5c-d.

We also evaluate the effect of adding a grading example to the prompt. On both the project and the TP, including a grading example consistently lowers predicted grades, regardless of which student submissions are used as examples. This is illustrated in Figures 6.5e-g and 6.6g-i. On the TP, allowing examples to be maximum-grade submissions yields the lowest predicted scores. Perhaps this occurs because presenting a perfect example raises the model's standards and leads to harsher comparisons. For the project, providing examples with grades between 10 and 19 produce the lowest predicted scores. Furthermore, on both tasks, adding a grading example worsens the model's prediction accuracy. This is evidenced by the RMSE, MAE, and AUC values shown in Table 6.2. On the TP, the best performance is attained when examples scored between 9 and 16 are used while on the project, the "select any code" framework performs best. For this prompt enhancement, analyzing the per-criterion model performance does not provide any new insights. A few criteria show occasional improvements when using a grading example but these effects are too small to discuss in detail.

Finally, we examined whether the example code's grade affects the LLM's grading. To achieve this, we computed correlations between the example grade and three measures, as shown in Table 6.3. The first of these measures is the LLM's predicted grade. The second one is the deviation from the true grade, which we use, among other things, to investigate whether higher example grades lead to greater overestimation. The last measure is the absolute deviation, selected to analyze the relation between example grade and prediction accuracy. However, the results differ by example-selection method for both the project and the TP so no clear pattern emerges.

6.6 Conclusion

In conclusion, Qwen2.5-Coder-7B can distinguish fully correct from incorrect submissions better than by chance, especially on the TP where its AUCs are highest. However, its numerical grade predictions remain far from reliable. Its MAE and RMSE values exceed those of a constant baseline grader in most cases. The French version of the vanilla criteria-based prompt proved the most effective overall. It outperformed all other configurations on both tasks and surpassed the constant baseline on every metric except RMSE for the TP. Across all experiments, French prompts yielded slightly higher correlations and AUCs than English prompts, although both struggled to predict exact scores accurately. Performance was similarly weak on the project and the TP. The LLM notably obtained better AUCs on the TP but still fell below the constant baseline for grade predictions. Finally, adding reference solutions or grading examples consistently lowered predicted grades and degraded both accuracy and correlation. Together, these findings suggest that, although LLMs hold promise for highlighting fully correct work, they still require human oversight and further refinement before they can grade student code autonomously.

Table 6.3. Correlation of example grade with various measure, separately for Project and TP. This table reports the correlation of the true grade of the code used in the prompt's grading example with the LLM-assigned grade, the difference between the LLM-assigned grade and the true grade and the absolute difference. The results differ by example-selection method for both the project and the TP so no clear pattern emerges.

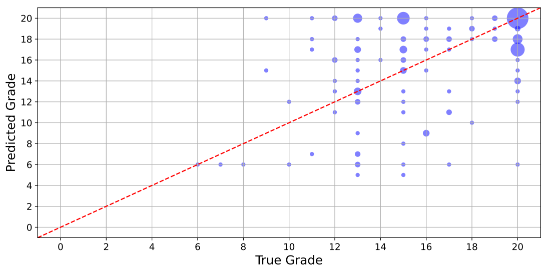
(a) Project

Examples set	Correlation of example grade with...		
	LLM grade	**Deviation**	**Absolute Deviation**
Non-max grades	0.00	-0.04	0.06
Any grade	0.34	-0.08	-0.05
Grades in [10,19]	0.13	-0.09	-0.03

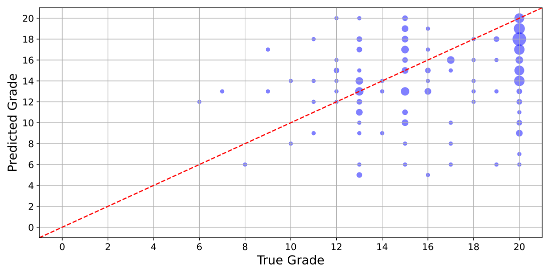
(b) TP

Examples set	Correlation of example grade with...		
	LLM grade	**Deviation**	**Absolute Deviation**
Non-max grades	0.10	-0.06	-0.16
Any grade	-0.03	0.07	-0.02
Grades in [9,16]	-0.10	0.15	0.09

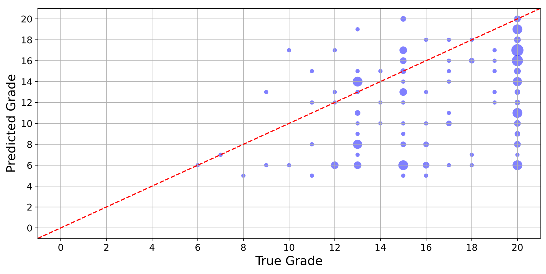
However, our prompt design exhibits limitations. First, we asked the LLM to assign a numerical grade for each criterion without providing detailed instructions. This potentially left too much interpretation to the model on how to map performance to a score. In addition, we opted not to include the autograder's output in the prompt although human graders typically have access to this information. We made this decision out of concern it might overly guide the LLM. Finally, few-shot prompting has been shown to improve performance in the literature [118]. Nevertheless, although Qwen2.5-Coder-7B can handle up to 131,000 tokens of context, we limited ourselves to a single grading example in order not to overwhelm the prompt with too much information. Also, the inclusion of a grading example actually worsened the model's predictions so we did not push these tests further.



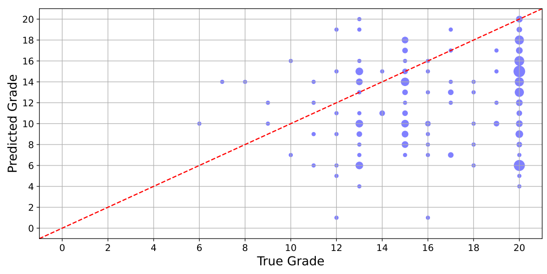
(a) Criteria-based (French)



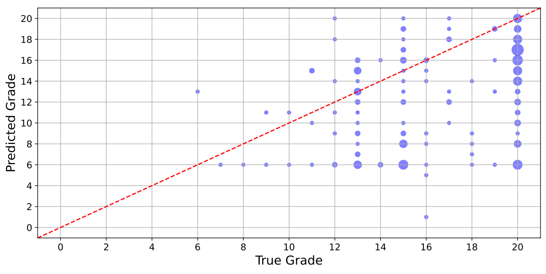
(b) Criteria-based (English)



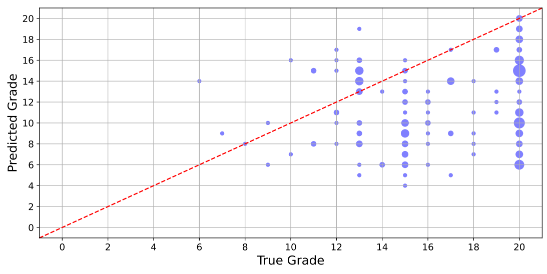
(c) Reference solution, simple approach (French)



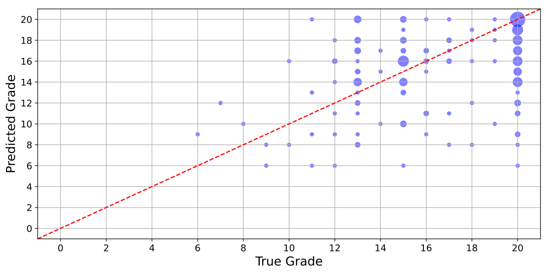
(d) Reference solution, simple approach (English)



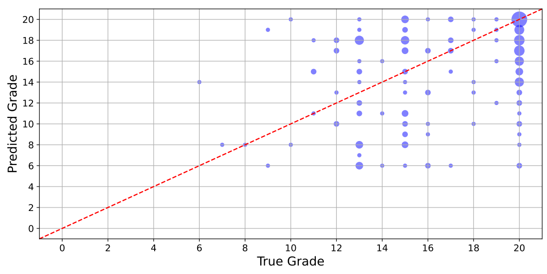
(e) Reference solution, both approaches (French)



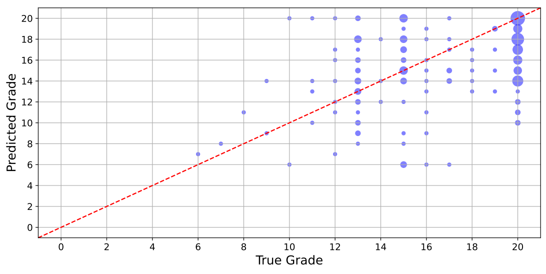
(f) Reference solution, both approaches (English)



(g) Grading example (non-max grades)



(h) Grading example (any grade)



(i) Grading example (grades in [9,16])

Figure 6.6. Scatter plots for the criteria-based grading approaches on the TP. They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot. The plots show that the model predicts higher grades than the true ones when the prompt is in French but lower grades when it is in English. They also reveal that providing a reference solution or a grading example causes the LLM to predict lower grades.

Chapter 7

Error Detection Experiments and Fine-tuning

In our final experiments, we evaluated the LLM's ability to detect errors in student code rather than to grade it directly. We provided the model with code samples and asked it to identify any errors and describe them. By highlighting potential errors, the LLM can help human graders work more efficiently, since, as shown in Section 6, Qwen2.5-Coder-7B is not yet reliable enough to grade code on its own.

Alongside evaluating the original Qwen2.5-Coder-7B [65] on this task, we applied LoRA fine-tuning, a parameter-efficient technique introduced in Subsection 2.5.2. We used this method to strengthen the model's error detection skills for a specific assignment. By training on examples from that assignment, we aimed to specialize the model for this coding task and improve its ability to identify implementation errors.

For these experiments, we once again selected the TP1 2023 peak finding assignment. We chose this task because, as noted in Section 6.1, its student grades show greater variability than those of most other assignments. Moreover, TPs are simpler than full projects, making it more feasible for the LLM to detect errors in student code. Finally, we were already familiar with this assignment from its use in previous code grading experiments in Chapter 6.

The chapter starts with Section 7.1, defining what we consider as correct code in our experiments. We then present the two fine-tuning experiments we conducted and their results. Section 7.2 describes the first one, in which the training set was generated by Qwen2.5-Coder-7B itself. In the second experiment, discussed in Section 7.3, we leveraged larger commercial models to produce a higher quality dataset.

7.1 Correct Code

In these error detection experiments, we needed an automatic way to check whether a given code meets all requirements for the peak detection assignment. We judged a solution correct if it scored 9 out of 9 on a restricted autograder. The original autograder from the course assigns up to 11 points, one for each of eleven criteria. The first criterion checks compilation and any compiler warnings. Eight tests then evaluate the code's behaviour on arrays with different characteristics. A ninth test verifies that the code includes a comment in the required format stating its complexity. The final test measures the implementation's time complexity.

We decided to ignore two of the eleven autograder criteria, meaning the maximum grade codes could receive was 9. The first of these criteria is compilation, which we ignored because a complete compilation failure prevents all autograder tests from running but a compilation warning does not. We treat solutions that only produce warnings as correct because they still work and errors causing warnings can easily be spotted without an LLM.

The second criterion we ignore is time complexity. The autograder assesses this by measuring execution time on arrays of different sizes. However, external factors can cause inconsistent results. Since our primary goal is to verify that the code correctly solves the peak detection problem, we set efficiency aside.

7.2 Initial Fine-tuning Attempt

7.2.1 Training Set

The first step in our fine-tuning process was to design the training set. We drew inspiration from Jiang et al. [99], who studied self-debugging. They showed that prompt engineering methods like few shot prompting perform poorly on this task when applied to small open-source LLMs. Thus, they introduced the LeDeX (Learning to self-Debug and eXplain code) framework in which models are trained on prompt-response pairs. Each prompt presents erroneous code and each response explains the errors and offers an improved code version. To build a high quality dataset, they used an automated pipeline. A first model generates explanations and code fixes. This model can either be the LLM that is fine-tuned itself or another larger model. The generated pairs are then filtered out to keep only examples whose corrected code passes predefined tests. Fine-tuning on this data yielded up to a 15.92% pass@1 improvement across four benchmarks, indicating a significant increase in the LLM's self-debugging capability [99].

Although this thesis does not focus on self-debugging, we adopted a similar approach. We built a training set of prompt-response pairs. Each prompt contains the TP1 2023 instructions, potentially flawed code and guidance for the LLM on how to assess and correct it. Each response explains the errors and provides a corrected code version. In other words, we filled the template shown in Figure 7.1. We also used this template to prompt the LLM when evaluating and correcting student submissions.

We needed triples of erroneous code, error explanations, and corrected code to fill the template. To obtain these, we asked Qwen to generate solutions for TP1 2023 using the prompt shown in Figure 7.2, adapted from Section 4.2. Qwen produced 800 code submissions. 271 passed the autograder tests and 529 failed.

We then took the 529 failing submissions and had Qwen explain their errors and provide corrected versions, yielding the required triples. This was achieved by applying two different

~~~
I am a university professor teaching an introductory course on programming in C. My
students have been assigned a project where they must implement two functions in the
peak.c file, following specific instructions and using the support file peak.h. I will
provide these elements along with the solution submitted by one of my students.

### Statement
[TP1 2023 instructions, files at the student's disposal (peak.h and peak.c, which must
be completed) and description of the elements the students must implement (the
functions and determining the complexity).]

### Student's code
```[INSERT CODE HERE]```

### Autograder's output
[INSERT AUTOGRADER OUTPUT HERE]

### Your task
You must analyse the student's code and determine whether it is correct.

1) If the student's code is fully correct and respects the statement, your answer must
be exactly as follows:
"""### Explanation ###
The student's code is fully correct and respects the statement.

### Correct Code ###
[Replicate the student code exactly]"""

Do not include any text outside these two sections. The section titled ``### Correct
Code ###'' will be used to extract the code portion.

2) If the student's code is not correct, then provide an explanation of what is wrong
and supply a corrected version of the code. Your answer must follow this exact format
with two sections:

### Explanation ###
[Provide an explanation of what is wrong in the code and how to correct it.]

### Correct Code ###
[Provide ONLY the corrected code for the peak.c file.]

Do not include any text outside these two sections. The section titled ``### Correct
Code ###'' will be used to extract the code portion.
~~~

(a) Prompt

```
### Explanation ###
[INSERT EXPLANATION HERE]

### Correct Code ###
```[INSERT CODE HERE]```
```

(b) Response

Figure 7.1. Prompt-response template. This template is used to create training examples by filling the purple parts in. The prompt is also used to evaluate and correct student submissions.

approaches and corresponding prompts, as shown in Figure 7.3. The first prompt asks Qwen to explain the error and then correct the code. The second also asks it to indicate the specific lines causing the problem. Out of 529 attempts, the first method produced 124 fully correct

```
I must implement a function in C for a programming exercise. Here is the statement:

### Statement
[TP1 2023 instructions, files at the student's disposal (peak.h and peak.c, which must
be completed) and description of the elements the students must implement (the functions
and determining the complexity).]

### Your task
I need you to provide me with the completed peak.c file so that the program works
correctly and compiles.
**Important**: Your answer must follow this exact format with two sections:

### Explanation ###
[Provide an explanation of your solution approach.]

### Correct Code ###
[Provide ONLY the complete code for the finished peak.c file.]

Do not include any text outside these two sections. The section titled ‘‘### Correct
Code ###’’ will be used to extract the code portion.
```

Figure 7.2. Code generation prompt. This prompt is used to ask the LLMs to provide TP1 2023 code.

fixes and the second produced 110. We then merged the successful corrections from both methods to enlarge our dataset, yielding 234 unique triples. They each contained the original faulty code, an explanation (with line references when available) and the corrected code. Filling the template in Figure 7.1 with these triples yielded 234 training examples.

We supplemented this training set with 50 examples in which the initial code in the prompt already passed all autograder tests. We chose these examples at random from the 271 correct submissions generated by Qwen. For these cases, the response also follows the Figure 7.3 template. Specifically, the explanation states that the code is correct and the "corrected" code is identical to the original as directed in the instructions given in the prompt. Including these examples teaches the LLM to recognize correct solutions and not always flag code as faulty. With these additional examples, our training set comprised 284 prompt-response pairs. We then used this set to fine-tune the model so it outputs the appropriate response when given each prompt.

One might wonder why the prompts in Figure 7.3 differ from the training template in Figure 7.1. The first, minor difference is that when generating corrections with Qwen, we adopt the role of a student. During training, we present ourselves as a professor. More importantly, the training prompt allows the LLM to acknowledge that the code is already correct. The correction prompt does not allow this, since it is used solely to fix faulty code. We include the autograder's output in both prompts because, in a real grading scenario, it is available. Omitting this information would be illogical, as it helps the LLM make more accurate assessments and corrections.

~~~
I must implement a function in C for a programming exercise. Here is the statement:

### Statement
[TP1 2023 instructions, files at the student's disposal (peak.h and peak.c, which must
be completed) and description of the elements the students must implement (the functions
and determining the complexity).]

### Previous Code
```[INSERT PREVIOUS CODE HERE]```

### Autograder Output
[INSERT AUTOGRADER OUTPUT HERE]

### Your Task
The previous code has issues that prevent it from compiling and working correctly. Your
revised solution must identify and explain these problems by indicating the specific lines that
cause them within the ``### Explanation ###'' section. Then, provide a corrected version
of the code. Your answer must follow this exact format with two sections:

"""### Explanation ###
[Provide an explanation of what was wrong in the previous code. In your explanation,
clearly explain the previous code's problems, specify which lines are problematic, and describe
why these issues occur.]

