

Rubric-Based Assessment of Introductory C Programming with LLM Agents

Filippo Manca¹, Alessio Ferrato^{1,*}

¹Department of Engineering, University of Roma Tre, Via Vito Volterra 62, 00146 Rome, Italy

Abstract

Grading programming assignments must account simultaneously for functional correctness and for qualitative aspects such as structure, readability, style. Automated Assessment Tools handle correctness well but capture the complex criteria instructors actually apply only poorly, and Large Language Models (LLMs), while more flexible, operate as stateless single-pass generators. This paper asks whether an *agentic* formulation of the grading task offers concrete advantages over direct LLM prompting. We present a grading agent that inspects student submissions against a teacher-defined rubric and produces inline, contextualized feedback without altering the original code. The agent is evaluated on real exam submissions written in C by first-year undergraduates, drawn from two examination sessions and previously graded by human instructors, and is compared both against those grades and against a non-agentic LLM baseline on the same corpus. The results are mixed. On the *function* component the agent improves over the baseline in both sessions, while on the *main* component it degrades sharply, driven by systematic over-penalisation of minor style violations that human graders tend to forgive. We report a taxonomy of the observed failure modes and the architectural refinements they suggest.

Keywords

Automated Assessment, Agentic AI, Large Language Models, Programming Education, Rubric-Based Grading

1. Introduction

Assessing programming assignments is demanding. In large introductory courses, instructors review dozens or hundreds of submissions, often across several exam variants, and each must be judged along two axes: whether the program does what it should, and whether it does so in a way that reflects sound practice. Delivery through Learning Management Systems such as Moodle has amplified the volume of submissions without reducing the effort required per submission. Manual grading suffers from a well-known intrinsic inconsistency: different graders, or the same grader at different moments, may assign divergent scores according to subjective criteria. Structured rubrics and peer assessment mitigate the problem but do not remove it. Automated Assessment Tools (AATs) sit at the opposite end of the trade-off: they automate correctness checking through unit tests, reference comparison, or static analysis, and are therefore fast and perfectly consistent, but they say little about structure, clarity, or documentation.

Large Language Models have opened a third path. Because they analyze code as text, they can in principle comment on structural and stylistic aspects, produce natural-language explanations, and be steered by flexible rubrics. While prompt engineering can help, most simple approaches fall apart in complex grading tasks. Agentic AI may resolve such problems. An agent built around an LLM can plan, selects tools, observes the outcome of its actions, and revises its strategy. If grading requires multi-step reasoning, checking a criterion, locating the relevant code region, inspecting its context, deciding whether a deduction is warranted, then a deliberative agent should be better matched to the task than a single-pass generator. This paper puts that hypothesis to an empirical test. Our contributions are:

1. a rubric-driven grading agent, implemented with smolagents¹, that annotates student C code

AI-LEAP 2026 - 1st International Workshop on AI Learning, Education And Practice, co-located with AIXIA 2026, University of Perugia, October 2026

*Corresponding author.

✉ filippo.manca@stud.uniroma3.it (F. Manca); alessio.ferrato@uniroma3.it (A. Ferrato)

🆔 0000-0001-5405-3018 (A. Ferrato)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

¹<https://github.com/huggingface/smolagents> (last access: October 27, 2026)

with inline, criterion-anchored comments and point deductions while leaving the submission itself untouched;

2. an evaluation on real exam data from two sessions of a first-year CS course against human instructor grades, using three state-of-the-art LLMs as the agent core;
3. a direct comparison with a previous non-agentic LLM pipeline [1] on the same corpus;
4. a characterisation of the failure modes that currently prevent agentic grading from matching human assessment, with the design implications we draw from them.

