

263-3300-10L Data Science Lab: Evaluation of a Swiss LLM-based Coding Assistant

Sinuo Liu*

Department of Computer Science
ETH Zurich, Switzerland

Imanol Schlag

ETH AI Center, ETH Zurich, Switzerland

Anna Kravchenko*

Department of Computer Science
ETH Zurich, Switzerland

Robert Matthew Smith

ETH AI Center, ETH Zurich, Switzerland

Abstract

This project investigates whether Swiss-hosted open large language models (LLMs) from the Apertus family can serve as reliable coding assistants when accessed via terminal-based tools. We design and implement an end-to-end evaluation pipeline that links SwissAI/CSCS endpoints, the OpenCode terminal agent, and the Terminal-Bench benchmark suite on the Clariden/Alps infrastructure. Using this pipeline, we compare Apertus-8B/70B-Instruct against strong open baselines, including Qwen3, DeepSeek-V3.1, Kimi-K2-Instruct, and Mistral-7B.

The empirical results reveal a systematic trade-off between latency and answer richness: smaller Swiss-hosted models respond rapidly but with more concise outputs, whereas larger baselines generate slower yet more detailed code and explanations. On containerised Terminal-Bench tasks, the strongest non-Swiss models currently achieve higher task success rates, indicating that Apertus, in its present form, is not yet competitive as a fully autonomous terminal agent, despite favourable serving latency and local hosting. We further document practical observations on infrastructure choices, such as model spin-up and endpoint readiness on SwissAI/CSCS compared to fully managed gateways (e.g., OpenRouter). Overall, the work delivers a reusable evaluation pipeline and an initial empirical characterisation of Swiss-hosted LLMs as coding assistants in realistic terminal workflows.

Keywords

large language models, coding assistants, SwissAI, Apertus, Terminal-Bench, command-line interfaces, benchmarking

*Equal contribution.

Author contributions using the CRediT framework [1]:

AK, SL: Conceptualization, Methodology, Software and infrastructure setup, Investigation, Analysis, Project administration, Visualization, Writing

Preprint. This course project work can be distributed as a preprint and has not been peer-reviewed. It does not constitute archival publication and remains eligible for submission to academic venues, including workshops, conferences, and journals.

License. The authors grant ETH Zurich and the ETH AI Center a non-exclusive license to display this work on their platforms to showcase student projects. Redistribution or publication by others outside of academic venues requires the consent of the authors.

Code. Associated code is available open-source and under the MIT License, which permits free reuse, free modification, and free distribution, provided proper attribution is given to the authors, and no liability is assumed by the authors. Follow up work is not required to be open-source and is not required to have an MIT License. The code and its MIT license are publicly available here:

<https://github.com/akrav4enk0/Data-Science-Lab>

263-3300-10L Data Science Lab, December 19, 2025, Zurich, Switzerland

© 2025 Copyright held by the owner/author(s).

1 Introduction

Large Language Models (LLMs) have fundamentally transformed how developers interact with code. Modern LLMs such as GPT-4, Claude, and various open-source alternatives can serve as coding assistants, helping developers write, debug, and optimize code through conversational interfaces.

The Apertus model family [2], developed by ETH Zurich, EPFL, and CSCS represents Switzerland's initiative to provide fully open LLMs with transparent source code, model weights, and training data. Trained on datasets covering over 1,000 languages and hosted entirely on Swiss supercomputing infrastructure (Alps at CSCS) [3], Apertus is designed for multilingual environments and compliance-sensitive applications.

This study evaluates whether Swiss-hosted open LLMs, particularly Apertus-8B and Apertus-70B, can serve as reliable coding assistants when accessed through terminal-based tools. We compare these models against state-of-the-art open-source LLMs including Qwen, DeepSeek, Kimi, GLM, Mistral across multiple dimensions: response latency, answer quality, operational stability, and task completion accuracy.

By building an end-to-end evaluation pipeline connecting SwissAI endpoints [4] to standardized benchmarks and interactive coding tools, we provide empirical evidence about the current capabilities and practical limitations of Swiss LLMs for real-world development workflows.