### Correct Code ###
[Provide ONLY the complete code for the finished peak.c file.]"""

Do not include any text outside these two sections. The section titled ``### Correct
Code ###'' will be used to extract the code portion.
~~~

Figure 7.3. Code correction prompt template. This template is used to ask the LLMs to correct a given code snippet by filling the purple parts in. The teal parts are only included when we use the alternative correction approach, which requires the LLMs to indicate the specific lines that cause issues.

7.2.2 Validation Set

We used the codes that Qwen failed to correct (425 with the basic prompt and 439 with the line highlighting prompt) to create the first part of our validation set. From these, we randomly selected 50 uncorrected codes to track performance during training. At each training stage, we measured how many of these 50 examples the LLM could correct using the training prompt (Figure 7.1). We did this knowing that the base model could not fix them with the correction prompts. Note that we excluded from this set any code that Qwen fixed with one correction prompt but not the other. Those examples were added to the training set and therefore were not eligible for validation.

We limited this part of the validation set to 50 examples because LoRA fine-tuning on a focused dataset typically converges in only a few epochs. Hence, we ran validation every few gradient updates to prevent overfitting [127, 152]. Validating all 50 examples requires the LLM to generate answers for each prompt and then run the autograder, allowing a few seconds between evaluations to avoid cached results. This process takes about 20 minutes in total, or roughly 25 seconds per example.

The second part of our validation set contained 20 examples using the Figure 7.1 prompt with correct initial code. We use these to verify that the model's explanation acknowledges the code is correct and that it returns the original code unchanged, as we ask it to do. These examples were drawn from the 221 correct Qwen submissions not used in training. We limited this set to 20 cases because they are less critical and validation is time-consuming.

It can be observed that we ask the LLM to check correct code for errors and, if any, correct them, while we include the autograder's output in the prompt. However, for the correct codes in this second validation set, this output is empty, since we define correct code as any submission that passes all tests. Moreover, we trained the model on 50 correct examples, teaching it to reply there is no errors when the autograder output is empty. Thus, the LLM should easily identify correct code. Since passing all tests means the code is considered correct, error detection and correction are not really needed for these examples. This validation simply checks that the model remains consistent.

7.2.3 Test Set

Our test set had two parts. The first part comprised the 171 TP1 2023 student submissions. Among these, 127 achieved a 9/9 autograder score and 44 needed improvement. We used student submissions as test examples because our ultimate goal is to detect and correct errors in student code. To evaluate the model, we completed the prompt template from Figure 7.1 with each student submission and asked the LLM to assess and correct them.

Although real student submissions reflect our target use case, we cannot easily verify whether the LLM truly pinpoints their mistakes. To address this, we created a second test set by manually inserting errors into 25 correct codes. We chose three different code variants so that we could evaluate the model across multiple code distributions.

We began with the two reference solutions, creating five erroneous variants of each by introducing a unique error, for a total of ten examples. Next, we selected ten student codes that had scored 20/20 and inserted errors into them. Finally, we modified five Qwen generated codes that passed all autograder tests, yielding 25 erroneous examples in total. Table D.3 in the Appendix lists the specific errors introduced. We inserted simple errors. This allowed us to use our training prompt template, request corrections from Qwen and manually verify whether the LLM accurately detected and fixed each mistake.

7.2.4 LoRA Configuration

We used LoRA [127, 131], a method to efficiently fine-tune neural networks we already introduced in Section 2.5, to adapt the model. LoRA is commonly used with LLMs as they are typically huge deep learning models. It makes fine-tuning efficient by limiting the rank of weight updates and retraining only a small set of parameters. The selected set of parameters depends on the application but is often the most influential layers, like the attention layers in transformer architectures [130]. This approach lowers training costs and storage needs, since only the updated parameters are saved. It also adds no extra inference latency because the updates are additive with respect to the base model parameters. Despite its simplicity, LoRA delivers performance comparable to full model fine-tuning [130].

We applied LoRA fine-tuning with HuggingFace's `transformers` [153] and `peft` [154] libraries. The `transformers` library offers many pretrained language models and straightforward tools to train and use them. The `peft` library focuses on parameter efficient fine-tuning, allowing adaptation of large models with minimal extra parameters and memory. Using `peft`'s LoRA support, we fine-tuned our models more efficiently.

Because our dataset was small, we took several common measures to prevent overfitting. We used a relatively low learning rate of $2e-4$, trained for only three epochs [152, 155]. We also applied a LoRA dropout of 0.05 and monitored both training and validation loss [152, 155]. We employed standard LoRA hyperparameters. The rank r was set to 16 and the scaling factor α to 32 [155, 156]. The scaling factor controls the magnitude of the low-rank parameter updates and is typically set to twice the rank value. Also, we considered reducing r further but doing so would have hindered the mode's ability to learn. We also omitted the LoRA bias term. We fine-tuned only the query and value projection layers in the transformer's attention module, a common choice for LLMs [157]. Resultingly, we updated just over 5 million parameters, representing only 0.07% of the model's 7.6 billion parameters.

We used the default `transformers` AdamW optimizer with weight decay and cross entropy loss. Due to GPU memory limits, we set the batch size to 2 but used four gradient accumulation steps. This means our effective batch size was 8. We applied 20 warmup steps and ran validation every five gradient updates to ensure stable learning.

Training ran quickly. It took approximately 25 minutes on four NVIDIA RTX A5000 GPUs. Nevertheless, validation was run every five gradient updates, making fine-tuning last around eight hours per epoch, leading to a total of 24 hours. We saved both the final checkpoint and the checkpoint at which the model performs best at correcting the 50 challenging validation codes. We call this second one the best checkpoint.

7.2.5 Evaluation

Error Detection and Correction

The corrected code the LLM produces after explaining the errors in the initial submission are mainly used to evaluate the model's performance. Error explanations alone could suffice to help human graders. We require a fixed version of the code to verify that the LLM truly identified and resolved the issues, as in Jiang et al. [99]. More specifically, we judge an LLM's explanation correct if its corrected code passes all autograder tests. This method exhibits drawbacks as it may allow the LLM to output wrong explanations but a correct code and the explanations would be deemed correct. However, manually checking explanations for all 171 student submissions is impractical. Consequently, this approach provides a quantitative measure of the LLM's error detection and correction ability.

We use the percentage of corrected codes, which we call the "Correct Ratio" as our primary metric for error detection and correction quality. We also compare each student submission's autograder score before and after LLM correction. This lets us determine whether the model actually improves code quality.

Then, we treat error detection as a classification task. We measure the model's ability to label code as fully correct or as containing errors. This should be straightforward because the prompt includes the autograder's output and we define any code that passes all tests as correct.

The model might output inadequate explanations yet still generate code that passes all tests. Thus, we manually reviewed several notable responses to understand what causes the LLM to make good or poor corrections. We also examined its responses on the second test set, in which we inserted errors by hand.

We also evaluated fine-tuning on two additional dimensions tied to training set quality as we suspected these to be causes for the model's disappointing performance. First, we measured the diversity of our training codes and compared it to the diversity of student submissions. Second, we checked whether the LLM actually patched the provided code or simply generated a completely new implementation. We did that both during training and testing.

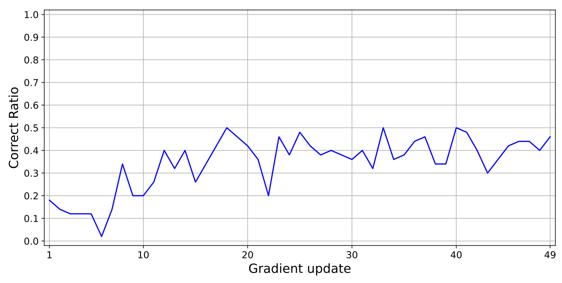
Training Set

The first additional training-set quality aspect we measured is the diversity of the training code. We compared it to the variety found in student submissions. Next, we verified that the LLM genuinely corrected the given code, both during dataset construction and in test-set evaluation, rather than generating an entirely new implementation.

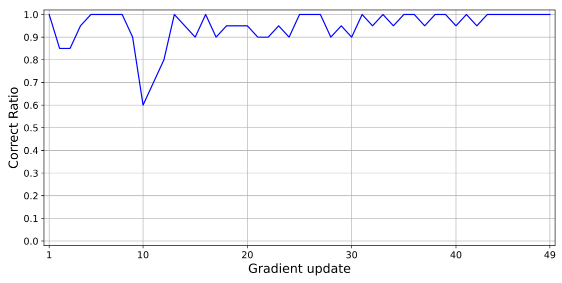
For both of these we used Measure Of Software Similarity (MOSS) [158], a system for detecting resemblance between pairs of code. MOSS is a tool created by Stanford University's Computer Science Department and freely available for non-commercial research purposes [158]. It provides an Internet service where instructors submit batches of source-code files in many languages including C [158]. Upon submission, MOSS computes similarity scores for every pair of programs and automatically filters out expected overlaps, such as standard libraries or instructor-provided code, to reduce false positives [158]. MOSS computes similarity by breaking each program into many small, overlapping chunks. It then assigns a single hash value (its "fingerprint") to each chunk [158]. Similarity is computed as the fraction of those fingerprints the two programs share out of the total number of fingerprints between them. We used MOSS to obtain a measure of similarity between pairs of codes. Every time we used MOSS to compute the similarity between pairs of codes, we provided the `peak.c` template given to the students as base file to MOSS, to avoid counting lines from this template as similarities between the two codes.

To measure a code dataset's variability, we randomly sampled 40 codes from the set and calculated MOSS similarity for every pair. We limited the sample size to 40 because the number of pairs grows rapidly, making the results harder to interpret at larger scales.

In order to check whether the LLM genuinely edited the given codes rather than writing new implementations, we also sampled 40 examples. For each, we used MOSS to compute the similarity score between each corrected code and its original version and then compared that score with the similarity between the corrected code and all other original codes.



(a) Incorrect examples



(b) Correct examples

Figure 7.4. Initial fine-tuning validation results. The left plot shows the evolution of the correct ratio on initially incorrect examples across the gradient updates. The right plot shows the evolution of the correct ratio on initially correct examples.

7.2.6 Results

To evaluate the quality of the fine-tuning, we compare the model's performance at three different stages. These stages are before any fine-tuning, at the best checkpoint, and at the last checkpoint.

Training Set

Training appeared effective. The loss decreased steadily and the model's correct ratio on the training examples rose from 45.69% before fine-tuning to 59.91% at the best checkpoint, reaching 66.38% by the end. This improvement shows the model learned from the training data.

Validation Set

Validation supports these findings. Figure 7.4a shows the correct ratio on initially incorrect validation examples. It begins at 0.2, dips slightly in the first few steps, then steadily rises and plateaus around 0.4, peaking at 0.5. This reflects clear improvement through training. In Figure 7.4b, the correct ratio on initially correct examples also falls briefly, then fluctuates near 1.0, indicating the model retains its ability to recognize code that does not need changes.

Test Set: Student Codes

Table 7.1a summarizes the model's error detection and correction performance on all student submissions. It reports the proportion of submissions that achieve a perfect autograder score after correction and the percentages whose scores improve, decline or remain unchanged. Table 7.1b focuses solely on submissions with a pre-correction score below 9/9, showing how often the model successfully identifies and fixes errors in these cases.

The tables show that fine-tuning increases the correct ratio compared to the pre-fine-tuning model. They also show that the model produces more correct codes after correction. At the best checkpoint, the model achieves the highest overall correct ratio and matches the final checkpoint's performance on originally incorrect codes. The final checkpoint further reduces

Table 7.1. Correction performance on student submissions before and after fine-tuning. The table summarizes the correction performance of the model for each checkpoint, for all fine-tuning attempts. The first subtable shows the results on all student codes. The second subtable shows the results on the subset of student codes that initially did not pass all autograder tests. The "Correct Ratio" metric designates the percentage of codes that are correct. The "Before correction" row gives the percentage of student codes that were correct before correction. The "Initial" fine-tuning is discussed in Section 7.2. The "Explanations" and "Lines" fine-tunings are discussed in Section 7.3.

(a) All codes

Fine-tuning	**Checkpoint**	**Correct Ratio (%)**	**% of codes whose grade have**		
			increased	**decreased**	**stayed the same**
–	Before correction	74.27	–	–	–
–	Before fine-tuning	75.44	22.22	20.47	57.31
Initial	Best	84.80	21.64	14.62	63.74
	Last	81.87	22.81	12.28	64.91
Explanations	Best	61.40	14.04	27.49	58.47
	Last	66.08	9.36	21.05	69.59
Lines	Best	62.57	14.04	22.81	63.15
	Last	71.35	8.77	12.28	78.95

(b) Initially incorrect codes

Fine-tuning	**Checkpoint**	**Correct Ratio (%)**	**% of codes whose grade have**		
			increased	**decreased**	**stayed the same**
–	Before fine-tuning	72.73	86.36	11.36	2.28
Initial	Best	79.55	88.64	11.36	0.00
	Last	79.55	84.09	9.09	6.82
Explanations	Best	45.45	54.55	11.36	34.09
	Last	29.55	36.36	20.45	43.19
Lines	Best	36.36	54.55	6.82	38.63
	Last	25.00	34.09	11.36	54.55

the number of cases where correction lowers the grade, suggesting it is less likely to break working code. Compared to the pre-fine-tuning model, the best checkpoint also fixes slightly more originally incorrect submissions, while maintaining the same number of unchanged cases. Overall, the best checkpoint seems less likely to deteriorate correct code and more likely to improve incorrect code than before fine-tuning.

To evaluate the model's ability to recognize correct code, we examined its classification performance on the test set. Table 7.2 presents confusion matrices for the model before fine-tuning, at the best checkpoint, and at the final checkpoint. In these matrices, a true positive (1) indicates a student submission that passed all autograder tests. A predicted positive means the LLM's explanation declared the code correct. Before fine-tuning, the model never labeled any code as correct. As training progressed, it learned to recognize correct

Table 7.2. Confusion matrices for code-correctness classification before and after fine-tuning. 1 designates a code that is fully correct (9/9 autograder score) and 0 designates a code that is not. The "Initial" fine-tuning is discussed in Section 7.2. The "Explanations" and "Lines" fine-tunings are discussed in Section 7.3.

(a) Before fine-tuning

	Predicted 0	Predicted 1
True 0	44	0
True 1	127	0

(b) Initial fine-tuning: Best checkpoint

	Predicted 0	Predicted 1
True 0	44	0
True 1	109	18

(c) Initial fine-tuning: Last checkpoint

	Predicted 0	Predicted 1
True 0	44	0
True 1	50	77

(d) Explanations: Best checkpoint

	Predicted 0	Predicted 1
True 0	35	9
True 1	103	24

(e) Explanations: Last checkpoint

	Predicted 0	Predicted 1
True 0	44	0
True 1	116	11

(f) Lines: Best checkpoint

	Predicted 0	Predicted 1
True 0	44	0
True 1	127	0

(g) Lines: Last checkpoint

	Predicted 0	Predicted 1
True 0	44	0
True 1	50	77

submissions, increasing true positives. There are no false positives. In other words, when the autograder output shows errors, the model never calls the code correct in its explanations. This behavior is expected because the prompt includes the autograder results.

Qualitative Analysis

Beyond inspecting metrics, we also conducted a qualitative analysis by manually examining selected, noteworthy model responses to student code. In some instances, the model correctly identified working code as correct in its explanation but still modified it and introduced errors. This occurred in four cases at the best checkpoint and eight at the final checkpoint. Typically, these submissions used unconventional structures or overly long functions. The LLM attempted to improve them but produced incorrect code.

We also examined cases where the LLM flagged correct code as incorrect and then made it incorrect. This occurred 30 times before fine-tuning and 12 and 13 times at the best and final checkpoints, respectively. In most of these cases, the model attempted efficiency improvements or misidentified errors, which led to worse code.

Next, we inspected cases where the initial code was correct, yet the LLM failed to recognize this while its revision still passed all tests. This happened 97 times before fine-tuning, 98 times at the best checkpoint and 38 times at the final checkpoint. In most instances, the LLM rewrote the entire program using a standard approach that achieved a 9/9 score.

Table 7.3. Error detection and correction performance on manually introduced errors. The table includes counts of error mentions, successful corrections, and full-code rewrites (out of 25). The "Initial" fine-tuning is discussed in Section 7.2. The "Explanations" and "Lines" fine-tunings are discussed in Section 7.3.

Fine-tuning	**Checkpoint**	**Mentions error**	**Corrects error**	**Rewrites code**
–	Before fine-tuning	7	10	9
Initial	Best	5	7	16
	Last	4	8	14
Explanations	Best	1	4	2
	Last	4	6	0
Lines	Best	7	13	1
	Last	6	15	0

We also examined five of the 58 cases in which a submission passed all nine autograder tests but then scored below 20/20 in the final human review. This outcome indicates issues that the autograder failed to detect. Before fine-tuning, the LLM mentioned the true problem in three cases, fixing it correctly twice but introducing a new error once and in the other two it raised irrelevant issues. At the best checkpoint, it detected the real issue three times but only fixed it once, while rewriting the code extensively in the other two. In the remaining cases, it either left the code unchanged or flagged unrelated errors before rewriting. At the final checkpoint, the LLM declared the code correct in three cases. Out of the other two, it fixed the error in one and in the last case correctly identified the issue but replaced the code entirely.

Test Set: Manually Inserted Errors

We also evaluated the LLM's responses on the 25 manually corrupted examples from our second test set. Table 7.3 shows, for each model version, how often the LLM mentioned the inserted error in its explanation and how often it fixed that error. It also reports how often the LLM rewrote the code (almost) entirely instead of making a targeted correction. Fine-tuning actually reduced the model's ability to detect and correct these simple mistakes. It also increased its tendency to rewrite the code rather than adjust the original implementation. All three model versions struggled to identify basic errors such as off-by-one or replacing $>$ with $\geq$ and frequently reported errors that did not exist. In addition, the model sometimes corrected an error in the revised code even when it failed to mention that error in its explanation.

Training Codes Variability

Another method we employed to evaluate the quality of our fine-tuning consisted in comparing code variability in our training set to that of student submissions. We checked this to try and explain why improvements obtained from fine-tuning were limited. We randomly sampled 40 training examples and 40 student programs and calculated pairwise similarities for each subset. Table 7.4 presents summary statistics for these distances and Figures C.4a and C.4b

Table 7.4. Code variability evaluation. The table provides summary statistics of pairwise code similarity for 40-code samples. It compares student submissions with LLM-generated examples. Similarity is computed using MOSS. The "Initial" training set variability is discussed in Section 7.2.6. The rest is discussed throughout Section 7.3.

Dataset	**Model**	**Mean**	**Median**	**Std**	**Min**	**Max**
Student codes	–	0.08	0.06	0.10	0.00	0.99
Initial training set	Qwen	0.42	0.41	0.25	0.00	0.98
Correct codes	o3	0.54	0.54	0.22	0.00	0.96
Correct codes	o4-mini	0.37	0.33	0.22	0.00	0.99
Correct codes	o3 (low effort)	0.42	0.41	0.25	0.00	0.96
Correct codes	o4-mini (low effort)	0.37	0.39	0.22	0.00	0.95
Correct codes	GPT4.1	0.37	0.36	0.21	0.00	0.95
Correct codes	GPT4.1 (1.5 temperature)	0.21	0.18	0.18	0.00	0.91
Correct codes (multi-sample)	o3 (low effort)	0.23	0.18	0.22	0.00	0.99
Correct codes (multi-sample)	o4-mini (low effort)	0.32	0.30	0.24	0.00	0.98
Correct codes (multi-sample)	GPT4.1 (1.5 temperature)	0.22	0.19	0.21	0.00	0.96
Improved training set	OpenAI models mix	0.20	0.14	0.19	0.00	0.98

in the Appendix display hierarchical clustering results. The table and figures reveal that the student code set exhibits much higher variability, with a median distance of 0.94, compared to 0.59 for the training set. This gap highlights a weakness in our training data that may limit the model's performance on diverse student code.

Genuinely Correcting

Our evaluations already revealed that the LLM often rewrites an entirely new program instead of editing the code provided in the prompt. To measure this formally we sampled 40 training examples and for each corrected code, computed its MOSS similarity to all 40 original codes, as mentioned in Subsection 7.2.5. Table 7.5a summarizes the true pair similarity summary statistics. Table 7.5b reports the "rank" distribution statistics. We define rank as the position of the true original code when all original codes are sorted by descending similarity with respect to a corrected code, meaning the ideal rank is 1.

The first row of each table indicates that, for many training examples, the "corrected" code is actually a broadly rewritten version rather than a targeted fix. This is evident because other original codes often show higher similarity to the corrected code than its true predecessor. The mean rank of the true original is 5 with a median of 2, with ranks reaching as high as 22. Additionally, half of the similarity scores fall below 68%, confirming that extensive modifications are common.

Table 7.5 also presents similarity metrics for student submissions before and after correction across the three model versions. These metrics appear in the first three rows of the table with "Student codes" as pre-correction codes. Before fine-tuning, these results are already worse than for the training set. This likely happens because student submissions are more varied

Table 7.5. Correction effectiveness evaluation. Summary statistics of pairwise code similarity and rank for both model-generated and student submissions, comparing pre- and post-correction versions across model checkpoints. The table assesses whether the post-correction codes are genuine corrections of the pre-correction versions. This is achieved by sampling 40 codes. For each, we use MOSS to compare the corrected code with its original version and then compared that score with the similarity between the corrected code and all other original codes. Rank is defined as the position of the true original code when all original codes are sorted by descending similarity with respect to a corrected code. The pre-fine-tuning and "Initial" results are discussed in Section 7.2.6. The o3 and o4-mini correction of Qwen-generated codes is discussed in Section 7.3.1. The "error introduction" codes and their correction are discussed in Section 7.3.3. The student codes correction obtained from the "Explanations" and "Lines" fine-tunings is discussed in Section 7.3.7.

(a) Similarity

Pre-correction codes	Post-correction codes	Mean	Median	Std	Min	Max
Qwen-generated codes	Pre-fine-tuning correction	0.65	0.68	0.18	0.28	0.98
Qwen-generated codes	o3 correction	0.56	0.64	0.27	0.17	0.98
Qwen-generated codes	o4-mini correction	0.62	0.65	0.20	0.22	0.99
Error introduction (Explanations)	OpenAI correct codes	0.96	0.97	0.03	0.83	0.99
Error introduction (Lines)	OpenAI correct codes	0.97	0.98	0.03	0.89	0.99
Student Codes	Pre-fine-tuning correction	0.39	0.38	0.29	0.00	0.89
Student Codes	Best checkpoint (Initial) correction	0.28	0.20	0.28	0.00	0.99
Student Codes	Last checkpoint (Initial) correction	0.42	0.31	0.37	0.00	0.99
Student Codes	Best checkpoint (Explanations) correction	0.62	0.76	0.39	0.00	0.99
Student Codes	Last checkpoint (Explanations) correction	0.83	0.92	0.23	0.08	0.99
Student Codes	Best checkpoint (Lines) correction	0.84	0.93	0.24	0.00	0.99
Student Codes	Last checkpoint (Lines) correction	0.89	0.98	0.21	0.00	0.99

(b) Rank

Dataset	Version	Mean	Median	Std	Min	Max
Qwen-generated codes	Pre-fine-tuning correction	5.28	2	1.00	1	22
Qwen-generated codes	o3 correction	8.32	3	3.42	1	32
Qwen-generated codes	o4-mini correction	6.04	1.5	1.87	1	27
Error introduction (Explanations)	OpenAI correct codes	1.03	1	0.16	1	2
Error introduction (Lines)	OpenAI correct codes	1	1	0.00	1	1
Student Codes	Pre-fine-tuning correction	8.90	1.5	12.93	1	40
Student Codes	Best checkpoint (Initial) correction	12.23	5	14.73	1	40
Student Codes	Last checkpoint (Initial) correction	11.20	1	14.86	1	40
Student Codes	Best checkpoint (Explanations) correction	7.03	1	12.22	1	40
Student Codes	Last checkpoint (Explanations) correction	1.21	1	0.97	1	7
Student Codes	Best checkpoint (Lines) correction	2.65	1	7.07	1	40
Student Codes	Last checkpoint (Lines) correction	2.00	1	6.09	1	40

than the Qwen-generated examples. Moreover, the training examples were generated by the model itself so it probably handles them better. Fine-tuning makes the model increasingly generate entirely new code rather than editing the original. The median similarity falls and mean rank rises, despite a slight increase in mean similarity at the final checkpoint. These findings suggest that our fine-tuning taught the model to produce a working solution without truly correcting the given code. This probably arises due to the training examples involving extensive rewrites.

Based on these results, we must interpret the improved correct ratios and post-correction grades with caution. The model often generates entirely new code instead of fixing the original. As a result, the correct ratio no longer reliably reflects the quality of its error detection explanation. The LLM's true ability to correct errors may not have improved.

7.2.7 Conclusion

Fine-tuning did raise the validation correct ratio from around 20% up to close to 50%. It also improved post-correction grades on student submissions. However, the last two sets of experiments have demonstrated that our training set exhibited two major weaknesses, namely its lack of code variability and the fact that in many examples, the corrected code was not actually a correction of the original code. Hence, we ran a second fine-tuning experiment aiming at addressing these two problems.

7.3 Improved Fine-tuning Attempt

Our initial fine-tuning attempt exhibited two principal limitations. First, the model often rewrote the buggy code instead of correcting it and the training set reinforced this behaviour. Second, the training set lacked diversity compared with real student submissions, which are our target data. Hence, this section describes the new fine-tuning approach we designed to address these issues. Section 7.3.1 explains how we obtained targeted corrections. Section 7.3.2 discusses how we increased the training set variability. Section 7.3.3 details how we combined these two methods to build an improved training set. Sections 7.3.4–7.3.6 cover the validation set, test set, LoRA configuration and evaluation process. Finally, Section 7.3.7 presents the results of this new fine-tuning attempt and Section 7.3.8 concludes.

7.3.1 Genuinely Correcting

The primary problem in the initial fine-tuning attempt was that our training examples often contained entirely rewritten code instead of true corrections. These taught the model to overwrite solutions and made the correct ratio unreliable. Despite our academic context and limited budget, we decided to use stronger commercial models to try fixing this. We expected them to produce genuine fixes and help us build a more reliable training set. We also wanted to assess whether the issues we encountered were due to Qwen2.5 Coder 7B's limitations.

Consequently, we used OpenAI's o3 and o4-mini models [12] to build a new training set with genuine code corrections. We chose OpenAI models for their strong performance, API ease-of-use and leading status in the field. The o3 model is OpenAI's flagship reasoning model at the time of writing, known as their most powerful across coding, math, science, and visual perception [159]. It displays state of the art performance on benchmarks like Codeforces [86, 87] and SWE-bench [12, 96, 159]. The o4-mini model is a smaller, cost-efficient variant optimized for fast, high-throughput reasoning in coding and visual tasks [12, 159]. It offers near-top performance at lower cost and with higher usage limits [12, 159].

Like in our initial fine-tuning attempt, we asked the o3 and o4-mini models via the OpenAI API to correct the Qwen-generated erroneous codes using the same prompt as before (Figure 7.3). We limited costs by sampling 100 codes per model. The o3 and o4-mini models produced almost only correct codes (96 and 93 respectively)[1]. However, they still tended to rewrite large portions of the code rather than apply targeted corrections. The second and third rows of each subtables in Table 7.5 show that their mean similarities between pre- and post-correction versions on a 40 code sample, which are 0.59 for o3 and 0.62 for o4-mini, are lower than Qwen's 0.65. Their rank distributions are worse as well. Thus, even these leading models failed to provide the code edits needed to build a reliable training set.

We therefore designed a new approach to build our dataset. Instead of generating and then correcting code, we used the OpenAI models to first produce correct TP1 2023 solutions. We then asked those same models to introduce errors into these working codes. This process yielded genuine pairs of faulty and fixed code, ensuring that each "corrected" version truly repairs its corresponding error. The detailed implementation is described in Subsection 7.3.3.

7.3.2 Increased Variability

The second issue was training set diversity. In our new method, we first generate correct solutions so improving diversity means generating more varied correct code. Table 7.4 presents pairwise MOSS similarity for 40 correct codes from o3 and o4-mini. The o3 outputs are less diverse than our initial training set, with mean and median similarity both at 0.54 versus 0.42 and 0.41. In contrast, o4-mini produces more varied code, with mean similarity at 0.37 and median at 0.33. However, neither matches the diversity of real student submissions. Note that we compare correct codes from o3 and o4-mini with the pre-correction codes in our initial training set and student data. This makes the comparison imperfect but it still highlights their relative variability.

Variability in LLM output is typically increased by adjusting generation parameters such as temperature, `top_k`, and `top_p` [160]. Temperature adjusts the randomness of token selection during generation [160]. The `top_k` parameter limits the model to sampling only from the k most probable tokens at each generation step [160]. The `top_p` parameter limits sampling to the smallest possible set of most-probable tokens whose cumulative probability is at least p [160]. For the three parameters, lower values make output more deterministic, while higher values introduce greater randomness and creativity.

Nonetheless, OpenAI's o3 and o4-mini models do not support changing temperature, `top_k` or `top_p`. They only offer a "reasoning effort" setting which can be set to low, medium or high and whose default value is medium [159]. It adjusts how much internal "think time" the model spends before generating its output [159]. Hence, to boost code diversity, we regenerated correct solutions, still using the prompt in Figure 7.2, this time with reasoning effort set to

[1]As a comparison, Qwen obtained only 23.4% of correct codes when asked for simple explanations and 20.8% when asked to pinpoint the erroneous lines.