2. Related Work

LLM-based automated grading. Early work asked whether LLMs could *solve* programming problems rather than assess them: Kiesler and Schiffner [2] evaluated ChatGPT-3.5 and GPT-4 on 72 introductory Python tasks, reporting accuracies of 94.4–95.8% and establishing the competence baseline on which assessment applications could build. Attention then shifted to assessment. Xie et al. [3] proposed *Grade-Like-a-Human*, a LangChain-orchestrated pipeline combining student-answer-aware rubric generation, grading with advanced prompting, and post-grading review, with significant gains in accuracy and consistency; the use of an orchestration framework gestures towards agentic design, yet the workflow remains predefined and lacks dynamic tool use. Alkafaween et al. [4] instead used GPT-4 to generate test suites for CS1 problems, and across 26 problems and more than 25,000 solutions the generated suites matched or exceeded instructor-authored ones. Lagakis et al. [5] examined GPT-3.5 and GPT-4 in a Greek-language Python MOOC, highlighting both the scalability benefit and the degradation observed for less-resourced languages. Ensemble and fine-tuning strategies have also been explored. CodEv [6] combines several LLMs with Chain-of-Thought prompting [7] and one-shot, self-reflection, and batching strategies, showing that ensembles of small-parameter models can approach large-parameter performance with excellent intra-model ($ICC > 0.97$) and good inter-model ($ICC > 0.84$) agreement; it nonetheless remains a static ensemble, without dynamic reasoning or external tools. BeGrading [8] instead fine-tunes a specialised model on 3,470 real samples augmented with synthetic data, a viable alternative to the dominant prompting paradigm.

Agentic approaches. To the best of our knowledge, LLM-based agents are recent, and where they appear in a programming context they are used for code generation, debugging, or optimisation, for producing code instead of judging it. The closest contribution is Berlin [9], whose Llama-3-70b Multi-Agent System raised the acceptance rate by 6.7% and reduced the failure rate by 11.7% over a single LLM on 120 Kattis problems; the task, however, is problem solving, not pedagogical assessment. To our knowledge, no prior work applies an agentic paradigm to rubric-based grading of student code. On the prompting side, Spiess et al. [10] showed that jointly optimising agentic patterns (ReAct, ReWOO) and prompt content yields improvements of up to 68.9 percentage points on complex multi-step tasks, while establishing that no single prompting strategy is universally optimal.

Known limitations. Reliability remains the central obstacle. Smolić et al. [11] report that LLMs do not reach full reliability in automated assessment, particularly for non-compiling code or memory-management defects that escape simple tests: in their study of 50 C assignments, GPT-3.5 achieved a correlation with human graders the authors judged insufficient for the use case, while Gemini’s correlations were low enough that the scores could plausibly have been assigned by chance. Alongside reliability, the literature converges on three weaknesses: reliance on probabilistic pattern matching, limited contextual reasoning and tool integration, and unclear generalization. Those are precisely the weaknesses an agentic architecture is meant to address.

3. An Agentic Grading System

3.1. Dataset and rubric

Our corpus consists of code submissions from a first-year introductory computer science course, produced during the mid-term examinations of academic years 2021 and 2023. All assignments are array-based tasks implemented in C, and the dataset comprises more than 25 graded exercises. Each submission had been graded by the two instructors teaching the course. They provided feedback through a standardised comment format following the schema `<comment> <error type> <point deduction>`; for example, the annotation below records a one-point deduction for not using a boolean variable to terminate a loop.

```
//**** NOT USING THE BOOLEAN VARIABLE TO END THE LOOP-1.
```

Every graded submission also carries a header summarising the total score and its distribution across the micro-tasks of the exercise (typically a *function* component worth up to 6 points and a *main program* component worth up to 2.5). For each exercise we collected the assignment description, the original student code as submitted through Moodle, the assessment criteria used, and the grades actually awarded by the instructors. This last element is what makes the corpus valuable since the reference is not a reconstructed gold standard but the assessment that determined the students' marks. To make the scoring system explicit, we grouped common student errors into the five categories of Table 1, each with indicative point deductions.

Table 1

Rubric categories used by the agent, with representative violations.

Category	Representative violations
Structure and organisation	All logic inside main instead of separate functions; function called with missing or incorrect parameters.
Iteration errors	Loop variables not updated (infinite loops); indexing outside array bounds; improper break or return inside loops.
Logical errors	Overly complex or incorrect conditionals; misuse of logical operators; misunderstanding of the problem. Typically yield partial or incorrect solutions that still compile and run.
Syntax and declaration	Uninitialised or undeclared variables; incorrect operators (e.g. = for ==); typographical mistakes causing compilation failure.
Style errors	Inconsistent or incorrect indentation; missing comments; excessive return statements in a single function.

3.2. Agent architecture and prompting

The agent proposed in this work is built on smolagents [12]: CodeAgent, which emits tool calls as executable Python and offers high expressivity, and ToolCallingAgent, which emits structured JSON and offers higher reliability. Because student code must be handled in a controlled manner, we also relied on the framework's sandboxing: the interpreter permits only the supplied tools and a safe list of built-ins, with E2B [13] and Docker executors available for stronger isolation.