We also examine the impact of infrastructure limits, specifically SwissAI endpoints on the Clariden cluster and OpenCode deployment. Our work focuses on building an end-to-end evaluation pipeline that runs locally, communicates with SwissAI endpoints, and is designed to scale to the Clariden infrastructure for larger experiments. Through quantitative benchmarking on 80 terminal tasks and qualitative interactive assessments, we provide empirical evidence about the current capabilities and practical limitations of Swiss LLMs for real-world development workflows.

2 Related work

Evaluation of LLM-based coding assistants requires understanding three areas: the models themselves, the tools that enable practical interaction, and the benchmarks used to assess them. Our study compares several prominent open-source models: Qwen3 from Alibaba demonstrates strong multilingual and coding capabilities across various parameter scales [5]; DeepSeek-V3.1 employs a mixture-of-experts architecture with 685 billion parameters achieving state-of-the-art results on coding benchmarks [6]; Kimi-K2 from Moonshot AI represents a frontier model with more than 1 trillion

parameters showing strong performance on complex reasoning tasks [7]; GLM-4.6 from Zhipu AI offers efficient multilingual capabilities optimized for Chinese and English [8]; Mistral-7B provides a compact and efficient model suitable for resource-constrained deployments [9]; Apertus leverages over 15 trillion high-quality tokens across 1,500 languages to achieve the robust linguistic understanding and versatility required for complex coding tasks [2].

The practical deployment of LLM coding assistants depends on accessible tooling that bridges models and development environments. OpenCode provides a terminal agent interface where developers describe tasks conversationally and receive executable commands [10]. To enable flexible model access, OpenAI-style API gateways have emerged, including SwissAI/CSCS for Swiss infrastructure, PublicAI for public access to European models [11], and OpenRouter for unified access across multiple providers [12].

Complementary to these tools, SWE-agent evaluates LLMs as software-engineering agents on real GitHub repositories, offering a repo-level perspective on code quality and task completion that complements our terminal-first focus [13; 14].

Evaluating accuracy requires benchmarks that reflect realistic developer workflows. Terminal-Bench addresses this need by providing 80 tasks that test an LLM’s ability to perform practical terminal operations including file manipulation, system configuration, git operations, game and shell scripting [15].

Our work builds on this ecosystem by connecting Swiss-hosted models through SwissAI endpoints to Terminal-Bench and OpenCode, creating an evaluation pipeline that assesses both quantitative performance and qualitative usability.

3 Methodology

3.1 Infrastructure

Our evaluation infrastructure centered on the Clariden cluster at CSCS, which hosts the SwissAI platform. SwissAI exposes models through API endpoints that follow the OpenAI API specification, enabling direct integration with existing OpenAI-compatible tools. Using the SwissAI model-launch tooling, we deployed Apertus-8B/70B-Instruct and other selected models on Clariden and exposed them via SGLang-based serving stacks.

This design allowed us to use opencode.ai and other standard CLI tools to interact with SwissAI-hosted models without custom client implementations. In parallel, we set up a local development environment with a Python virtual environment and Docker Desktop to run Terminal-Bench containers on the laptop. While the cluster-based deployment provided the computational resources necessary for hosting large models.

3.2 Model Deployment

We utilized SwissAI infrastructure to deploy models on the Clariden cluster through a SLURM job submission system. The models were served using SGLang or vLLM frameworks, both of which expose OpenAI-compatible API endpoints. Smaller models (Mistral-7B, Qwen3-80B) ran on single nodes, while larger models (DeepSeek-V3.1, Kimi-K2, GLM-4.6) required multi-node deployment across 4 nodes with 16-way tensor parallelism due to their size. The OCF (Open Compute Framework) enabled service discovery and health

monitoring. The Apertus models ran continuously as persistent services, while the comparison models were provisioned on-demand for evaluation periods. This deployment approach provided consistent API-based access across all models while keeping computation within CSCS infrastructure.

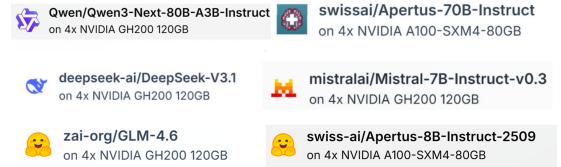


Figure 1: A few of spun up models on CSCS Cluriden cluster.