```
### Your task
I need you to provide me with **five different** complete ‘peak.c‘ implementations, each
using a different approach, so that the program works correctly and compiles.

**Important**: For each implementation, follow this exact format and label them
‘‘Version 1,’’ ‘‘Version 2,’’ up to ‘‘Version 5.’’ Do **not** include any text outside
these sections.

#### Version X
##### Explanation
[Explain your solution approach for this version.]

##### Correct Code
‘‘‘[Provide **only** the complete code for ‘peak.c‘ for this version.]‘‘‘

(Repeat for Versions 1 up to 5)
```

Figure 7.5. "Your task" prompt section for the multi-sample code generation approach. This is used to replace the "Your task" section in the original TP1 2023 code-generation prompt to request five distinct code implementations per response.

low. As Table 7.4 shows, this had no effect on o4-mini's diversity. The mean similarity was unchanged and the median similarity slightly higher. It did improve o3's diversity. The mean similarity fell from 0.54 to 0.42 but the results remained below those of student code.

To address our need for greater output diversity, we turned to GPT-4.1 [26]. It is OpenAI's strongest coding model after their reasoning series [26, 159] and offers control over generation parameters. Unlike o3 and o4-mini, GPT-4.1 allows users to adjust both temperature and `top_p` (though not `top_k`) [26, 159]. Since `top_p` defaults to 1.0 and cannot be raised, we only experimented with temperature. Initial tests showed that values from 1.6 and above almost always yielded nonsensical code, whereas a temperature of 1.5 rarely did. We consequently generated correct TP1 2023 solutions with GPT-4.1 at a temperature of 1.5. As shown in Table 7.4, GPT-4.1's default variability matches o4-mini. Raising the temperature to 1.5 drops the median pairwise similarity to 0.18, making it far more diverse than any other OpenAI model we tested. Also, note that despite the higher temperature, GPT-4.1 maintained a 90% correct ratio.

Generating solutions with GPT-4.1 at temperature 1.5 significantly increased code diversity but still fell short of the variation observed in student submissions. To address this, we designed a new method to boost code variability. We adapted our prompt to ask the model to generate multiple distinct TP1 2023 solutions in a single response. The revised "Your task" section of this prompt is shown in Figure 7.5. We call this the multi-sample approach. We applied this multi-sample prompt to o3 and o4-mini with low reasoning effort and to GPT-4.1 at temperature 1.5. As Table 7.4 shows, o3 and o4-mini saw substantial diversity gains as their average similarities fell by 0.19 and 0.14 respectively though they remained less varied than GPT-4.1. Surprisingly, GPT-4.1's average similarity rose slightly from 0.21 to 0.22 and the median from 0.18 to 0.19. The greatest diversity emerged when combining all three models' outputs, yielding a mean similarity of 0.20 and median of 0.14.

7.3.3 Improved Training Set

To construct our new training set, each example requires three elements to fill the template in Figure 7.1. These elements are a code with potential pitfalls, explanations of those errors, and a corrected code version. We obtained these by first generating correct TP1 2023 solutions and then introducing errors into them.

To generate correct TP1 2023 solutions, we used the multi-approach prompt from Figure 7.5. We queried o3, o4-mini, and GPT-4.1 with low reasoning effort and a temperature of 1.5, issuing 25 prompts to each model. With five codes per response, this produced 375 programs. Out of these, 329 passed all autograder tests where 104 were obtained from o3, 113 from o4-mini and 112 from GPT-4.1. As shown in Table 7.4, this procedure yielded a training set with much higher diversity than our original dataset. This is notably exhibited by its twice smaller average pairwise similarity.

Subsequently, we injected errors into these correct solutions and collected explanations of those errors to fill the Figure 7.1 template. The key challenge was to have the LLM write its explanations as if it were grading the faulty code, identifying and explaining the mistakes, rather than describing how it had inserted them. To achieve this, we developed the prompt shown in Figure 7.6.

Although the prompt instructs the LLM to explain each injected error as if grading the code, manually verifying every example would be too time consuming. To address this, we created an alternative prompt, also shown in Figure 7.6, that asks the LLM only to list the line numbers where it introduced errors. While this approach provides less detailed feedback, it is more reliable for building our dataset efficiently.

We then used those prompts to have the LLMs inject errors into the 329 correct codes and report either explanations or line numbers for each fault. We faced a trade off. Medium reasoning effort and low temperature are likely to yield accurate, instruction following explanations but produce less varied errors. In contrast, low reasoning effort and higher temperature generate more creative errors but may sometimes deviate from our instructions. To balance accuracy and diversity, we divided the workload equally between these two settings across all three models. Our prompts also explicitly encouraged creative error generation to maximize the variety of mistakes.

We applied both prompts, to request error explanations and to list erroneous lines, to all 329 correct codes, yielding two sets of incorrect examples. For the explanation prompt, we excluded GPT-4.1 because it repeatedly described how it injected the errors in the code despite clear instructions. Instead, we used o3 with low and medium effort and o4-mini with medium effort on 82 codes each and o4-mini with low effort on 83 codes. For the line prompt, we used o3 with low effort on 55 codes and with medium effort on 54 codes. We also used o4-mini with low and medium effort on 55 codes each, and GPT-4.1 at temperatures 1.0 and 1.5 on 55 codes each.

The LLMs injected errors into all 329 correct codes using both the explanation prompt and the line number prompt. However, not every injected error made the codes fail the autograder tests. Specifically, 211 of the 329 codes from the explanations set failed at least one test and

```
Take the correct C implementation below and deliberately introduce one or more errors,
which can be any type of error (be creative), while leaving everything else unchanged.
Then you should explain the mistake as if you were grading the code (not as a comment about your own
modification)/give the line(s) in which you introduced error(s) and present the erroneous code.