Architecturally the system is a *deliberative* agent: it maintains an internal representation of the task, plans which checks to perform, and revises that plan on the basis of what it observes, in contrast with reactive agents that map percepts to actions through static condition-action rules. Reasoning follows the ReAct pattern [14], alternating *Thought* (what to check next), *Action* (invoking an analysis tool), and *Observation* (interpreting the result); memory carries context across iterations, so the outcome of one check informs the next. Figure 1 sketches the pipeline. Assignment text, student code, and rubric are supplied to the agent, which traverses the rubric one criterion at a time. Each pass may add at most one inline comment for that criterion, restricted to the relevant section of the program, and the annotated

code produced by pass i becomes the input of pass $i + 1$. After all N criteria have been examined, the resulting score is compared with the instructor’s grade.

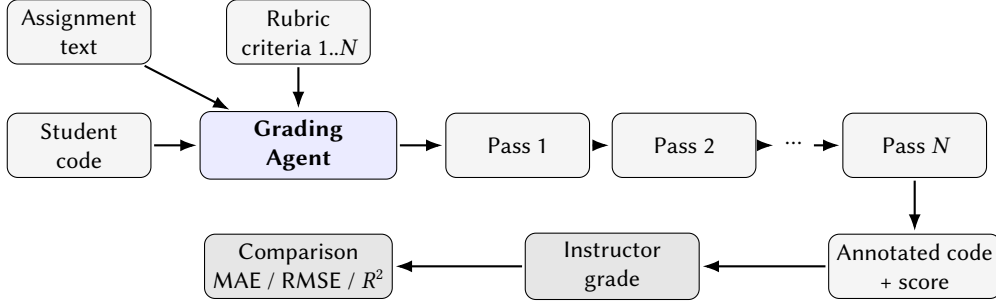


Figure 1: Pipeline of the agentic grading system. The rubric is traversed one criterion per pass; each pass may add at most one inline comment, and the annotated output of one pass feeds the next.

The current design fixes the sequence of criteria externally instead of letting the agent choose what to check and in what order; this trades full autonomy for predictability while the calibration failures described in Section 5 are still being characterised. We treat this as a first step: once misattribution and over-penalisation are brought under control (Section 6), a natural extension is to let the agent itself decide which criteria to check and in what order, moving traversal from a fixed pipeline toward the fuller deliberative planning the architecture is designed to support. An initial and somewhat counter-intuitive finding shaped the prompting strategy. Prior work on the same corpus with non-agentic LLMs [1] relied on long, heavily structured prompts. Transferring that style to the agent degraded performance: with highly detailed instructions the agent tended to lose track of individual requirements, failing to attend to all of them simultaneously. A short, precise prompt scoped to a single criterion allowed it to concentrate on the information actually needed for that check, consistent with the observation of Spiess et al. [10] that prompting strategies do not transfer uniformly across models and task formulations. The resulting template instructs the agent to analyse the code for one named violation, within one named section, and to add the corresponding comment at most once, in the form `//***** {criterion} - {points}`. Four constraints proved essential: never modify the student’s C code; never replace or alter pre-existing comments, only add new ones on a separate line; return the complete code through `final_answer()`; and do not attempt to execute the C code, only analyse it as text. The last was added after observing the failure mode described in Section 5.

4. Experimental Setup

We evaluated three state-of-the-art models as the agent core: gemini-2.0-flash-001 [15], GPT-4o [16], and DeepSeek-V3 [17]. Temperature was set to 0 for all interactions to obtain deterministic outputs, enabling reproducible results and fair comparison. Agent-produced scores are compared with instructor scores using three standard regression metrics, which give complementary views on accuracy, consistency, and explanatory power:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i| \quad \text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2} \quad R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{\sum_{i=1}^N (y_i - \bar{y})^2} \quad (1)$$

where y_i is the score assigned by the human grader, $\hat{y}_i$ the score predicted by the agent, N the number of graded exercises, and $\bar{y}$ the mean human score. MAE gives a direct measure of error magnitude but does not emphasise large deviations; RMSE penalises large divergences more heavily; R^2 assesses how well the agent’s scores explain the variance in the human ones. Section 6 returns to the interpretation of R^2 on components whose human scores have very low variance. Scores are reported separately for the *function* component, the *main* component, and the *total*, mirroring the decomposition used by the instructors.