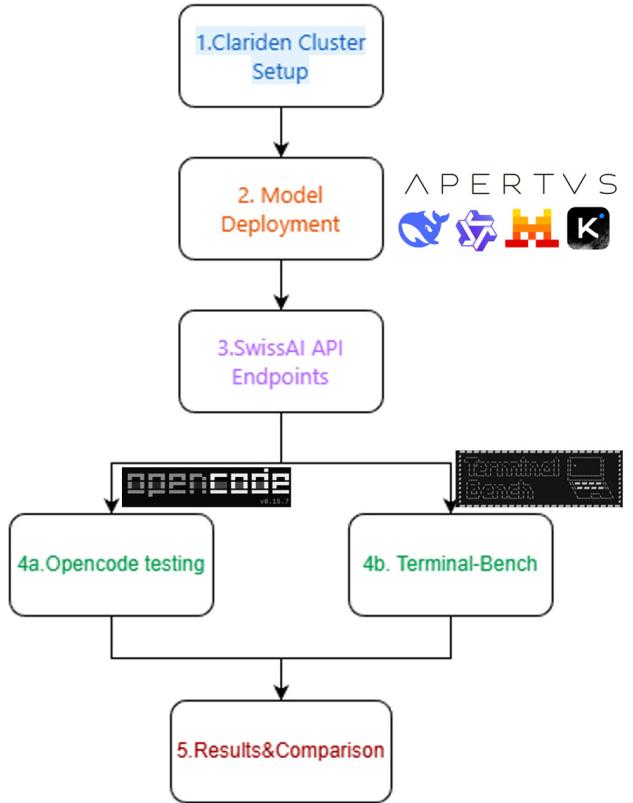


Figure 2: Working flow

3.3 Opencode and CLI Testing

To assess practical usability and response speed in a realistic developer workflow, we combined an OpenAI-compatible CLI client with interactive experiments in the opencode.ai terminal agent.

We first configured OpenCode to use the SwissAI/CSCS endpoints by setting OPENAI_BASE_URL, OPENAI_API_KEY and OPENAI_MODEL to point to the currently deployed model on Clariden/Alps.

This allowed us to interact with Apertus and baseline models directly from a terminal UI without custom integration code, mirroring how a developer would use a coding assistant inside an existing project.

We used OpenCode interactively with a curated set of CLI-oriented prompts (e.g., writing and refactoring short Python functions, simple shell utilities, and explanations of generated code). For these sessions we recorded qualitative observations on instruction-following, clarity of explanations, error handling, and perceived latency.

```
Generate Python unit tests for a function that checks for prime numbers.
anyachenko (02:09 AM)

Write test_prime.py

import unittest

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

class TestPrime(unittest.TestCase):
    def test_prime_numbers(self):
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))

    def test_non_prime_numbers(self):
        self.assertFalse(is_prime(1))
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(9))

    def test_edge_cases(self):
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(-1))
        self.assertFalse(is_prime(-5))

if __name__ == '__main__':
    unittest.main()
```

Figure 3: Example OpenCode TUI session using Qwen3-Next-80B-A3B-Thinking model answering a coding prompt.

```
# Implementing Python prime check function
/share to create a shareable link

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
Build grok-code (02:38 AM)
```

Figure 4: Example OpenCode TUI session using Qwen3-Next-80B-A3B-Thinking model answering a coding prompt.

In particular, we observed that Apertus-8B (Fig. 3) tends to return short, focused code snippets, whereas larger baselines such as Qwen3 (Fig. 4) frequently produce longer, tutorial-style responses.

This behaviour complements the quantitative latency measurements obtained from the non-interactive client.

After, we built a non-interactive dedicated `swissai-demo` client in Python client, which sends a fixed coding prompt to a chosen model and measures wall-clock latency from request initiation to receipt of the full response, in order to systematically evaluate each model's interactive performance.