### Statement
[TP1 2023 instructions, files at the student's disposal (peak.h and peak.c, which must
be completed) and description of the elements the students must implement (the functions
and determining the complexity).]

### Correct peak.c
‘‘‘[INSERT CORRECT CODE HERE]‘‘‘

### Your task
Take the Correct peak.c above and introduce one or more errors but do not change
anything else. Do not add, remove, or modify any comments in the code. Then output
exactly two sections as shown below.
**Important**: Your answer must follow this exact format with two sections:

### Explanation ###
[Explain the error(s) as if you were grading the code. For example: "In line 15, the code uses the
comparison operator incorrectly, so it fails to ..."]

### Erroneous Lines ###
[Here you give the line(s) in which you introduced error(s). First give the line number, then the line
content. For example, if you modify line 15 and change it from ‘if a < b:‘ to ‘if a > b:‘, you should
output: "Line 15: ‘if a > b:‘"]

### Erroneous Code ###
‘‘‘[Here you paste the entire peak.c with the introduced error(s).]‘‘‘

Do not include any text outside these two sections. The section titled ‘‘### Erroneous
Code ###’’ will be used to extract the code portion.
```

Figure 7.6. Error introduction prompt template. This prompt is used to inject errors in correct code by replacing the purple part to generate our improved training set. The blue parts are used to ask the LLM to provide explanations of the errors introduced. The teal parts are used to ask the LLM for the lines in which the errors were introduced.

257 of the 329 codes from the lines set were actually incorrect. There were many cases were the errors the model introduced were memory allocation errors but these are not penalized by our autograder.

We created two separate training sets from these faulty codes. For the first set, we used the original prompt-response template (Figure 7.1), which includes an “Explanations” section. For the second set, we replaced that section with an “Erroneous lines” section, instructing the model to list the line numbers containing errors instead of describing them. The rest of the template remained identical. We kept the sets separate for two reasons. Each of them approximately matches the initial fine-tuning set's size of 232 triples and listing error locations is a distinct task from explaining them.

This new training set creation technique successfully produced genuine original-corrected code pairs, as Table 7.5 shows. The median similarity between each pre- and post-correction code is 0.97 for the explanations set and 0.98 for the erroneous lines set. Only one example in both sets combined has a similarity rank above 1 (rank 2). This represents a major improvement compared to our original training data.

Like we did in our first fine-tuning, we added 50 examples in which the initial code was already correct to each training set. For those cases, the response states that the code is correct and reproduces the original code. We selected these examples from correct codes in which the OpenAI LLMs failed to introduce errors. This yielded a first training set of 211 triples consisting of correct code, an erroneous version and explanations, to which we added 50 correct examples. It also produced a second set of 257 triples, plus 50 correct examples, resulting in final sizes of 261 and 307 respectively.

7.3.4 Validation Set

As in the initial fine-tuning attempt, our validation set comprised two parts. The first part contained initial codes with errors to assess correction performance. The second part contained correct codes to verify that the model recognizes and preserves valid solutions. For the incorrect codes, we reused the Qwen-generated codes that Qwen2.5-Coder-7B could not fix before fine-tuning. This makes direct comparisons with our previous fine-tuning unfair because the new training examples come from OpenAI models, not Qwen. We treat this validation set simply as a progress indicator for detecting and correcting TP1 2023 errors, not as a head-to-head comparison. We also chose to keep these examples because our new dataset creation method did not generate cases that the base Qwen model could not solve. For the correct codes, we randomly selected 20 from those in which the OpenAI models failed to introduce errors and were consequently excluded from our training sets.

7.3.5 Test Set and LoRA Configuration

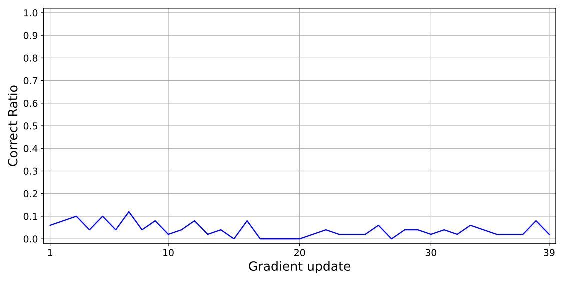
We used the same test set as in our first fine-tuning, comprising student submissions and manually corrupted codes. Likewise, we retained our original LoRA configuration. Resultingly, only the training set and the validation examples with correct code changed from our initial experiment.

7.3.6 Evaluation

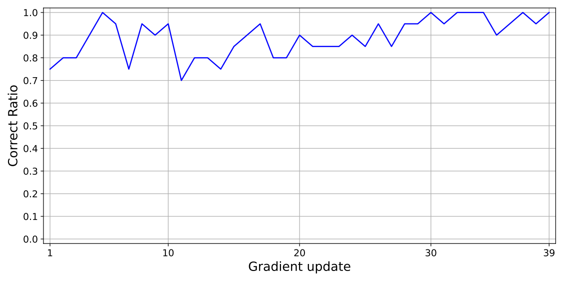
We performed the same evaluations as in our first fine-tuning experiment, with one change. When assessing the model trained on the erroneous lines dataset, we prompt it to list the line numbers containing errors instead of explaining them.

7.3.7 Results

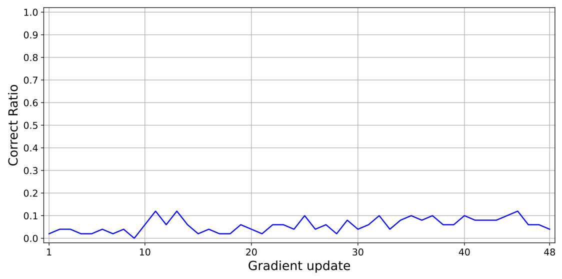
In this improved fine-tuning, we trained the model twice. We trained it once on the explanations dataset and once on the erroneous lines dataset. For each, we evaluated the model before fine-tuning, at the best checkpoint and at the final checkpoint. We then compared the outcomes of both methods and against our initial fine-tuning results.



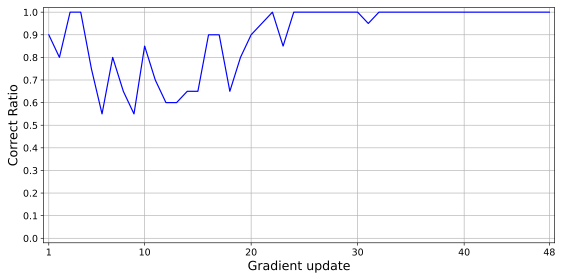
(a) Explanations: Correct examples



(b) Explanations: Incorrect examples



(c) Lines: Correct examples



(d) Lines: Incorrect examples

Figure 7.7. Improved fine-tuning validation results. The left plots show the evolution of the correct ratio on initially incorrect examples across the gradient updates. The right plots show the evolution of the correct ratio on initially correct examples. The top row shows the results from explanations set fine-tuning and the bottom row shows the results from lines set fine-tuning.

Training Set

Results on the training sets indicate successful fine-tuning. On the explanations dataset, the correct ratio rose from 53.18% before fine-tuning to 64.98% at the best checkpoint and 67.82% at the final checkpoint. On the erroneous lines dataset, it increased from 62.65% to 64.59% and then to 70.43%.

Validation Set

Validation results paint a different picture. Across the parameter updates, the correct ratio on initially incorrect examples remained barely exceeds 10% throughout training for the explanations dataset displayed in Figure 7.7a. The same pattern holds for the erroneous lines dataset shown in Figure 7.7c. This indicates little improvement in error detection and correction. Moreover, with the explanations dataset, the model never consistently recognizes correct code and leaves it unchanged, as revealed by Figure 7.7b.

Test Set: Student Codes

Table 7.1 shows that fine-tuning with our new datasets worsened correction performance. After training, the model produces more incorrect post-correction codes than before fine-tuning. Even worse, there are more incorrect codes after correction than before. Table 7.1b indicates the model still improves more of the initially incorrect codes than it degrades. However, that margin is much smaller than for the pre-fine-tuning model or after our first experiment. Both versions also show more cases of unchanged grades at the final checkpoint, indicating the model learns to leave correct code intact. Still, for the originally incorrect codes there are more grade decreases at the final checkpoint than at the best one. In other words, extended training leads to more harmful edits and overall the fine-tuning had a strongly negative impact on error detection and correction.

Table 7.2's confusion matrices assess the model's ability to label code as fully correct or not. The results are unsatisfactory. Notably, the best checkpoint of the explanations fine-tuning is the only model version that occasionally labels incorrect code as correct. This happens even though the prompt includes autograder errors, meaning we explicitly tell the model the autograder has detected problems. At the final checkpoint, it correctly identifies all faulty cases but only recognizes 11 correct codes as correct. The erroneous lines fine-tuning performs slightly better. At its best checkpoint, it never mislabels erroneous code and at its final checkpoint, it matches the performance of the last checkpoint from our first fine-tuning.

Qualitative Analysis

In addition to our quantitative analysis, we manually reviewed several notable LLM responses to student codes. We started with cases where the model at the best checkpoint of the explanations training mistakenly labeled faulty code as correct. In each instance, the model asserted the code was error-free and left it unchanged, despite the presence of actual errors.

We also examined instances where the initial code was correct but the LLM misclassified it as faulty and then broke it. In the explanations training, the best checkpoint produced 41 such cases and the final checkpoint 27. In the erroneous lines training, there were 36 at the best checkpoint and 15 at the end. In each case, the model detected errors that did not exist or were not detected by the autograder.

Furthermore, we reviewed cases where the model flagged correct code as faulty but its correction still earned a perfect score. In the explanations training, the best checkpoint produced 63 such instances and the final checkpoint 89. In the erroneous lines training, there were 92 at the best checkpoint and 35 at the end. Typically, the LLM noted minor issues, made slight edits and the code still passed all tests. Occasionally, it mentioned errors in its explanations but left the code unchanged.

Finally, we examined five student submissions that passed all autograder tests and obtained 9/9 but scored below 20/20 after human revision. This means some issues like complexity which is ignored in our restricted autograder, were not detected by the autograder. Across all four model versions, each identified and corrected the true problem in only one of these five cases. It was not the same case for every model. This suggests that the models' poor performance does not stem from autograder limitations but from their weakness in detecting errors.

Test Set: Manually Inserted Errors

Table 7.3 reports the results on our second test set with manually inserted errors. Models fine-tuned on the explanations dataset perform worse at detecting and fixing faults than both the pre-fine-tuning model and the model from our first fine-tuning. In contrast, the models trained on the erroneous lines dataset achieve the best correction results and their best checkpoints match the pre-fine-tuning model in error detection. These findings differ from those on student code, where these models performed poorly. This is likely because our hand crafted errors are simpler and more similar to those generated by the OpenAI models in our training sets than to the errors in student submissions.

Table 7.3 also reveals that these four models are far less likely to rewrite the entire program and instead attempt targeted fixes. In fact, at the final checkpoint for both training methods, none of the corrected examples involved extensive rewrites.

Genuinely Correcting

The results on our second test set showed that the LLM learned to apply targeted fixes instead of generating entirely new code. Table 7.5 summarizes the similarity between student submissions before and after correction obtained by the newly fine-tuned models. Both the average and median similarity increased and the average rank decreased compared to the pre-fine-tuning and initial fine-tuning models. Also, the final checkpoint outperformed the best checkpoint, indicating that full training further improved the model's ability to genuinely correct code. These results demonstrate that our new training data generation method consisting in introducing errors in correct code successfully taught the model to edit code in place. Nevertheless, the minimum similarity remains 0.00 and the maximum rank reaches 40. This indicates that the model still occasionally rewrites a student's entire submission.

7.3.8 Conclusion

In conclusion, we increased training code variability and created two genuine code correction datasets. However, fine-tuning on these data reduced the model's error detection and correction performance. This is evident from the lower correct ratio observed on student submissions. In fact, the model performed worse after this fine-tuning than it did initially and our first fine-tuning approach proved more effective.

Nonetheless, these results require careful interpretation. As Table 7.5 showed, the pre-fine-tuning model and the one from our first experiment often rewrote entire programs rather than correcting them. Their correct ratio reflects their ability to generate valid TP1 2023 solutions more than their error-detection skills. Consequently, it is inappropriate to compare those ratios with the metrics from our improved fine-tuning methods.

Limitations

Our fine-tuning process also has important limitations. To start with, we treated any code that scored 9/9 on our restricted autograder as correct but these tests are not exhaustive. In fact, of the 127 student submissions that earned a full autograder score, only 69 received 20/20 after human review. Nevertheless, we required an automated method to assess code correctness.

Moreover, expecting an LLM to master error detection and correction, a complex task, using such small, constrained datasets may have been overly optimistic. Still, TP1 2023 is fairly simple, with few valid implementations so enlarging the dataset would probably have made it redundant. Training directly on student submissions would be ideal but the fact that the assignments content changes every year makes that approach impractical.

It can also be noticed that the training set correct ratio remained fairly low at the final checkpoints. It attained 66.38% for the first fine-tuning and 67.82% and 70.43% for the subsequent attempts. Although the training correct ratio was still rising, the validation correct ratio had already plateaued for the first fine-tuning and showed no improvement for the other two. Hence, continuing training would risk overfitting the model to the training data.

Lastly, reinforcement learning fine-tuning for LLMs has become popular [39], especially since DeepSeek-R1's release [13]. It could have been an alternative approach. However, our work follows the framework of Jiang et al. [99], making supervised fine-tuning a more fitting choice.

Chapter 8

Conclusion

8.1 Conclusion

The objective of this thesis was to assess whether large language models (LLMs) can accelerate the code correction process in introductory programming courses. To explore this, we conducted three experiments that each examined a different aspect of the task.

The first, preliminary, experiment evaluated LLMs' ability to solve programming tasks from the course. This provided a baseline for their raw coding capabilities. GPT-4o often produced correct code, but its performance declined as task complexity increased—especially on TP1 and the 2024–2025 project. This limitation appeared even though GPT-4o was the strongest model available at the time. Nevertheless, asking for corrections consistently improved outcomes and using a well-structured prompt also helped. Hence, a student using these models could achieve high autograder scores on these tasks with minimal effort.

The second experiment investigated whether an LLM could grade student code. Using Qwen2.5-Coder-7B, we tested multiple prompting approaches and compared the model's predicted grades to human-assigned scores. The model was better than random chance at distinguishing fully correct from incorrect submissions, particularly on the TP, where its AUC values were highest. However, its numeric grade predictions were not reliable. Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) often exceeded those of a simple constant baseline grader. Among all tested prompts, only the vanilla criteria-based approach in French consistently surpassed the constant baseline across metrics. Nonetheless, it was still far from providing grades that could be used without human review.

The third experiment assessed the LLM's ability to detect and describe errors in student code. Here, we fine-tuned Qwen2.5-Coder-7B with LoRA with a focus on the TP1 2023 assignment, using two different training-set construction methods. Both datasets comprised prompt-response pairs. Each prompt contained the task and a possibly buggy solution. Each response explained the errors and provided a corrected version. To build the first set, we used Qwen to generate TP1 solutions and then explain and fix the incorrect ones it produced. To build the second one, OpenAI models wrote correct solutions and then introduced errors to circumvent Qwen's tendency to rewrite code instead of fixing it. Fine-tuning on the first set improved error detection and correction scores but often led the model to replace entire programs rather than truly correct them. Fine-tuning on the second reduced this behaviour but lowered overall performance in detecting and fixing errors.

Taken together, these experiments show that, while LLMs show promise, the models we evaluated are not reliable enough to detect and correct errors in student code on their own. This aligns with observations from recent research. The literature shows that LLMs can readily fix known issues, yet identifying errors independently remains a significant challenge [6, 90, 92]. In fact, even advanced commercial models augmented with debugging tools manage to resolve only around half of the bugs in complex benchmarks [90]. This highlights the gap between AI and human debugging [90]. In other words, current LLMs still miss many issues that a human developer would catch. For now, their most realistic use is to assist human graders rather than replace them. However, they struggled to detect errors reliably in our experiments so further improvements are also needed for them to be effective assistants. To conclude, current LLMs are not ready yet to act as error-detecting tools or autonomously grade students.

8.2 Limitations

Our results indicate that LLMs are not proficient enough to detect errors and grade code. However, this thesis exhibits some limitations. To start with, our choice of Qwen2.5-Coder-7B, although strong for its size, may not have been optimal. Due to time constraints, we relied on published benchmarks and a few manual tests. Additionally, reliable benchmarks that cover C programming and code editing for small models were hard to find. Also, large language models are advancing at an impressive pace. During the writing of this thesis, several models surpassed Qwen's coding performance and future releases will likely continue to improve model capabilities.

Another key limitation is the narrow scope of our evaluation. We tested LLMs on a small set of tasks from one course selected for its relevance to our use case. All tasks were in C, while many LLMs are known to perform best in Python [75, 76]. The results and conclusions might differ for other courses, other tasks or if the assignments were in Python.

Moreover, in our code-grading experiments, we used minimal prompt design. We asked the LLM to assign a numeric grade for each criterion without clear instructions or an explicit scale, which perhaps left too much room for interpretation. We also chose not to include the autograder's output, even though human graders usually have this information. Although prior work reports gains from few-shot prompting [118], we provided only one grading example to avoid overloading the prompt. This may have been a poor choice. In practice, adding that example worsened predictions, so we did not pursue further few-shot tests.

Finally, our error detection and correction experiments also display limitations. First, obtaining a 9/9 on our restricted autograder was sufficient for a code to be considered correct since we needed an automatic way to determine correctness. Still, these autograder tests are not exhaustive. Among 127 student submissions with a full autograder score, only 69 received 20/20 after human review. Moreover, given the small and narrow datasets we built, expecting an LLM to master error detection and correction may have been unrealistic.

8.3 Recommendations for Further Research

In this section, we make some suggestions for further research based on the limitations we discussed. To start with, we recommend reproducing these experiments with stronger and more recent models. This includes both larger commercial systems and the latest open-source models released after Qwen2.5-Coder. A targeted model selection phase with thorough tests would also probably help.

In addition, the scope should be broadened beyond a single course and language. Testing more assignments with different levels of difficulty and adding Python tasks would show whether the findings hold across contexts. Many LLMs perform best in Python [75, 76] so assessing over multiple languages would help separate model limits from language effects.

For grading, future work should invest more in prompt design. Prompts should define each criterion clearly, specify the grading scale and state what evidence from the code the model must cite. Also, including the autograder's results would better reflect what human graders have access to. Moreover, given reported gains from few-shot prompting [118], future studies should test prompts with several graded examples. These examples could have varying grades to show different levels of code quality to the LLM. Requiring a confidence score for each criterion may also help flag cases that need human review.

With respect to error detection and code correction, we recommend evaluating code with stricter and more complete methods. The autograder could include more hidden tests, edge cases and robustness checks and, if possible, correctness should not rely on the autograder alone. Frameworks such as "debug-gym" [90] provide another interesting avenue to explore. They let the model use basic debugging tools. It can compile the code, run tests, read stack traces, inspect files and propose small patches in short loops. In this setting, the model sees failing tests and logs, makes an edit and repeats until the tests pass. This feedback can improve error detection and raise correction success rates.

Lastly, fine-tuning would certainly benefit from a larger, more diverse training set. It could be built from a single task that allows different valid implementations or combine multiple tasks across difficulty levels. The data should include both correct and incorrect solutions each paired with clear explanations and a corrected version. Also, beyond supervised fine-tuning, reinforcement learning is a promising option [39]. After recent advances such as DeepSeek-R1 [13], testing RL or preference-based methods for code grading and correction is a logical next step.

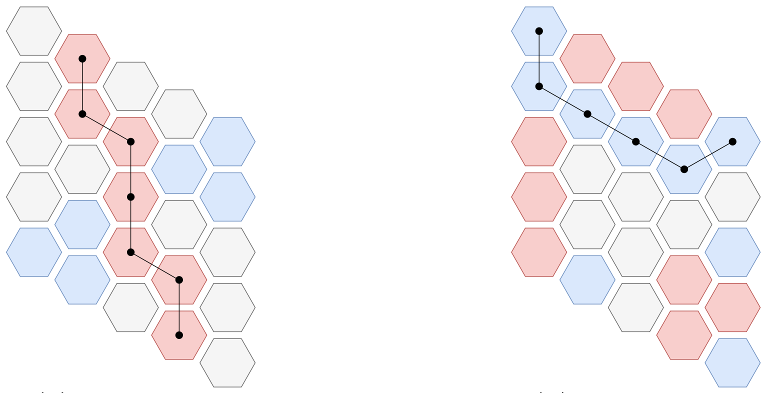
Appendix A

Tasks Description

A.1 Project 1 2023

In Project 1 from 2023, students implement HexGame, a game based on an $N \times N$ grid of hexagons for two players, red and blue. Taking turns, each player selects an unclaimed hexagon and assigns a colour to it. The game starts with a grid of empty hexagons and the red player is the first to play.

The aim is to finish with the most islands. An island is a group of adjacent hexagons sharing the same colour. Two hexagons belong to the same island if they are the same colour and can be reached by moving only through neighbouring hexagons of that color. Figure 3.1 shows two examples of grids. The game ends when no unclaimed hexagons remain or when a single island connects either the left and right borders or the top and bottom borders of the grid. Figure A.1 shows examples of games ending because of the second condition. Regarding the


(a) A red island connects the top and bottom borders

(b) A blue island connects the left and right borders

Figure A.1. Example of HexGame games ending. If an island connects the left and right or top and bottom borders of the grid, the game ends.

hexagons indexing in the grid, we use the matricial view. That is, the hexagon at the top-left corner of the grid has coordinates $(0, 0)$, the hexagon at the top-right corner has coordinates $(0, N - 1)$, the hexagon at the bottom-left corner has coordinates $(N - 1, 0)$, and the hexagon at the bottom-right corner has coordinates $(N - 1, N - 1)$.Figure A.2 shows the indexing of hexagons in a $n \times n$ grid. The file students have to submit is `hexgame.c`, in which students

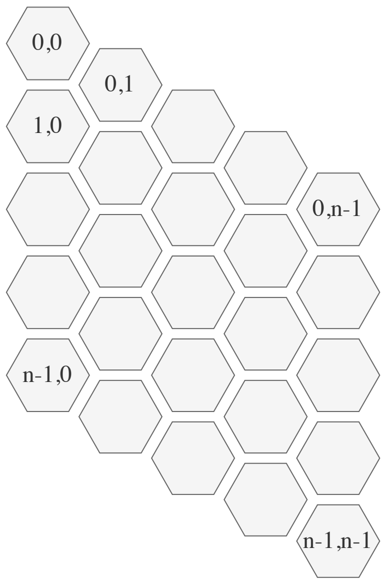


Figure A.2. Hexagon indexing in a $n \times n$ grid. The hexagon at the top-left corner has coordinates $(0, 0)$, the hexagon at the top-right corner has coordinates $(0, N - 1)$, the hexagon at the bottom-left corner has coordinates $(N - 1, 0)$ and the hexagon at the bottom-right corner has coordinates $(N - 1, N - 1)$.

must implement the game. It defines the HexGame structure and all functions declared in `hexgame.h`.

As already mentioned, the `hexgame.h` file declares the `HexGame` structure and all associated functions. It also defines a `Player` enumeration, which is used to represent the players. Based on this, `PLAYER_RED` and `PLAYER_BLUE` represent the two players. Moreover, `PLAYER_EMPTY` indicates an unclaimed hexagon and `PLAYER_ERROR` indicates an error occurred. Finally, `hexgame.h` defines a non-opaque `Move` structure, representing an action. It contains the player who made the move as well as the coordinates of the selected hexagon.

In total, there are eleven functions to define that enable gameplay. These functions are listed below.

- `HexGame *hexgameCreate(int board_size)`: Initializes a new instance of the `HexGame` structure, corresponding to a new game with a grid of size `board_size` × `board_size`.
- `void hexgameFree(HexGame *game)`: Frees the memory allocated to `game`.
- `int hexgameGetBoardSize(HexGame *game)`: Returns the size of the grid in `game`.
- `Player hexgameGetNextPlayer(HexGame *game)`: Returns the player who is about to play next in `game`. If the game is over, it returns `PLAYER_ERROR`.

- `int hexgameGetNumberOfIslands(HexGame *game, Player player)`: Returns the number of islands for the specified player in `game`.
- `bool hexgameIsValidMove(HexGame *game, Move move)`: Returns `true` if the move is valid in `game`, otherwise returns `false`. A move is valid if the hexagon is unclaimed and the player is the next to play.
- `void hexgamePlayMove(HexGame *game, Move move)`: Updates the game state for the specified move in `game`. If the move is invalid or the game has ended, it does nothing.
- `Player hexgameGetCellOwner(HexGame *game, int row, int col)`: Returns the owner of the hexagon at position (row, col) in `game`. If the hexagon is unclaimed, it returns `PLAYER_EMPTY`. If the position is invalid, it returns `PLAYER_ERROR`.
- `void hexgameSetCellOwner(HexGame *game, int row, int col, Player player)`: Sets the owner of the hexagon at position (row, col) in `game` to `player`. This function must only be used for debugging purposes and does not modify the next player to play.
- `bool hexgameIsGameOver(HexGame *game)`: Returns `true` if the game is over, otherwise returns `false`.
- `Player hexgameGetWinner(HexGame *game)`: Returns the winner of the game in `game` (the player having the most islands). If it is a tie, it returns `PLAYER_EMPTY`.

Finally, the instructions guide students through implementation. They are told to begin by identifying the fields for the `HexGame_t` structure. The next step is to implement the functions `hexgameCreate`, `hexgameFree`, `hexgameGetBoardSize` and `hexgameGetNextPlayer`. After that, they can focus on hexagon manipulation with the `hexgameGetCellOwner`, `hexgameSetCellOwner` and `hexgameIsValidMove` and `hexgamePlayMove`. The last part is more complex as it requires being able to isolate the various islands. The students are thus advised to first think about an algorithm allowing to isolate a single island. This should make them think about an exercise from the theoretical course and the practical sessions. Then, they should think about how to mark the different islands. Lastly, they can implement the `hexgameGetNumberOfIslands`, `hexgameIsGameOver` and `hexgameGetWinner` functions, potentially using auxiliary functions.

In addition to implementing all the functions, students must also comment the time complexity of their `hexgameGetNumberOfIslands` implementation in terms of N, the grid size.