5. Results

Table 2

Agreement between agent-assigned and instructor-assigned scores.

Year / Component	Model	MAE	RMSE	R^2
2021 – function	deepseek-v3	1.100	1.158	0.540
2021 – function	gemini-2.0-flash	1.640	1.740	-0.039
2021 – main	deepseek-v3	1.240	1.390	-12.567
2021 – main	gemini-2.0-flash	0.520	0.684	-2.287
2021 – total	deepseek-v3	2.340	2.402	-0.465
2021 – total	gemini-2.0-flash	2.160	2.259	-0.295
2023 – function	deepseek-v3	1.660	1.702	-0.316
2023 – function	gemini-2.0-flash	3.260	3.391	-4.221
2023 – main	deepseek-v3	1.080	1.101	-19.200
2023 – main	gemini-2.0-flash	0.400	0.548	-4.000
2023 – total	deepseek-v3	2.740	2.774	-2.120
2023 – total	gemini-2.0-flash	3.460	3.577	-4.189

Table 2 reports performance across models, years, and components. **DeepSeek-V3** behaves heterogeneously. Its best result is on the function component of the 2021 exercises ($R^2 = 0.540$), indicating a moderate ability to align with human criteria for functional logic. The main component reveals the most systemic weakness, with strongly negative R^2 in both years, suggesting difficulty in interpreting the structure and organisation of the main program. In some cases the model assigns zero to a main component that a strict reading of its own deductions would have driven below zero; grades are floored at zero because negative marks are not used.

Gemini-2.0-flash exhibits polarised grading, oscillating between drastic under-scoring and excessive over-scoring, which compromises the consistency of its calibration. Degradation across sessions is pronounced on the function component (MAE from 1.640 to 3.260, R^2 from -0.039 to -4.221), while the main component shows lower absolute errors than DeepSeek despite negative correlations, suggesting comparatively better handling of structural assessment. Its principal weakness surfaces on the most problematic submissions, where it generates redundant comments for multiple violations of the same kind without consolidation, repeatedly penalising similar aspects instead of retaining only the most serious one.

GPT-4o is absent from Table 2. Its agentic runs exhibited a failure mode more severe than any observed with the other models: the agent repeatedly attempted to *execute* the C code instead of analysing it, exhausting its step budget and compromising the correction. The same tendency appeared occasionally with DeepSeek and Gemini, where a prompt adjustment sufficed to suppress it; with GPT-4o it proved persistent. We report this as a substantive result. An agent’s propensity to reach for tool execution when a purely analytical response is required is model-dependent and cannot be assumed away.

5.1. Comparison with the non-agentic baseline

Table 3 contrasts agentic and non-agentic DeepSeek-V3, the only model present both here and in the earlier LLM evaluation on the same corpus [1]. Holding the model constant lets us compare the agentic and non-agentic pipelines on the same corpus; the two conditions also differ in prompting regime, as noted in Section 5 above, so this comparison is consistent with a contribution from the agentic layer. The pattern is consistent across both sessions and, we believe, more interesting than a uniform result would have been. On the *function* component, the part of the exercise that carries the algorithmic weight and the largest share of the marks, the agent improves on every metric in both years. On the *main* component it degrades sharply the aggregate figures favour the non-agentic pipeline.

Table 3

Agentic versus non-agentic DeepSeek-V3 on the same corpus. Lower is better for MAE and RMSE, higher for R^2 ; best value per row in bold.

Year	Component	MAE		RMSE		R^2	
		LLM	Agent	LLM	Agent	LLM	Agent
2021	function	1.273	1.100	1.536	1.158	0.302	0.540
2021	main	0.434	1.240	0.544	1.390	0.184	-12.567
2021	total	1.565	2.340	1.859	2.402	0.299	-0.465
2023	function	1.840	1.660	2.064	1.702	-0.584	-0.316
2023	main	0.319	1.080	0.400	1.101	0.350	-19.200
2023	total	2.051	2.740	2.297	2.774	-0.366	-2.120

5.2. Failure modes

Inspection of the generated corrections revealed recurring categories of error:

- **Misattribution of errors** was the most common: the agent flags problems that do not exist, for instance claiming that a loop increment (`i++`) or a return statement is missing when it appears on the line under examination. The same defect occurs in the non-agentic setting, but at lower frequency.
- **Sequence misinterpretation** leads the model to misread the order of operations, asserting that an array is initialised before its length is set when the code does no such thing; this may originate in tokenisation, which flattens the sequential dependencies of a program into a representation from which they are hard to recover.
- **Over-sensitivity in style assessment** is pronounced with DeepSeek, which inserts subjective comments with very low tolerance thresholds (i.e., “very bad indentation”, “poor explanation of the called function”) for minor violations that would not warrant a deduction.
- **Lack of contextual calibration** applies style criteria rigidly, penalising individual badly indented lines even when the code is globally well structured.
- **Uncalibrated subjectivity** prevents the agent from distinguishing impactful stylistic problems from cosmetic imperfections.

These last three modes are not independent, and together they explain the asymmetry in Table 3. Style-related deductions are individually small but concentrated in the main function, and their high frequency produces a cumulative effect that drives the agent’s main-component scores away from the instructors’. Human graders are markedly more lenient on formal aspects once the functional logic is substantially correct; the agent is not.

6. Discussion

Where the agentic layer helps. The improvement on the function component in both sessions is the clearest positive signal. This is the part of the task that most resembles what agentic reasoning is designed for: locating a specific behaviour in an unfamiliar program, checking it against a criterion, and deciding on the evidence. It is also the part carrying the greater pedagogical weight, since it concerns whether the student solved the problem instead of how neatly they wrote it down. A second advantage escapes the metrics entirely. In the earlier LLM study many exams were returned with only the score header and the code, the specific inline comments having been lost in generation; our agent preserves them in every correction. For a system meant to support teachers rather than replace them this matters, since a score without a justification is of limited use to a reviewing instructor and of none to a student.

On reading R^2 for the main component. The main component is worth at most 2.5 points and, in practice, most students score at or near the maximum. Human scores on this component therefore have very low variance, placing a very small denominator in the R^2 expression. Errors modest in absolute terms (MAE between 0.4 and 1.24 points) consequently generate values as extreme as -19.2 . These are arithmetically correct but should not be read as evidence that the agent’s main-component assessments are twenty times worse than predicting the mean in any intuitive sense; MAE and RMSE are the more informative metrics here. We flag this explicitly because the pattern is likely to recur in any rubric with a low-variance sub-component.

Cost and throughput. The economic dimension is often overlooked. Advanced models such as Claude-3.5 Sonnet and GPT-4o, while performing better, incur operating costs that can reach several dollars per single exam, making large-scale adoption impractical for institutions with limited budgets. On throughput, each assignment currently requires roughly 20 minutes of agent processing, so a session of 100 assignments amounts to some 33 hours of computation. This is unattended time and compares favourably with manual grading, but it is not negligible and follows directly from the multi-pass design.

Implications and limitations. The calibration inconsistencies and the tendency to over-penalise indicate that systems of this kind should, for now, support human assessment. The realistic role is a first-pass annotator whose output the instructor reviews and finalises, shifting the teacher from grader to supervisor of the grading process; the rubric-anchored output makes this viable, since every deduction is tied to a named criterion and a specific line. Three limitations qualify our findings. The corpus is small and drawn from a single course, language, and problem family, which limits generalisation. The comparison with the non-agentic baseline holds the model constant but not the prompt, since the two systems required different prompting regimes to function at all. Finally, we evaluate agreement with instructor scores, not the pedagogical usefulness of the feedback, which would require a study with students.

7. Conclusions and Future Work

We examined whether an agentic formulation of automated code assessment offers advantages over direct LLM prompting, implementing a deliberative agent on the smolagents framework that grades student C submissions against a teacher-defined rubric and evaluating it on real exam data against human instructor grades and a non-agentic baseline. The agent improves on the function component in both sessions, demonstrably preserves inline feedback where the baseline does not, and enforces a structured, transparent grading procedure. It also over-penalises style, mishandles the main component, and, with one of the three models tested, fails outright by attempting to execute code it was asked to read. Four directions follow from the failure analysis. First, a **two-level validation pipeline**, in which a second agent specialised in verifying the first one’s output acts as a “grader of the grader”, assessing each comment for pertinence, accuracy, and consistency with the rubric, targeting the misattribution and sequence errors that currently dominate. Second, **intelligent comment consolidation**, identifying comments that address similar aspects of the code and retaining only the one carrying the greater penalty, which would directly attack the cumulative over-penalisation in the main component. Third, **dynamic prompt optimisation**, adapting the level of detail and the sensitivity of the assessment to the preliminary score of a submission. Fourth, **continual learning** from instructor feedback, allowing the agent to refine its judgement criteria towards the expectations of a specific educational context.