We logged comprehensive per-model summaries including mean latency, response length in tokens and characters, and qualitative observations about answer completeness and clarity. Our primary latency probe used a Fibonacci sequence generation task, executed 50 times per model to capture performance variability and ensure statistical reliability. The aggregated statistics (mean latency and qualitative differences in answer verbosity) are reported in Results and provide a controlled view of serving performance.

3.4 Terminal-bench

We conducted quantitative evaluation using Terminal-Bench core v0.1.1, which provides 80 containerized terminal tasks covering realistic developer workflows including file operations, system administration, shell scripting, text processing, and development tools. For each model, we executed all 80 tasks once, recording the number of resolved trials (tasks completed successfully) and unresolved trials (tasks failed or timed out).

blind-maze-explorer-5x5	get-bitcoin-nodes	pytorch-model-cli-hard
blind-maze-explorer-algorithm	git-multi-branch	qemu-alpine-sh
blind-maze-explorer-algorithm.easy	git-workflow-hack	qemu-startup
blind-maze-explorer-algorithm.hard	gnupg-codesign	raman-fitting
build-and-run-qemu	git-submodule-transform	raman-fittransform
build-linux-kernel-qemu	hello-world	reshard-c4-data
build-tcc-qemu	heterogeneous-dates	results.json
carrot-pipeline-training	httpd-ssl	run-multi-code
chess-best-move	incompatible-python-fasttext	run-log
conda-env-conflict-resolution	incompatible-python-fasttext_base_with_hint	run-metadata.json
configure-git-webserver	intrusion-detection	sanitize-git-repo
count-lines-of-tokens	justify-ansi-to-utf8	sanity-qemu-hard
crack-7z-hash	modernize-fortran-build	security-vuln-minio
crack-7z-hash.easy	new-encrypt-command	simple-sheets-put
crack-7z-hash-hard	nginx+request-logging	simple-web-scraper
cross-socket	oci	sonic-eval
cron-broken-network	openssl-selfsigned-cert	sqlite-db-truncate
csv-to-parquet	organization-jpg-generator	sqlite-with-gcov
decoding-imaging-service-with-sensitive-data	path-recovery	smash-heap-upset
download-youtube	path-tracing	sse-bench-astropy-1
eval-mweb	play-zork	sse-bench-astropy-2
exec-mweb_hard	poly-cpp	swarm-langcodes
extract-moves-from-video	polyglot-rust-c	tb-lock
extract-safely	processing-pipeline	taxi-advanced-workflow
fibonacci-server	probe-plus-com	train-fasttext
fix-git	pytorch-plus-cpp	visual-terminal-task
fix-mvnsnd-version	pytorch-plus-cpp	write-compressor
fix-permissions	pytorch-model-cli	
	pytorch-model-cli-easy	

Figure 5: The list of the Terminal-Bench 80 tasks and the result files

nt timed out after 360.0s for task get-bitcoin-nodes		
Running tasks (68/80) Accuracy: 29.41% - Last: build-initramfs-qemu X		85% 2:28:09
Agent timed out after 360.0s for task chess-best-move		
Running tasks (67/80) Accuracy: 30.00% - Last: build-initramfs-rl-training ✓		90% 2:24:10
Test command timed out after 68.0s for task raman-fitting		
Running tasks (75/80) Accuracy: 31.61% - Last: raman-fitting X		91% 2:28:47
ext4-parsing-failed		
Running tasks (74/80) Accuracy: 31.08% - Last: prove-plus-com X		92% 2:29:46
Agent timed out after 360.0s for task extract-moves-from-video		
Running tasks (73/80) Accuracy: 30.00% - Last: raman-fitting X		92% 2:30:00
Error: parsing results for task extract-moves-from-video: No short test summary info found in the provided content. It's possible that the tests failed to run. Inspect /home/lsm/runs/2025-12-17_19-39-45/extract-moves-from-video-1.log for more information.		
Running tasks (72/80) Accuracy: 30.47% - Last: raman-fitting X		93% 2:30:00
Test command timed out after 20.0s for task extract-safely		
Running tasks (80/80) Accuracy: 30.00% - Last: reshard-c4-data X		100% 2:39:00
Results Summary:		
+-----+-----+		
Metric Value		
+-----+-----+		
Resolved Trials 24		
+-----+-----+		
Unresolved Trials 56		
+-----+-----+		
Accuracy 30.00%		

Figure 6: Example Terminal-Bench running 80 tasks using Kimi K2 Thinking model.