```
### Task
Determine whether the student's code is correct or incorrect based on the instructions
and provided information. Focus on:
1. **Compliance**:  Check whether all requirements from the instructions are met
(structure, functions, expected behaviour).
2. **No errors**: Ensure there are no obvious mistakes in the code.

### Expected Format
1.  You must respond **only** in a standardized binary format:
- If the code fully meets the specification and has no detectable errors:
RESULT: [CORRECT]
- Otherwise:
RESULT: [INCORRECT]
2.  **No other content is allowed** in the response:  no justification, explanation or additional
commentary.Any additional content outside this format will be considered incorrect.

1.  Analyze the code, explaining any potential issues it contains.
2.  After analyzing the code, end your response with a conclusion in the **following format** (to be
strictly followed):
- If the code fully meets the specification and has no detectable errors:
RESULT: [CORRECT]
- Otherwise:
RESULT: [INCORRECT]

**Important**:
- Your analysis must come before the conclusion.
- The conclusion must always follow exactly the requested format ("RESULT: [CORRECT]" or "RESULT:
[INCORRECT]"). This is crucial for automated result extraction.
- Do not add any text after the conclusion.
```

Figure B.1. Explanation of the binary grading approach. This piece of text is inserted in the "Task" section of the prompt for the code grading experiments.
Note: The colours are used to distinguish between the versions without explanations and with explanations of the prompt. Also, we display an English translation but in practice this grading approach was only used in French.

Appendix B

Prompts

B.1 Code Grading

```
### Task
Evaluate the student's code based on the instructions and provided information. Focus on:
1. **Compliance**:  Check whether all requirements from the instructions are met
(structure, functions, expected behaviour).
2. **No errors**: Ensure there are no obvious mistakes in the code.

### Expected Format
1. Analyze the code, explaining any potential issues it contains.
2. After analyzing the code, end your response with a conclusion in the **following
format** (to be strictly followed):
Score: [x/10]
Where x is an integer between 0 and 10 corresponding to the code's quality according to
the following criteria:
- 10/10: The code is perfectly correct and fully meets the specification.
- 7-9/10: The code is generally correct, with a few minor errors or possible
optimizations.
- 4-6/10: The code has several moderate errors or notable violations of the
specifications.
- 0-3/10: The code contains serious errors or does not comply with the specification.

**Important**:
- Your analysis must come before the conclusion.
- The conclusion must always follow exactly the requested format (``Score: [x/10]'').
This is crucial for automated result extraction.
- Do not add any text after the conclusion.
```

Figure B.2. Explanation of the out-of-ten grading approach. This piece of text is inserted in the "Task" section of the prompt for the code grading experiments.
Note: We display an English translation but in practice this grading approach was only used in French.

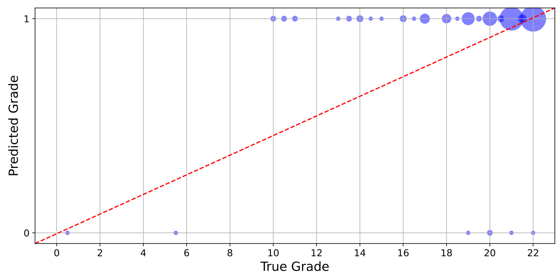
```
### Expected Evaluation Example
Below is, as an example, code submitted
by another student, along with the grades
and analyses I assigned it. This example
is provided solely as a reference for the
style and structure of the correction. Do
**not** correct this code.
1. Student code
[Student code]
2. Evaluation
[Evaluation]
```

(a) Additional content for grading example version

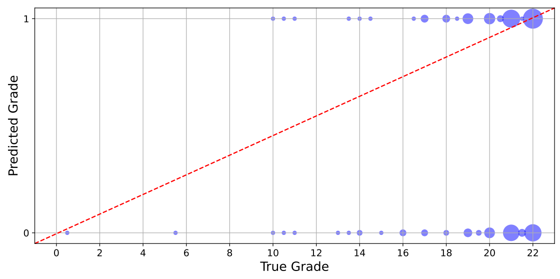
~~~
### Model Solution
**Important Note**: The model solution
presented below is an exemplary
implementation that adheres to the
assignment requirements and would receive
the maximum score of 22/22/20/20. However,
it is not the only valid solution. Any
solution that satisfies the instructions
and the evaluation criteria can also
receive full marks. Please keep this in
mind during evaluation to allow
flexibility for correct but alternative
approaches.
```[Model Solution]```
~~~

(b) Additional content for example solution version

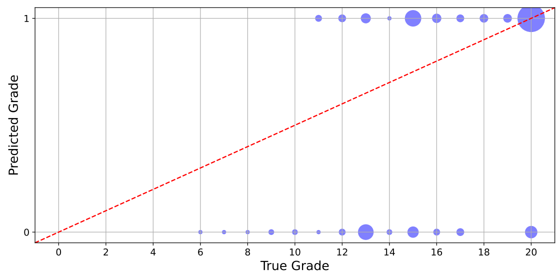
Figure B.3. Content to add to the enhanced version of the criteria-based prompt. The example solution is inserted between the "Instructions" and "Student code" sections of the prompt. The grading example is placed at the end of the prompt.
Note: The colours are used to distinguish between Project and TP instructions. Also, we display an English translation even though the grading example approach was only used in French in practice.



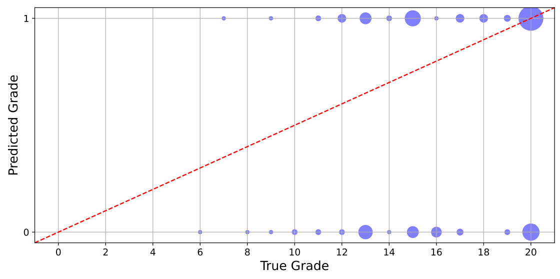
(a) Project, no explanations



(b) Project, explanations

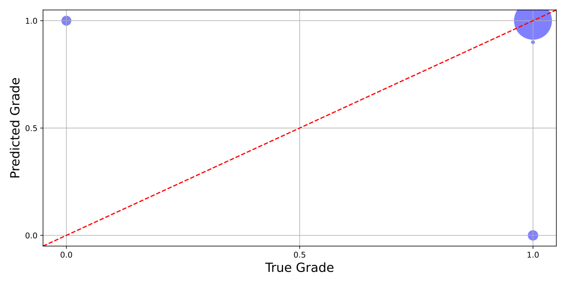


(c) TP, no explanations

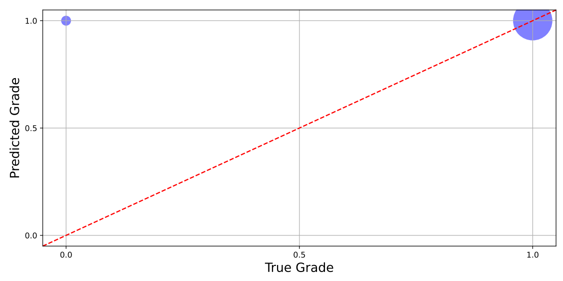


(d) TP, explanations

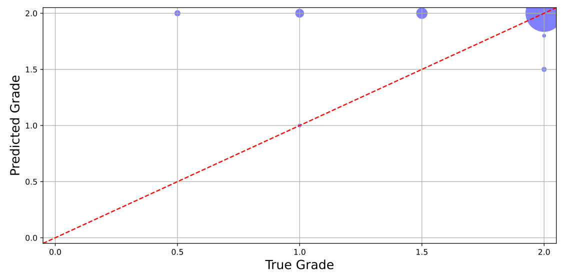
Figure C.1. Scatter plots for the binary grading approaches. They plot predicted grade against true grade. In the predicted grades, 1 designates a code that is fully correct and 0 a code that is not. The circle sizes are proportional to the number of examples at that position in the plot. The plots show that the model overestimates the codes correctness, especially when giving no explanations.