These refinements point towards treating autonomy itself as a tunable parameter rather than a fixed property of the system [18], letting instructors start from fully reviewable suggestions and delegate progressively as confidence in the agent’s calibration grows. Whether these refinements close the gap with non-agentic pipelines is an open question worth answering.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Sonnet 4 in order to: organise and check content, paraphrase and reword, and perform grammar and spelling checks. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] E. Cipriano, A. Ferrato, C. Limongelli, D. Schicchi, D. Taibi, Leveraging large language models to assist teachers in code grading, in: *Proceedings of the 26th International Conference on Artificial Intelligence in Education (AIED 2025)*, Springer, 2025.
- [2] N. Kiesler, D. Schiffner, Large language models in introductory programming education: ChatGPT's performance and implications for assessments, *arXiv preprint arXiv:2308.08572* (2023).
- [3] W. Xie, J. Niu, C. J. Xue, N. Guan, Grade like a human: Rethinking automated assessment with large language models, *arXiv preprint arXiv:2405.19694* (2024).
- [4] U. Alkafaween, I. Albluwi, P. Denny, Automating autograding: Large language models as test suite generators for introductory programming, *Journal of Computer Assisted Learning* 41 (2025) e13100.
- [5] P. Lagakis, S. Demetriadis, G. Psathas, Automated grading in coding exercises using large language models, in: *Interactive Mobile Communication, Technologies and Learning*, Springer, 2023, pp. 363–373.
- [6] E.-Q. Tseng, P.-C. Huang, C. Hsu, P.-Y. Wu, C.-T. Ku, Y. Kang, CodEv: An automated grading framework leveraging large language models for consistent and constructive feedback, in: *2024 IEEE International Conference on Big Data (BigData)*, IEEE, 2024, pp. 5442–5449.
- [7] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, *Advances in Neural Information Processing Systems* 35 (2022) 24824–24837.
- [8] M. Yousef, K. Mohamed, W. Medhat, E. H. Mohamed, G. Khoriba, T. Arafa, BeGrading: Large language models for enhanced feedback in programming education, *Neural Computing and Applications* 37 (2025) 1027–1040.
- [9] C. Berlin, Exploring an AI Agentic Workflow for Solving Challenging Coding Problems: An Evaluation of a Large Language Model Based Multi-Agent System, Master's thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science (EECS), Stockholm, Sweden, 2025. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-361854>, tRITA-EECS-EX 2025:48.
- [10] C. Spiess, M. Vaziri, L. Mandel, M. Hirzel, AutoPDL: Automatic prompt optimization for LLM agents, *arXiv preprint arXiv:2504.04365* (2025).
- [11] E. Smolić, M. Pavelić, B. Boras, I. Mekterović, T. Jaguš, LLM generative AI and students' exam code evaluation: Qualitative and quantitative analysis, in: *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, IEEE, 2024, pp. 1261–1266.
- [12] A. Roucher, A. V. del Moral, T. Wolf, L. von Werra, E. Kaunismäki, smolagents: A smol library to build great agentic systems, <https://github.com/huggingface/smolagents>, 2025. Hugging Face.
- [13] E2B Team, E2B: Virtual machines for AI agents, <https://e2b.dev/>, 2024. Accessed: December 2024.
- [14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, in: *International Conference on Learning Representations*, 2023.
- [15] Google, Gemini 2.0 flash – official documentation, <https://ai.google.dev/gemini-api/docs>, 2024. Accessed: 2025.
- [16] OpenAI, GPT-4o system card, *arXiv preprint arXiv:2410.21276* (2024).
- [17] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al., DeepSeek-V3 technical report, *arXiv preprint arXiv:2412.19437* (2024).

- [18] A. Geninatti Cossatin, F. Ferrero, L. Ardissono, N. Mauro, The autonomy equation: How agentic AI reshapes trust and workload in routine productivity applications, *Information Processing & Management* 63 (2026) 104681. URL: <https://www.sciencedirect.com/science/article/pii/S0306457326000725>. doi:<https://doi.org/10.1016/j.ipm.2026.104681>.