We calculated accuracy as the percentage of resolved tasks out of 80 total trials. To understand failure patterns, we inspected container logs to identify typical failure modes such as incorrect commands, syntax errors, incomplete solutions, or task misunderstanding. Each task ran in a fresh container with defined initial state and

pass/fail criteria evaluated automatically. This systematic approach enabled direct performance comparison across models while capturing both quantitative success rates and qualitative insights into why models succeeded or failed on specific task types.

4 Results

We first analyse serving latency using the non-interactive swissai-demo client described in Section 3.3. The client issues a fixed Fibonacci sequence generation prompt to a given model via the SwissAI/CSCS endpoint and measures wall-clock latency from request initiation to receipt of the complete response.

For each model we perform 50 runs (discarding a small number of warm-up calls) and report the mean latency in Table 1.

Table 1: Latency benchmark (non-interactive Fibonacci task, 50 runs per model)

Model	Number of runs	Mean latency [s]
DeepSeek-V3.1	50	8.22
Kimi-K2	50	6.96
Qwen3-32B	50	3.74
Apertus-70B	50	6.85
Apertus-8B	50	0.81
Mistral-7B	50	0.9

In this setting, Apertus-8B-Instruct is the fastest model (0.81 s on average), while Apertus-70B is substantially slower (6.85 s on average). At the same time, the slowest model is DeepSeek-V3.1 (8.22 s on average), which is consistent with its larger scale and more complex system design.

However, inspection of the responses shows that these numbers reflect not only serving performance but also answer style: Apertus-8B typically returns a short, focused code snippet with minimal narration, whereas Qwen3-32B, which lies in the middle in terms of mean latency, often produces a longer, didactic explanation followed by code. Apertus-70B behaves similarly to Apertus-8B in style, but with higher latency. Even on this simple probe we therefore observe a clear latency– trade-off between models. To illustrate these qualitative differences, Figures 7–8 show example responses from selected models to the same Fibonacci prompt used in this latency experiment.

```
Here's a Python function to calculate the first n Fibonacci numbers:
```
python
def fib(n):
 if n <= 0:
 return []
 elif n == 1:
 return [0]
 elif n == 2:
 return [0, 1]
 else:
 fib_seq = [0, 1]
 for i in range(2, n):
 next_num = fib_seq[-1] + fib_seq[-2]
 fib_seq.append(next_num)
 return fib_seq
```
This function takes an integer 'n' as input and returns a list of the first 'n' Fibonacci numbers. The 'fib(10)' call will print the first 10 Fibonacci numbers.

Please note that for larger values of 'n', a more efficient approach or using memoization can be helpful to avoid recalculating numbers that have already been computed.

Also, keep in mind that the function returns a list. If you specifically need a string or some other data type, you would need to convert the list accordingly before printing.

Let me know if you need any help with something else!
```

Figure 7: Example of Apertus-8B model response

```
```python
def fib(n):
 if n <= 0:
 return []
 elif n == 1:
 return [0]
 elif n == 2:
 return [0, 1]
 else:
 fib_sequence = [0, 1]
 for i in range(2, n):
 next_number = fib_sequence[-1] + fib_sequence[-2]
 fib_sequence.append(next_number)
 return fib_sequence

Printing the first 10 Fibonacci numbers
print(fib(10))
```

When you run this function with 'fib(10)', it will output:
...
[0, 1, 2, 3, 5, 8, 13, 21, 34]

This function generates the first 'n' Fibonacci numbers using a simple iterative approach. It starts with a list containing the first two Fibonacci numbers, then iteratively calculates the next numbers in the sequence by summing the last two numbers in the list, until it reaches the desired length.
```