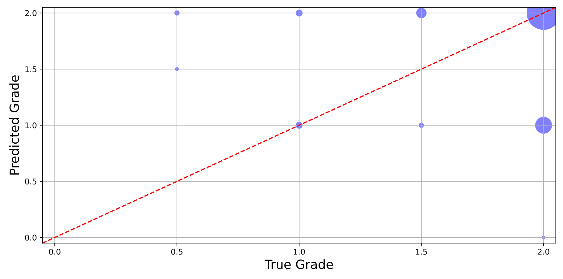
(a) Warning (French)



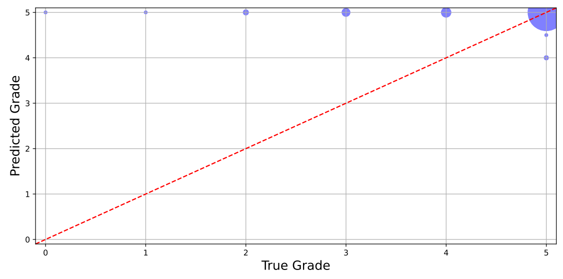
(b) Warning (English)



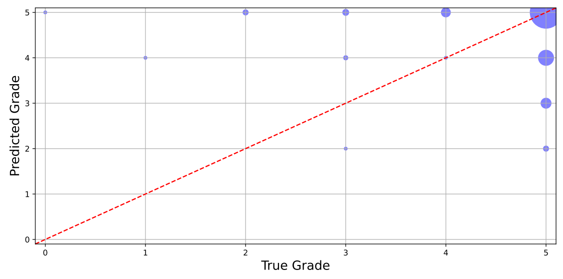
(c) hexgameCreate/Free (French)



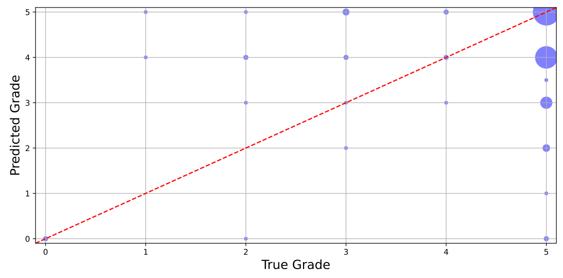
(d) hexgameCreate/Free (English)



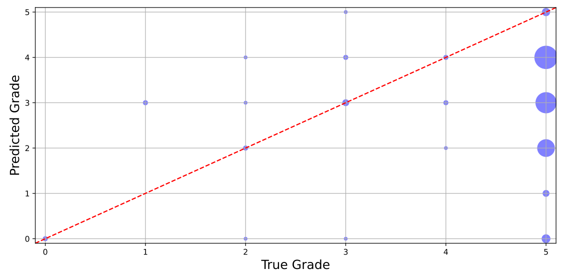
(e) Core functions (French)



(f) Core functions (English)



(g) hexgameGetNumberOfIslands (French)

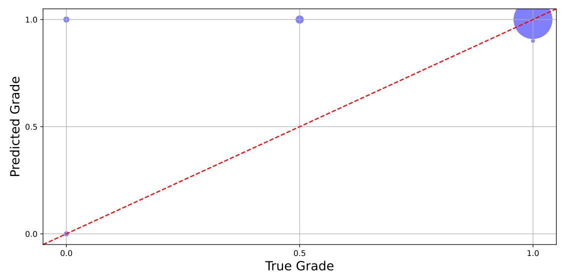


(h) hexgameGetNumberOfIslands (English)

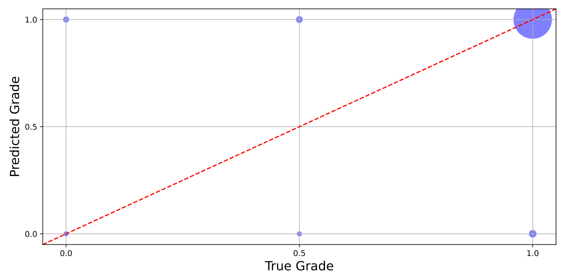
Figure C.2. Scatter plots per criterion for the vanilla criteria-based grading approach on the Project (part 1). They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot.

Appendix C

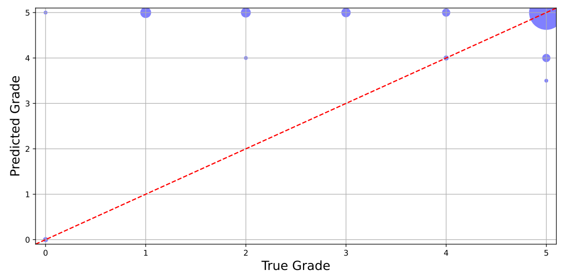
Figures



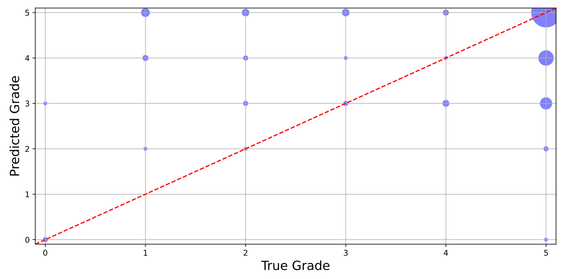
(i) hexgameGetWinner (French)



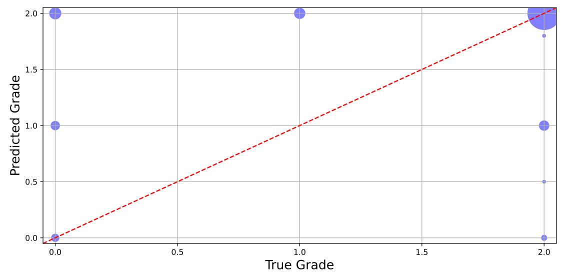
(j) hexgameGetWinner (English)



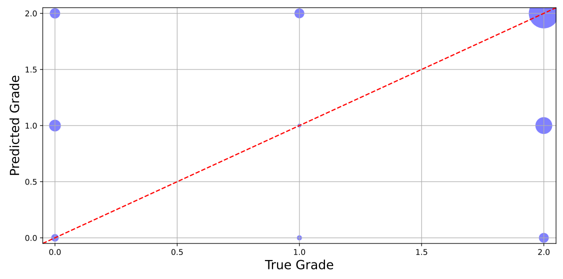
(k) hexgameIsOver (French)



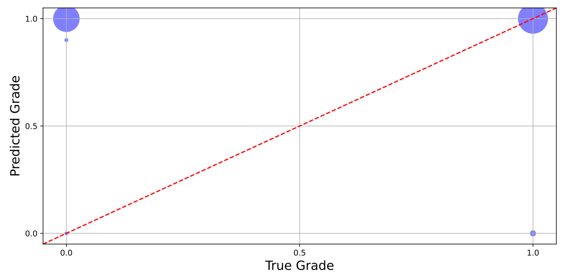
(l) hexgameIsOver (English)



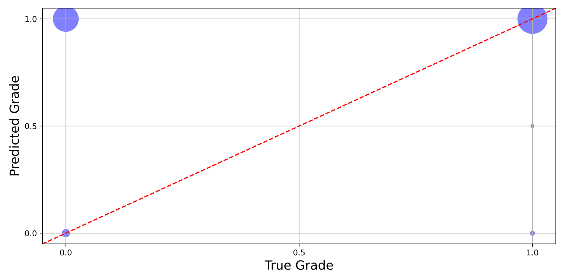
(m) Complexity (French)



(n) Complexity (English)

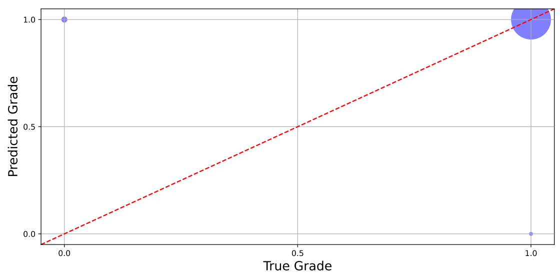


(o) Memory leaks (French)

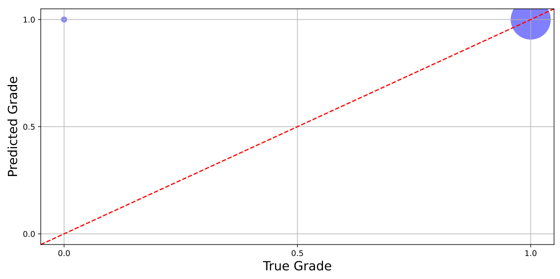


(p) Memory leaks (English)

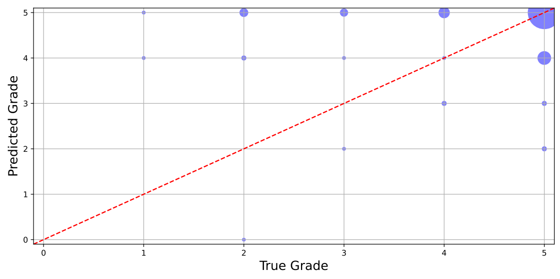
Figure C.2. Scatter plots per criterion for the vanilla criteria-based grading approach on the Project (part 2). They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot.



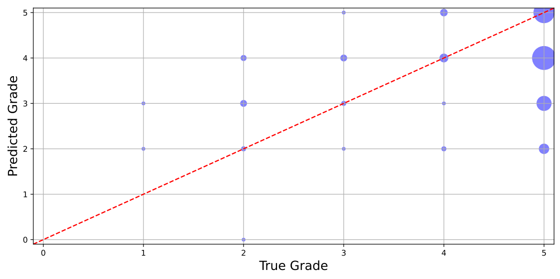
(a) Warning (French)



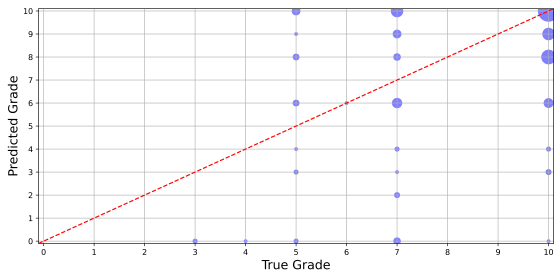
(b) Warning (English)



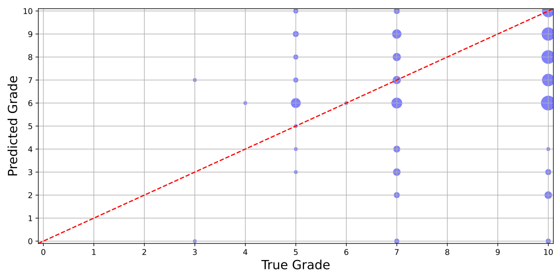
(c) isPeak (French)



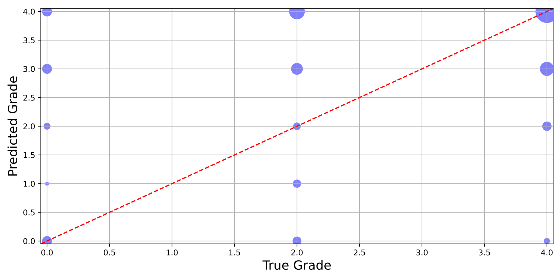
(d) isPeak (English)



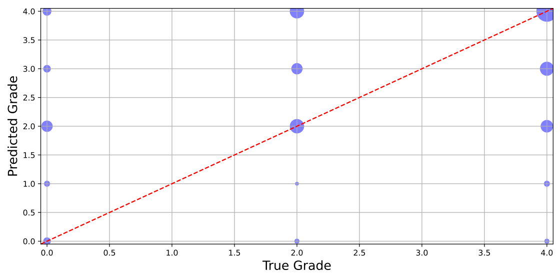
(e) findPeak2D (French)



(f) findPeak2D (English)

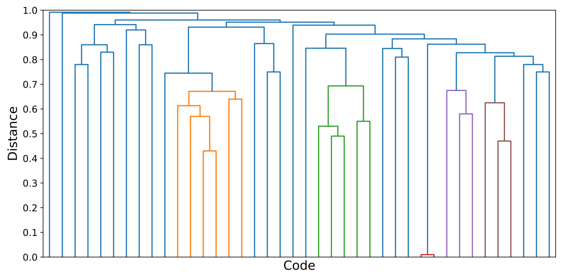


(g) Complexity (French)

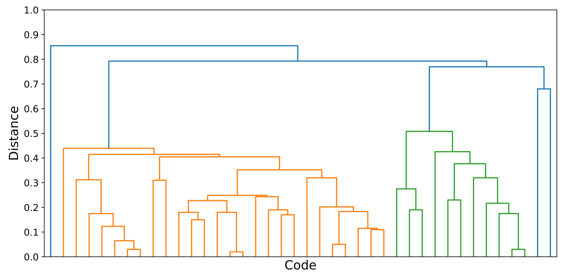


(h) Memory leaks (English)

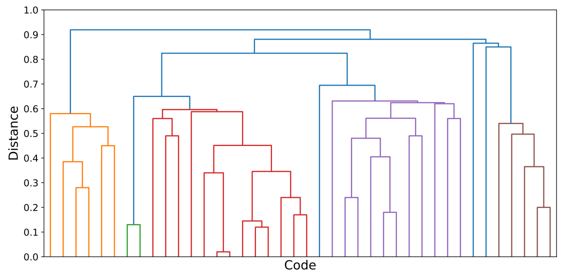
Figure C.3. Scatter plots per criterion for the vanilla criteria-based grading approach on the TP (part 1). They plot predicted grade against true grade. The circle sizes are proportional to the number of examples at that position in the plot.



(a) Student codes



(b) Initial Training set



(c) Improved Training set

Figure C.4. Hierarchical clustering: training sets vs student codes. These clusterings are obtained by starting with each code as its own cluster and then repeatedly merging the two clusters whose average pairwise distance is smallest [161]. Distance is defined as $1 - \text{similarity}$. This means that the lower two branches meet, the more similar the codes are.

Appendix D

Tables

D.1 Code Grading Results

D.2 Error Detection Evaluation

Table D.1. Performance metrics on the project for each criterion. **Bolded values** represent the best score per criterion in each column. The LLM with the French prompt performs worse than the constant baseline on six of eight criteria. The English prompt leads to worse results in every criterion except Warning.

Criterion	**Language**	**RMSE**	**MAE**
Warning	FR	0.33	0.11
	EN	**0.23**	**0.05**
Constant baseline	–	**0.23**	**0.05**
hexgameCreate/Free	FR	**0.32**	**0.11**
	EN	0.51	0.26
Constant baseline	–	**0.32**	**0.11**
Core functions	FR	0.80	0.26
	EN	1.04	0.55
Constant baseline	–	**0.75**	**0.25**
hexgameGetNumberOfIslands	FR	1.30	0.83
	EN	2.16	1.82
Constant baseline	–	**0.93**	**0.32**
hexgameGetWinner	FR	**0.16**	**0.04**
	EN	0.24	0.07
Constant baseline	–	0.19	0.05
hexgameIsOver	FR	1.40	**0.63**
	EN	1.44	0.84
Constant baseline	–	**1.34**	0.67
Complexity	FR	0.77	**0.39**
	EN	0.87	0.53
Constant baseline	–	**0.76**	0.40
Memory leaks	FR	0.67	0.45
	EN	0.64	**0.42**
Constant baseline	–	**0.50**	0.44

Table D.2. Performance metrics on the TP for each criterion. **Bolded values** represent the best score per criterion in each column. The French prompt approach falls below the constant baseline on three of four criteria. The English approach surpasses the French on two criteria. Still, it is below the baseline on the two remaining criteria.

Criterion	Language	RMSE	MAE
Warning	FR	0.15	**0.02**
	EN	**0.13**	**0.02**
Constant baseline	–	**0.13**	**0.02**
isPeak	FR	1.04	0.53
	EN	1.30	0.97
Constant baseline	–	**0.91**	**0.40**
findPeak2D	FR	2.88	2.05
	EN	3.26	2.52
Constant baseline	–	**1.98**	**1.61**
Complexity	FR	1.68	1.17
	EN	1.62	**1.14**
Constant baseline	–	**1.53**	1.36

Table D.3. Errors introduced manually in test set.

Original Code	**Error introduced**
Reference solution 1	In `isPeak`, we turned all the "$>$" and "$<$" operators into "$\geq$" and "$\leq$" operators.
Reference solution 2	In `isPeak`, we turned all the "$>$" and "$<$" operators into "$\geq$" and "$\leq$" operators.
Reference solution 1	In `isPeak`, instead of `i< n-1` and `j< n-1`, we wrote `i< n` and `j< n`.
Reference solution 2	In `isPeak`, instead of `i< n-1` and `j< n-1`, we wrote `i< n` and `j< n`.
Reference solution 1	In `isPeak`, we removed the boundary checks: `i>0`, `j>0`, `i< n-1`, `j< n-1`.
Reference solution 2	In `isPeak`, we removed the boundary checks: `i>0`, `j>0`, `i< n-1`, `j< n-1`.
Reference solution 1	In `findPeak2Drec`, we flipped the comparison from `if (M[rowmid][j] > vmax)` to `if (M[rowmid][j] < vmax)`.
Reference solution 1	In the `findPeak2Drec` recursive calls, we replaced `rowmid-1` and `rowmid+1` by `rowmid`.
Reference solution 2	In `findPeak2D_fast_rec`, at lines 74 and 90 we changed `if (M[i][j] > vmax)` to `if (M[i][j] < vmax)`.
Reference solution 2	In `findPeak2D_fast_rec`, at lines 116–118, we turned the `==` comparators into = signs.
Student submission 1	In `findPeak2D`, at line 36 we changed `mid = left + (right-left)/2` into `mid = left + (left-right)/2`.
Student submission 2	In `isPeak`, we changed all the `||`'s into `&&`'s in the `if` conditions.
Student submission 3	We modified the auxiliary `greatest_in_col` function by swapping 1st and 2nd coordinates, such that the function now finds the greatest in row instead of column.
Student submission 4	In the `findPeak2Drecursive` calls, we replaced `jMid+1` and `jMid-1` by `jMid`.
Student submission 5	In `findPeak2D`, we swapped the `j += dist/2` and `j -= dist/2`.
Student submission 6	In `findPeak2D`, we swapped `res[0] = imax` and `res[1] = j`.
Student submission 7	In `isPeak`, we turned all `INT_MIN`'s into `INT_MAX`'s.
Student submission 8	In `findPeak1`, at line 33 we wrote `int a = (end-start)/2` instead of `int a = (end + start)/2`.
Student submission 9	In `isPeak`, we turned the `<`'s of `i+1< n` and `j+1< n` into `<=`'s.
Student submission 10	We removed the `#include <limits.h>`, making compilation fail.
LLM generation 1	In `findPeak2D` at line 28, we swapped the first and second coordinates, going from `if (M[mid][i] > M[mid][maxIdx])` to `if (M[i][mid] > M[maxIdx][mid])`.
LLM generation 2	In `findPeak2D`, we replaced the $\leq$ of `while (low <= high)` by `<`.
LLM generation 3	In `findPeakUtil` at line 32 of the auxiliary call, we replaced the argument `(mid+high+1)/2` by `(mid/high)/2`.
LLM generation 4	In `isPeak`, we turned all the `||`'s into `&&`'s.
LLM generation 5	In `isPeak`, we turned all the $\geq$'s into `>`'s.

Bibliography

[1] Cem Dilmegani and Mert Palazoğlu. "The Future of Large Language Models in 2025". https://research.aimultiple.com/future-of-large-language-models/. Accessed: 2025-06-04. May 2025 (pages 1, 3).

[2] Shervin Minaee et al. "Large Language Models: A Survey". In: (2025). arXiv: 2402.06196 [cs.CL]. URL: https://arxiv.org/abs/2402.06196 (pages 1, 3, 4).

[3] Nefi Alarcon. "OpenAI Presents GPT-3, a 175 Billion Parameters Language Model". https://developer.nvidia.com/blog/openai-presents-gpt-3-a-175-billion-parameters-language-model/. Accessed: 2025-06-04. NVIDIA, July 2020 (pages 1, 3).

[4] OpenAI. "GPT-4". OpenAI. Mar. 14, 2023. URL: https://openai.com/index/gpt-4-research/ (visited on 08/08/2025) (page 1).

[5] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021) (pages 1, 7, 8, 12, 19, 32, 33).

[6] Runchu Tian et al. "DebugBench: Evaluating Debugging Capability of Large Language Models". 2024. arXiv: 2401.04621 [cs.SE]. URL: https://arxiv.org/abs/2401.04621 (pages 1, 11, 24, 75).

[7] Shen Wang et al. "Large Language Models for Education: A Survey and Outlook". 2024. arXiv: 2403.18105 [cs.CL]. URL: https://arxiv.org/abs/2403.18105 (pages 1, 14).

[8] Enkelejda Kasneci et al. "ChatGPT for good? On opportunities and challenges of large language models for education". In: *Learning and individual differences* 103 (2023), p. 102274 (pages 1, 14).

[9] Junaid Qadir. "Engineering education in the era of ChatGPT: Promise and pitfalls of generative AI for education". In: *2023 IEEE global engineering education conference (EDUCON)*. IEEE. 2023, pp. 1–9 (pages 1, 14).

[10] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". 2024. arXiv: 2406.00515 [cs.CL]. URL: https://arxiv.org/abs/2406.00515 (pages 1, 8).

[11] Deborah Etsenake and Meiyappan Nagappan. "Understanding the Human-LLM Dynamic: A Literature Survey of LLM Use in Programming Tasks". 2024. arXiv: 2410.01026 [cs.SE]. URL: https://arxiv.org/abs/2410.01026 (pages 1, 7).

[12] OpenAI. “Introducing OpenAI o3 and o4-mini”. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-06-04. OpenAI, Apr. 2025 (pages 1, 4, 6, 7, 10–12, 15, 64, 65).

[13] Daya Guo et al. “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning”. In: *arXiv preprint arXiv:2501.12948* (2025) (pages 1, 5–7, 10–12, 14, 18, 19, 73, 76).

[14] GitHub. “GitHub Copilot: Your AI pair programmer”. https://github.com/features/copilot. Accessed: 2025-06-19; launch announced June 29, 2021 (technical preview) :contentReference[oaicite:8]index=8 (pages 1, 11, 27).

[15] Stack Overflow. “Technology: 2024 Stack Overflow Developer Survey”. https://survey.stackoverflow.co/2024/technology. Survey fielded May 19–June 20, 2024; results released July 24, 2024; accessed 2025-06-25. July 2024 (pages 1, 11, 24, 28).

[16] Sida Peng et al. “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot”. 2023. arXiv: 2302.06590 [cs.SE]. URL: https://arxiv.org/abs/2302.06590 (page 1).

[17] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023) (pages 1, 7, 9, 19).

[18] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017) (page 3).

[19] Jared Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020) (page 3).

[20] Deval Shah. “Beyond Bigger Models: The Evolution of Language Model Scaling Laws”. https://medium.com/@aiml_58187/beyond-bigger-models-the-evolution-of-language-model-scaling-laws-d4bc974d3876. Accessed: 2025-06-04. Australian Institute for Machine Learning (AIML), Oct. 2024 (pages 3, 4).

[21] Jordan Hoffmann et al. “Training Compute-Optimal Large Language Models”. 2022. arXiv: 2203.15556 [cs.CL]. URL: https://arxiv.org/abs/2203.15556 (pages 3, 4).

[22] W Zaremba et al. “Trading inference-time compute for adversarial robustness, 2025”. In: *URL https://cdn. openai. com/papers/trading-inference-time-compute-for-adversarial-robustness-20250121_1. pdf* () (pages 4–6, 19).

[23] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837 (pages 4, 6, 17, 19).

[24] OpenAI. “Hello GPT-4o”. https://openai.com/index/hello-gpt-4o/. Accessed: 2025-06-05. OpenAI, May 2024 (pages 4, 15, 24, 26).

[25] OpenAI. “Model Release Notes”. https://help.openai.com/en/articles/9624314-model-release-notes. Accessed: 2025-06-05 (page 4).

[26] OpenAI. “Introducing GPT-4.1 in the API”. https://openai.com/index/gpt-4-1/. Accessed: 2025-06-05. OpenAI, Apr. 2025 (pages 4, 66).

[27] Dave Bergmann. “What is a context window?” IBM. Nov. 7, 2024. URL: https://www.ibm.com/think/topics/context-window (visited on 07/11/2025) (page 4).

[28] Anthropic. “Introducing the next generation of Claude”. https://www.anthropic.com/news/claude-3-family. Accessed: 2025-06-05. Anthropic, Mar. 2024 (page 4).

[29] Anthropic. “Introducing Claude 4”. https://www.anthropic.com/news/claude-4. Accessed: 2025-06-05. Anthropic PBC, May 2025 (page 4).

[30] Demis Hassabis. “Our next-generation model: Gemini 1.5”. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/. Accessed: 2025-06-05. Google DeepMind, Feb. 2024 (pages 4, 9, 12).

[31] Tulsee Doshi. “Gemini 2.5: Our Most Intelligent Models Are Getting Even Better”. https://blog.google/technology/google-deepmind/google-gemini-updates-io-2025/. Accessed: 2025-06-05. Google DeepMind, May 2025 (pages 4, 6).

[32] Meta. “Introducing Meta Llama 3: The most capable openly available LLM to date”. https://ai.meta.com/blog/meta-llama-3/. Accessed: 2025-06-05. Meta AI, Apr. 2024 (page 4).

[33] Mistral AI. “Models Overview”. https://docs.mistral.ai/getting-started/models/models_overview/. Accessed: 2025-06-05. Mistral AI, 2025 (page 4).

[34] Mistral AI team. “Large Enough”. https://mistral.ai/news/mistral-large-2407. Accessed: 2025-06-05. Mistral AI, July 2024 (page 4).

[35] Mistral AI. “Medium is the new large”. https://mistral.ai/news/mistral-medium-3. Accessed: 2025-06-05. Mistral AI, May 2025 (page 4).

[36] xAI. “Announcing Grok-1.5”. https://x.ai/news/grok-1.5. Accessed: 2025-06-05. xAI, Mar. 2024 (page 4).

[37] xAI. “Grok-1.5 Vision Preview”. https://x.ai/news/grok-1.5v. Accessed: 2025-06-05. xAI, Apr. 2024 (page 4).

[38] xAI. “xAI”. https://x.ai/. Accessed: 2025-06-05. xAI (page 4).

[39] Hakim Sidahmed et al. “Parameter Efficient Reinforcement Learning from Human Feedback”. 2024. arXiv: 2403.10704 [cs.LG]. URL: https://arxiv.org/abs/2403.10704 (pages 5, 73, 76).

[40] Wenhao Liu et al. “Aligning Large Language Models with Human Preferences through Representation Engineering”. 2024. arXiv: 2312.15997 [cs.CL]. URL: https://arxiv.org/abs/2312.15997 (page 5).

[41] Mehdi Khamassi, Marceau Nahon, and Raja Chatila. “Strong and weak alignment of large language models with human values”. In: *Scientific Reports* 14.1 (2024), p. 19399 (page 5).

[42] Yichen Gong et al. “Safety Misalignment Against Large Language Models”. In: *Proceedings 2025 Network and Distributed System Security Symposium*. 2025 (page 5).

[43] Aneesh Pappu et al. “Measuring memorization in RLHF for code completion”. 2024. arXiv: 2406.11715 [cs.LG]. URL: https://arxiv.org/abs/2406.11715 (page 5).

[44] Long Ouyang et al. "Training language models to follow instructions with human feedback". In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744 (page 5).

[45] Harrison Lee et al. "RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback". 2024. arXiv: 2309.00267 [cs.CL]. URL: https://arxiv.org/abs/2309.00267 (page 5).

[46] Yuntao Bai et al. "Constitutional ai: Harmlessness from ai feedback". In: *arXiv preprint arXiv:2212.08073* (2022) (page 5).

[47] Anthropic. "Claude's Constitution". https://www.anthropic.com/news/claudes-constitution. Accessed: 2025-06-05. Anthropic, May 2023 (pages 5, 18).

[48] Chuanpeng Yang et al. "Survey on knowledge distillation for large language models: methods, evaluation, and application". In: *ACM Transactions on Intelligent Systems and Technology* (2024) (page 5).

[49] Xunyu Zhu et al. "A survey on model compression for large language models". In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 1556–1577 (page 5).

[50] Emmanuel Ohiri and Richard Poole. "What is the cost of training large language models?" https://www.cudocompute.com/blog/what-is-the-cost-of-training-large-language-models. Accessed: 2025-06-05. CUDO Compute, May 2025 (page 5).

[51] Yuxian Gu et al. "MiniLLM: Knowledge Distillation of Large Language Models". 2024. arXiv: 2306.08543 [cs.CL]. URL: https://arxiv.org/abs/2306.08543 (page 5).

[52] Rohan Taori et al. "Alpaca: A strong, replicable instruction-following model". In: *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3.6 (2023), p. 7 (pages 5, 6, 18).

[53] Wei-Lin Chiang et al. "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality". https://lmsys.org/blog/2023-03-30-vicuna/. Accessed: 2025-06-05. Mar. 2023 (pages 5, 6).

[54] Weilin Cai et al. "A survey on mixture of experts in large language models". In: *IEEE Transactions on Knowledge and Data Engineering* (2025) (page 6).

[55] Aske Plaat et al. "Reasoning with Large Language Models, a Survey". 2024. arXiv: 2407.11511 [cs.AI]. URL: https://arxiv.org/abs/2407.11511 (page 6).

[56] Xuezhi Wang et al. "Self-Consistency Improves Chain of Thought Reasoning in Language Models". 2023. arXiv: 2203.11171 [cs.CL]. URL: https://arxiv.org/abs/2203.11171 (pages 6, 17).

[57] Jia Li et al. "Structured chain-of-thought prompting for code generation". In: *ACM Transactions on Software Engineering and Methodology* 34.2 (2025), pp. 1–23 (page 6).

[58] Shunyu Yao et al. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models". 2023. arXiv: 2305.10601 [cs.CL]. URL: https://arxiv.org/abs/2305.10601 (page 6).

[59] Anthropic. “Claude 3.7 Sonnet and Claude Code”. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2025-06-05. Anthropic PBC, Feb. 2025 (page 6).

[60] Ansong Ni et al. “NExT: Teaching Large Language Models to Reason about Code Execution”. 2024. arXiv: 2404.14662 [cs.LG]. URL: https://arxiv.org/abs/2404.14662 (page 6).

[61] Che Jiang et al. “On Large Language Models’ Hallucination with Regard to Known Facts”. 2024. arXiv: 2403.20009 [cs.CL]. URL: https://arxiv.org/abs/2403.20009 (page 6).

[62] Yuqi Zhu et al. “Uncertainty-guided chain-of-thought for code generation with llms”. In: *arXiv preprint arXiv:2503.15341* (2025) (page 6).

[63] OpenAI. “Introducing GPT-4.5”. https://openai.com/index/introducing-gpt-4-5/. Accessed: 2025-06-05. OpenAI, Feb. 2025 (page 6).

[64] Mistral AI team. “Codestral”. https://mistral.ai/fr/news/codestral. Accessed: 2025-06-19. May 2024 (pages 7, 9, 10, 32, 33).

[65] Binyuan Hui et al. “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186* (2024) (pages 7, 9, 10, 12, 17, 18, 32, 34, 50).

[66] Chengyuan Liu et al. “More Than Catastrophic Forgetting: Integrating General Capabilities For Domain-Specific LLMs”. 2024. arXiv: 2405.17830 [cs.CL]. URL: https://arxiv.org/abs/2405.17830 (page 7).

[67] Suhas Kotha, Jacob Mitchell Springer, and Aditi Raghunathan. “Understanding Catastrophic Forgetting in Language Models via Implicit Inference”. 2024. arXiv: 2309.10105 [cs.CL]. URL: https://arxiv.org/abs/2309.10105 (page 7).

[68] Cody Blakeney et al. “Does your data spark joy? Performance gains from domain upsampling at the end of training”. 2024. arXiv: 2406.03476 [cs.LG]. URL: https://arxiv.org/abs/2406.03476 (page 7).

[69] OpenAI. “GPT-4o System Card”. https://openai.com/index/gpt-4o-system-card/. Accessed: 2025-06-19. Aug. 2024 (pages 7–10).

[70] Anthropic. “Claude 3.5 Sonnet”. https://www.anthropic.com/news/claude-3-5-sonnet. Accessed: 2025-06-19. June 2024 (pages 7–9).

[71] Qihao Zhu et al. “Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence”. In: *arXiv preprint arXiv:2406.11931* (2024) (pages 7, 9, 10, 12, 14, 17, 18, 32).

[72] Meta AI Team. “Introducing Llama 3.1: Our most capable models to date”. https://ai.meta.com/blog/meta-llama-3-1/. Accessed: 2025-06-19. July 2024 (pages 8–10, 32).

[73] Jacob Austin et al. “Program Synthesis with Large Language Models”. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732 (pages 8, 12).

[74] Mistral AI Team. “Large Enough”. https://mistral.ai/fr/news/mistral-large-2407. Accessed: 2025-06-19. July 2024 (pages 8, 9).

[75] Mohammed Latif Siddiq et al. "The Fault in our Stars: Quality Assessment of Code Generation Benchmarks". In: *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2024, pp. 201–212 (pages 8, 75, 76).

[76] GitHub Staff. "Octoverse: AI leads Python to top language as the number of global developers surges". https://github.blog/news-insights/octoverse/octoverse-2024/. Accessed: 2025-06-19. Oct. 2024 (pages 8, 75, 76).

[77] Federico Cassano et al. "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation". 2022. arXiv: 2208.08227 [cs.LG]. URL: https://arxiv.org/abs/2208.08227 (page 8).

[78] Qinkai Zheng et al. "Codegeex: A pre-trained model for code generation with multilingual benchmarking on Humaneval-X". In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 5673–5684 (page 8).

[79] Anthropic. "Claude 3.5 Haiku". https://www.anthropic.com/claude/haiku. Accessed: 2025-06-19. Oct. 2024 (pages 9, 10).