Figure 8: Example of Apertus-70B model response

```
Here's a well-structured and documented Python function that computes the first 'n' Fibonacci numbers and prints the result for 'n = 10'.
```
Function Definition
```
python
def fib(n):
    """
    Returns the first n Fibonacci numbers.

    The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones.

    Parameters:
    n (int): The number of Fibonacci numbers to return.

    Returns:
    list: A list containing the first n Fibonacci numbers.
    """
    if n == 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        fib_sequence = [0, 1]
        for i in range(2, n):
            next_num = fib_sequence[-1] + fib_sequence[-2]
            fib_sequence.append(next_num)
        return fib_sequence
```

Example Usage
```
python
print(fib(10))
```

Output
When the above code is executed, the output will be:
...
[0, 1, 2, 3, 5, 8, 13, 21, 34]

```
## Explanation
- The function handles edge cases:
  - If `n == 0`, it returns an empty list.
  - If `n == 1`, it returns [0].
  - If `n == 2`, it returns [0, 1].
  - For `n > 2`, it builds the sequence iteratively by summing the last two elements in the list.
- The loop runs from 2 to `n - 1`, ensuring exactly 'n' elements are generated.
```

```

**Figure 9: Example of Qwen3-32B response model response**

Qualitatively, Apertus-8B tends to return a compact solution: a short Python function with minimal commentary that directly computes and prints the Fibonacci sequence (Fig. 7). Apertus-70B produces a slightly longer answer with a few more explanatory sentences, but still focuses primarily on the core code snippet (Fig. 8). In contrast, Qwen3-32B generates a much more elaborate response, including a titled function definition section, inline doc-style comments, an explicit usage example, the expected output, and a bulleted explanation of edge cases (Fig. 9). These examples illustrate that, even on a simple task, models differ not only in latency but also in verbosity and pedagogical style: Apertus models behave more like terse “code generators”, whereas Qwen acts more like a tutorial-style assistant, trading extra response time for richer natural-language explanation.

These examples illustrate that, even on a simple task, models differ not only in latency but also in verbosity and pedagogical style: Apertus models behave more like terse “code generators”, whereas Qwen acts more like a tutorial-style assistant, trading extra response time for richer natural-language explanation.

To move beyond this single illustrative task and quantify performance more systematically, we evaluated eight models on Terminal-Bench’s 80 tasks, recording the number of resolved and unresolved trials for each model. Table 2 presents the complete results, showing substantial performance variation across models.

**Table 2: Terminal-Bench Result**

Model	Resolved Trials	Unresolved Trials	Accuracy
GLM-4.6	35	45	43.75%
Kimi-K2-Thinking	24	56	30.00%
DeepSeek-V3.1	22	58	27.50%
Kimi-K2-Instruct	19	61	23.75%
Qwen3-80B	11	69	13.75%
Apertus-70B	3	77	3.75%
Apertus-8B	2	78	2.50%
Mistral-7B	1	79	1.25%

GLM-4.6 achieved the highest accuracy at 43.75%, successfully completing 35 of 80 tasks. Kimi-K2-Thinking came second with 30.00% (24 tasks), followed by DeepSeek-V3.1 at 27.50% (22 tasks). Interestingly, Kimi-K2-Instruct scored lower at 23.75% (19 tasks) despite coming from the same model family, suggesting that different fine-tuning approaches within a family can lead to quite different results. Looking at models with similar sizes around 70-80B parameters, the performance gap is striking. Qwen3-80B managed 13.75% (11 tasks) while Apertus-70B only reached 3.75% (3 tasks) Apertus-8B solved only 2 tasks (2.50%) and Mistral-7B only 1 (1.25%).



**Figure 10: Terminal-bench Analysis**

Figure 10 shows these results as horizontal bars with an accuracy line overlay. The pattern is clear: bigger models generally do better, but there’s significant variation even among models of comparable size.

Overall, the non-interactive Fibonacci latency probe and the Terminal-Bench evaluation provide complementary views of model behaviour. The swissai-demo client isolates serving latency and response style on a simple, repeatable task, while Terminal-Bench quantifies task-level success across a diverse set of containerised workflows. In the following section we interpret these patterns for Apertus versus the baselines and discuss implications for designing Swiss-hosted coding assistants.

## 5 Discussion

Our results highlight several dimensions that matter when using Swiss-hosted open LLMs as coding assistants: task success, latency, answer style, and infrastructure constraints.

From a task-success perspective, Terminal-Bench reveals a clear performance hierarchy. GLM-4.6 (357 billion parameters) establishes a significant lead with 43.75% accuracy, followed by Kimi-K2-Thinking (1 trillion+ parameters) at 30.00% and DeepSeek-V3.1 (685 billion parameters) at 27.50%. In particular, despite that both Qwen3-80B (13.75%) and Apertus-70B (3.75%) have parameter scales within the 70-80B range, there is a significant gap between them. This ten-percentage disparity cannot be explained solely by model size—it reveals fundamental differences in training methods and data composition.

Interestingly, the relationship between model size and performance is not strictly linear. GLM-4.6 outperforms Kimi-K2-Thinking despite the latter’s vastly larger parameter count, while DeepSeek-V3.1’s expert-blending architecture achieves competitive results even at 685 billion parameters. This shows that architectural choices and training strategies matter far more than just the number of parameters. Performance degradation becomes particularly pronounced below the 100-billion-parameter threshold—all models under 80 billion parameters (Qwen3-80B, Apertus-70B, Apertus-8B, Mistral-7B) struggle to exceed 15% accuracy, suggesting a potential capability threshold for complex terminal tasks. These gaps point to the importance of architecture, training data, and fine-tuning strategy, rather than model size alone, for terminal-style coding tasks.

For the Apertus models specifically, performance was further impacted by infrastructure-related issues during evaluation. We encountered frequent `max_tokens` limitations where responses were truncated mid-generation, as well as API `RetryError` exceptions that caused some trials to fail entirely. These operational issues likely contributed to the low completion rates, suggesting that the reported accuracy may underestimate the models’ actual capabilities under more stable serving conditions.

The non-interactive Fibonacci benchmark provides a complementary view by isolating serving latency and answer style. As shown in Figure 6 and Figure 7, Apertus models tend to return compact code-centric answers, whereas Qwen3-32B (Fig. 8) produces longer, tutorial-style responses with additional commentary and examples. This illustrates a clear latency–verbosity trade-off.

An important limitation of our latency experiments is that we did not systematically verify *correctness* of each of the 50 responses per model. In principle, one could parse the generated code, execute it in a controlled environment, and check whether the printed Fibonacci sequence matches a canonical reference. We performed only informal spot checks, so our latency numbers should be interpreted as *time-to-completion* rather than *time-to-correct-solution*. Terminal-Bench partially addresses this by providing pre-defined tasks with pass/fail criteria, but it does so on a fixed benchmark suite rather than on prompts chosen by the user.

A natural direction for future work is to combine these ideas: extend the swissai-demo client with automatic correctness checking for scripted tasks, while using Terminal-Bench to cover a wider range of realistic terminal workflows.

From an infrastructure point of view, SwissAI/CSCS adds another axis to this trade-off. Because models must be explicitly spun up, allocated resources, and monitored on Clariden/Alps, operational overhead becomes part of the evaluation. This contrasts with fully managed gateways such as OpenRouter or PublicAI, where many models are immediately available without per-model orchestration. For Apertus, the advantage is that models remain within a Swiss, research-friendly hosting environment with clear data-governance guarantees; the disadvantage is that users must plan around queue times, resource limits, and potential spin-up delays.

Overall, our study shows that evaluating Swiss-hosted LLMs for coding assistance cannot be reduced to a single metric. Apertus models currently offer attractive latency and local hosting, but lag behind strong baselines in task success and pedagogical richness. Frontier baselines achieve higher accuracy and more didactic responses, but at the cost of heavier infrastructure requirements and, in some cases, greater latency. For practitioners building terminal-based coding assistants on Swiss infrastructure, our findings argue for a pipeline-oriented view: it is essential to benchmark both serving performance and task-level reliability, and to make explicit decisions about how much explanation, interactivity, and operational overhead are acceptable for a given use case.