[80] Gemini Team et al. "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context". In: *arXiv preprint arXiv:2403.05530* (2024) (pages 9, 10).

[81] Mistral AI team. "Un Ministral, des Ministraux". https://mistral.ai/news/ministraux. Accessed: 2025-06-19. Oct. 2024 (pages 9, 10, 32).

[82] Qwen Team. "Qwen2.5-Coder Series: Powerful, Diverse, Practical." https://qwenlm.github.io/blog/qwen2.5-coder-family/. Accessed: 2025-06-19. Nov. 2024 (page 9).

[83] DeepSeek-AI et al. "DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence". https://github.com/deepseek-ai/DeepSeek-Coder-V2. Also available as arXiv preprint: https://arxiv.org/abs/2406.11931; Accessed: 2025-06-19. June 2024 (page 10).

[84] Meta AI Team. "Llama 3.2: Revolutionizing edge AI and vision with open models". https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed: 2025-06-19. Sept. 2024 (pages 10, 32).

[85] Qwen Team. "QwQ-32B: Embracing the Power of Reinforcement Learning". https://qwenlm.github.io/blog/qwq-32b/. Accessed: 2025-06-19. Mar. 2025 (pages 10, 11).

[86] Naman Jain et al. "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code". 2024. arXiv: 2403.07974 [cs.SE]. URL: https://arxiv.org/abs/2403.07974 (pages 10, 11, 33, 64).

[87] Shanghaoran Quan et al. "CodeElo: Benchmarking Competition-level Code Generation of LLMs with Human-comparable Elo Ratings". 2025. arXiv: 2501.01257 [cs.CL]. URL: https://arxiv.org/abs/2501.01257 (pages 11, 64).

[88] Thomas Dohmke. "GitHub Copilot X: The AI-powered developer experience". https://github.blog/news-insights/product-news/github-copilot-x-the-ai-powered-developer-experience/. Updated: May 21, 2024; Accessed: 2025-06-19. Mar. 2023 (page 11).

[89] Mehmet Akhoroz and Caglar Yildirim. "Conversational AI as a Coding Assistant: Understanding Programmers' Interactions with and Expectations from Large Language Models for Coding". 2025. arXiv: 2503.16508 [cs.HC]. URL: https://arxiv.org/abs/2503.16508 (page 11).

[90] Xingdi Yuan et al. "debug-gym: A Text-Based Environment for Interactive Debugging". 2025. arXiv: 2503.21557 [cs.AI]. URL: https://arxiv.org/abs/2503.21557 (pages 11, 13, 14, 75, 76).

[91] Yacine Majdoub and Eya Ben Charrada. "Debugging with open-source large language models: An evaluation". In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2024, pp. 510–516 (page 11).

[92] Gladys Tyen et al. "LLMs cannot find reasoning errors, but can correct them given the error location". 2024. arXiv: 2311.08516 [cs.AI]. URL: https://arxiv.org/abs/2311.08516 (pages 11–14, 75).

[93] Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". 2021. arXiv: 2102.04664 [cs.SE]. URL: https://arxiv.org/abs/2102.04664 (pages 12, 14).

[94] Weixiang Yan et al. "CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation". 2024. arXiv: 2311.08588 [cs.CL]. URL: https://arxiv.org/abs/2311.08588 (page 12).

[95] Aider. "o1 tops aider's new polyglot leaderboard". https://aider.chat/2024/12/21/polyglot.html#the-polyglot-benchmark. Accessed: 2025-06-20. Dec. 2024 (pages 12, 32–34).

[96] Carlos E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770 (pages 12, 14, 64).

[97] OpenAI. "Introducing OpenAI o1". https://openai.com/o1/. Accessed: 2025-06-20. Sept. 2024 (page 12).

[98] Tuan Dinh et al. "Large language models of code fail at completing code with potential bugs". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 41386–41412 (page 12).

[99] Nan Jiang et al. "LeDex: Training LLMs to Better Self-Debug and Explain Code". 2025. arXiv: 2405.18649 [cs.CL]. URL: https://arxiv.org/abs/2405.18649 (pages 13, 14, 51, 56, 73).

[100] Marcus Messer et al. "Automated grading and feedback tools for programming education: A systematic review". In: *ACM Transactions on Computing Education* 24.1 (2024), pp. 1–43 (pages 14, 15).

[101] Stefan Haller et al. "Survey on Automated Short Answer Grading with Deep Learning: from Word Embeddings to Transformers". 2022. arXiv: 2204.03503 [cs.CL]. URL: https://arxiv.org/abs/2204.03503 (page 14).

[102] Yunlong Pan and Ross H Nehm. "Large Language Model and Traditional Machine Learning Scoring of Evolutionary Explanations: Benefits and Drawbacks". In: *Education Sciences* 15.6 (2025), p. 676 (pages 14, 15).

[103] Da-Wei Zhang et al. "Evaluating large language models for criterion-based grading from agreement to consistency". In: *npj Science of Learning* 9.1 (2024), p. 79 (page 15).

[104] Cheng-Han Chiang et al. "Large Language Model as an Assignment Evaluator: Insights, Feedback, and Challenges in a 1000+ Student Course". 2024. arXiv: 2407.05216 [cs.CL]. URL: https://arxiv.org/abs/2407.05216 (pages 15–17).

[105] Paraskevas Lagakis, Stavros Demetriadis, and Georgios Psathas. "Automated grading in coding exercises using large language models". In: *Interactive Mobile Communication, Technologies and Learning*. Springer, 2023, pp. 363–373 (pages 15, 17).

[106] Mina Yousef et al. "BeGrading: large language models for enhanced feedback in programming education". In: *Neural Computing and Applications* 37.2 (2025), pp. 1027–1040 (page 16).

[107] Maciej Pankiewicz and Ryan S. Baker. "Large Language Models (GPT) for automating feedback on programming assignments". 2023. arXiv: 2307.00150 [cs.HC]. URL: https://arxiv.org/abs/2307.00150 (page 16).

[108] Imen Azaiz, Natalie Kiesler, and Sven Strickroth. "Feedback-generation for programming exercises with gpt-4". In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. 2024, pp. 31–37 (page 16).

[109] Priscylla Silva and Evandro Costa. "Assessing Large Language Models for Automated Feedback Generation in Learning Programming Problem Solving". 2025. arXiv: 2503.14630 [cs.SE]. URL: https://arxiv.org/abs/2503.14630 (page 16).

[110] Vibhor Agarwal et al. "CodeMirage: Hallucinations in Code Generated by Large Language Models". 2024. arXiv: 2408.08333 [cs.SE]. URL: https://arxiv.org/abs/2408.08333 (page 16).

[111] Aditya Pathak et al. "Rubric Is All You Need: Enhancing LLM-based Code Evaluation With Question-Specific Rubrics". 2025. arXiv: 2503.23989 [cs.SE]. URL: https://arxiv.org/abs/2503.23989 (page 16).

[112] Umar Alkafaween, Ibrahim Albluwi, and Paul Denny. "Automating autograding: Large language models as test suite generators for introductory programming". In: *Journal of Computer Assisted Learning* 41.1 (2025), e13100 (page 16).

[113] Goda Nagakalyani et al. "TA Buddy: AI-Assisted Grading Tool for Introductory Programming Assignments". In: *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2*. 2025, pp. 1736–1736 (page 16).

[114] Qianhui Zhao et al. “Peer-aided Repairer: Empowering Large Language Models to Repair Advanced Student Assignments”. 2024. arXiv: 2404.01754 [cs.SE]. URL: https://arxiv.org/abs/2404.01754 (page 16).

[115] Zhiyu Fan et al. “Automated repair of programs from large language models”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1469–1481 (page 16).

[116] Jiho Shin et al. “Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code”. 2025. arXiv: 2310.10508 [cs.SE]. URL: https://arxiv.org/abs/2310.10508 (page 17).

[117] Kaiyan Chang et al. “Efficient Prompting Methods for Large Language Models: A Survey”. 2024. arXiv: 2404.01077 [cs.CL]. URL: https://arxiv.org/abs/2404.01077 (pages 17, 19).

[118] Tom B. Brown et al. “Language Models are Few-Shot Learners”. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165 (pages 17, 25, 31, 37, 48, 75, 76).

[119] Archiki Prasad et al. “Self-Consistency Preference Optimization”. 2024. arXiv: 2411.04109 [cs.CL]. URL: https://arxiv.org/abs/2411.04109 (page 17).

[120] Baizhou Huang et al. “Enhancing Large Language Models in Coding Through Multi-Perspective Self-Consistency”. 2024. arXiv: 2309.17272 [cs.CL]. URL: https://arxiv.org/abs/2309.17272 (page 17).

[121] Chenyan Zhao, Mariana Silva, and Seth Poulsen. “Language Models are Few-Shot Graders”. 2025. arXiv: 2502.13337 [cs.CL]. URL: https://arxiv.org/abs/2502.13337 (page 17).

[122] Pranab Sahoo et al. “A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications”. 2025. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927 (page 17).

[123] Lei Wang et al. “Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models”. 2023. arXiv: 2305.04091 [cs.CL]. URL: https://arxiv.org/abs/2305.04091 (page 18).

[124] Xue Jiang et al. “Self-planning Code Generation with Large Language Models”. 2024. arXiv: 2303.06689 [cs.SE]. URL: https://arxiv.org/abs/2303.06689 (page 18).

[125] Microsoft Learn. “Prompt engineering techniques”. Mar. 26, 2025. URL: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering?tabs=chat (visited on 06/23/2025) (page 18).

[126] Sonam Gupta et al. “Selective Self-to-Supervised Fine-Tuning for Generalization in Large Language Models”. 2025. arXiv: 2502.08130 [cs.CL]. URL: https://arxiv.org/abs/2502.08130 (page 18).

[127] Edward J Hu et al. “Lora: Low-rank adaptation of large language models.” In: *ICLR* 1.2 (2022), p. 3 (pages 18, 19, 54, 55).

[128] Meta AI. "AD-DROP: Attribution-Driven Dropout for Robust Language Model Fine-Tuning". URL: https://ai.meta.com/research/publications/ad-drop-attribution-driven-dropout-for-robust-language-model-finetuning/ (visited on 06/23/2025) (page 18).

[129] Tim Dettmers et al. "Qlora: Efficient finetuning of quantized llms". In: *Advances in neural information processing systems* 36 (2023), pp. 10088–10115 (page 18).

[130] Hugging Face. "LoRA (Low-Rank Adaptation)". URL: https://huggingface.co/learn/llm-course/chapter11/4 (visited on 06/23/2025) (pages 18, 55).

[131] Joshua Noble. "What is LoRA (low-rank adaption)?" Jan. 28, 2025. URL: https://www.ibm.com/think/topics/lora (visited on 06/23/2025) (pages 18, 55).

[132] Shengyu Zhang et al. "Instruction Tuning for Large Language Models: A Survey". 2024. arXiv: 2308.10792 [cs.CL]. URL: https://arxiv.org/abs/2308.10792 (pages 18, 33).

[133] Haojie Zhang et al. "Fine-Tuning Pre-Trained Language Models Effectively by Optimizing Subnetworks Adaptively". 2022. arXiv: 2211.01642 [cs.CL]. URL: https://arxiv.org/abs/2211.01642 (page 19).

[134] Chunlei Xin et al. "Beyond full fine-tuning: Harnessing the power of LoRA for multi-task instruction tuning". In: *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. 2024, pp. 2307–2317 (page 19).

[135] Sorouralsadat Fatemi, Yuheng Hu, and Maryam Mousavi. "A Comparative Analysis of Instruction Fine-Tuning LLMs for Financial Text Classification". 2024. arXiv: 2411.02476 [cs.CL]. URL: https://arxiv.org/abs/2411.02476 (page 19).

[136] Andrew Godbout. "Islands of Hex". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE '23), Nifty Assignments Special Session.* Vol. 2. SIGCSE '23. Toronto, ON, Canada: Association for Computing Machinery, Mar. 2023, p. 1275. DOI: 10.1145/3545947.3573223. URL: https://doi.org/10.1145/3545947.3573223 (page 21).

[137] Andrew Godbout. "Islands of Hex". Nifty Assignments handout, http://nifty.stanford.edu/2023/godbout-islands-of-hex/. Accessed: 2025-07-11. 2023 (page 21).

[138] OpenAI. "ChatGPT Free Tier FAQ". https://help.openai.com/en/articles/9275245-chatgpt-free-tier-faq. Accessed: 19 June 2025. 2025 (page 24).

[139] OpenAI. "Introducing vision to the fine-tuning API". https://openai.com/index/introducing-vision-to-the-fine-tuning-api/. Accessed: 2025-06-25. Oct. 2024 (page 26).

[140] OpenAI Help Center. "File Uploads FAQ". https://help.openai.com/en/articles/8555545-file-uploads-faq. Updated over 3 weeks ago; accessed 2025-06-26. 2025 (page 26).

[141] Kunhao Zheng et al. "What Makes Large Language Models Reason in (Multi-Turn) Code Generation?" 2025. arXiv: 2410.08105 [cs.CL]. URL: https://arxiv.org/abs/2410.08105 (page 26).

[142] GitHub Documentation. "Getting free access to Copilot Pro as a student, teacher, or maintainer". https://docs.github.com/en/copilot/managing-copilot/managing-copilot-as-an-individual-subscriber/getting-started-with-copilot-on-your-personal-account/getting-free-access-to-copilot-pro-as-a-student-teacher-or-maintainer. Accessed: 2025-06-25. 2025 (page 28).

[143] cppreference.com. "Function definition". Accessed July 3, 2025. n.d. URL: https://en.cppreference.com/w/c/language/function_definition (visited on 07/03/2025) (page 30).

[144] Meta. "Llama 3.2 Acceptable Use Policy". https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/USE_POLICY.md. Accessed: 2025-07-15. 2025 (page 32).

[145] Terry Yue Zhuo et al. "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions". In: *arXiv preprint arXiv:2406.15877* (2024) (pages 32, 33).

[146] BigCode Project. "BigCodeBench Leaderboard". https://bigcode-bench.github.io/. Accessed: 2025-07-15. 2025 (pages 32, 34).

[147] "Exercism". https://exercism.org/. Accessed: 2025-08-08. Exercism, 2025. URL: https://exercism.org/ (page 33).

[148] LiveCodeBench. "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code". https://github.com/LiveCodeBench/LiveCodeBench. Accessed: 2025-07-16. 2025 (page 33).

[149] Qwen Team. "Qwen2.5-Coder: Code More, Learn More!" https://qwenlm.github.io/blog/qwen2.5-coder/. Accessed: 2025-07-16. Sept. 2024 (pages 33, 34).

[150] LiveCodeBench. "LiveCodeBench Leaderboard". https://livecodebench.github.io/leaderboard.html. Accessed: 2025-07-16. 2025 (page 34).

[151] Artificial Analysis. "Artificial Analysis: AI Model and API Providers Analysis". https://artificialanalysis.ai/. Accessed: 2025-07-16. 2025 (page 34).

[152] Anirban Sen. "Finetuning LLMs using LoRA". https://anirbansen2709.medium.com/finetuning-llms-using-lora-77fb02cbbc48. Accessed: 2025-07-22. Sept. 2023 (pages 54, 56).

[153] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *ArXiv preprint arXiv:1910.03771* (2019). URL: https://github.com/huggingface/transformers (page 56).

[154] Sourab Mangrulkar et al. "PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods". https://github.com/huggingface/peft. 2022 (page 56).

[155] Hugging Face. "Supervised Fine-Tuning". https://huggingface.co/learn/llm-course/en/chapter11/. Accessed 2025-07-22. 2025 (page 56).

[156] Hugging Face. “Examples of using PEFT with TRL to finetune 8-bit models with Low Rank Adaption (LoRA)”. https://huggingface.co/docs/trl/en/peft_integration. Accessed 2025-07-22. 2025 (page 56).

[157] Microsoft. “LoRA: Low-Rank Adaptation of Large Language Models”. https://github.com/microsoft/LoRA. GitHub repository, accessed 2025-07-22. 2023 (page 56).

[158] Alex Aiken. “MOSS: A System for Detecting Software Similarity”. https://theory.stanford.edu/~aiken/moss/. Accessed: 2025-07-23. 1994 (page 57).

[159] OpenAI. “Models — OpenAI Platform Documentation”. https://platform.openai.com/docs/models. Accessed on 25 July 2025. 2025 (pages 64–66).

[160] Andrii Chornyi. “Understanding Temperature, Top-k, and Top-p Sampling in Generative Models”. Accessed: 2025-05-16. Oct. 2024. URL: https://codefinity.com/blog/Understanding-Temperature%2C-Top-k%2C-and-Top-p-Sampling-in-Generative-Models (page 65).

[161] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17.3 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2 (page 86).

[162] Meta AI. “Llama 4: Multimodal Intelligence”. https://ai.meta.com/blog/llama-4-multimodal-intelligence/. Accessed: 2025-06-05. Meta AI, Apr. 2025.

[163] Sissie Hsiao. “Faites-en plus avec Gemini : Essayez 1.5 Pro et des fonctionnalités plus intelligentes”. https://blog.google/intl/fr-fr/nouvelles-de-lentreprise/gemini-update-mai-2024-fenetre/. Accessed: 2025-06-19. May 2024.