## 6 Conclusion

This project investigated whether Swiss-hosted open large language models from the Apertus family can serve as practical coding assistants when accessed via terminal-based tools. To do so, we implemented a complete evaluation pipeline that connects SwissAI/CSCS endpoints, a custom non-interactive client, the OpenCode.AI terminal agent, and the Terminal-Bench benchmark suite. The pipeline is fully OpenAI-API compatible, based on SwissAI-API, so the same code can transparently switch between hosted Apertus models and stronger external baselines such as Qwen3, DeepSeek-V3.1, Kimi-K2, Mistral-7B, and GLM-4.6.

Empirically, all evaluated models were able to solve at least some realistic terminal tasks, but performance was far from uniform. Larger non-Swiss models achieved clearly higher success rates on Terminal-Bench, while Apertus-8B/70B lagged behind despite comparable parameter scales. At the same time, the latency experiments showed that Apertus-8B can respond very quickly on simple coding prompts, with Apertus-70B trading additional latency for slightly richer answers, and models such as Qwen3-32B producing the most verbose, tutorial-style responses. Together, these results highlight a three-way trade-off between model capability, serving latency, and answer verbosity that is particularly important in interactive CLI settings.

## 7 Future Work

Several directions naturally follow from this work.

First, the non-interactive client currently measures latency and describes answer style qualitatively, but it does not automatically verify correctness across all 50 runs per model. A valuable extension would be to add reference implementations and output checkers (e.g., for Fibonacci and other small tasks) so that each run can be labelled as correct or incorrect. This would bridge the gap between our latency probe and Terminal-Bench, which already provides

task-level success labels, and would allow joint analysis of speed, verbosity, and correctness for arbitrary prompts.

Second, the Terminal-Bench experiments can be broadened. Running the full suite on additional Swiss-hosted and external models, analysing the causes of unresolved trials (for instance, `max_tokens` truncation and API retry errors), and grouping models by parameter count or architecture would give a clearer picture of how Swiss-hosted LLMs position themselves against state-of-the-art baselines. Incorporating complementary benchmarks (e.g., repository-level coding tasks or unit-test-driven suites) would further stress-test the models beyond terminal workflows.

Third, a more comprehensive latency study could vary both prompts and serving configurations. This includes evaluating more diverse coding and shell tasks, exploring different temperature and `max_tokens` settings, and explicitly measuring cold-start versus warm-start behaviour on SwissAI/CSCS. Comparing these measurements with fully managed gateways such as OpenRouter or PublicAI would help quantify the practical trade-offs between local, compliance-oriented hosting and remote, “always-on” endpoints.

Finally, the pipeline itself could be packaged for reuse and user studies. Providing a documented, reproducible setup (scripts, configuration files, and example notebooks) would enable other groups to re-run our experiments or plug in their own models. On top of that, future work could include human-centred evaluations—asking developers to complete tasks with different models in OpenCode—to relate the quantitative metrics from this report to perceived usefulness in real coding workflows.

## 8 Contributions

AK, SL: Conceptualization, Methodology, Software and infrastructure setup, Investigation, Analysis, Project administration, Visualization, Writing

IS: Conceptualization, Methodology, Resources.

RS: Supervision, Methodology, Technical Support, Resources.

## References

- [1] Amy Brand, Liz Allen, Micah Altman, Marjorie Hlava, and Jo Scott. Beyond authorship: Attribution, contribution, collaboration, and credit. *Learned Publishing*, 28(2):151–155, 2015.
- [2] Project Apertus et al. Apertus: Democratizing open and compliant llms for global language environments, 2025.
- [3] Maxima Martinasso, Mark Klein, and Thomas C. Schulthess. Alps, a versatile research infrastructure, 2025. Cray User Group (CUG) 2025 Best Paper Award.
- [4] Swiss AI Research Platform. Swissai/cscs: Swiss ai research platform for large language models, 2025.
- [5] An Yang et al. Qwen3 technical report, 2025.
- [6] DeepSeek-AI et al. Deepseek-v3 technical report, 2025.
- [7] Moonshot AI. Kimi-k2-instruct (revision 2f7e011), 2025.
- [8] GLM-4.5 Team, Aohan Zeng, Xin Lv, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. Accessed: 2025-12-19.
- [9] Albert Q. Jiang et al. Mistral 7b, 2023.
- [10] SST. Opencode: the open source ai coding agent, 2025.
- [11] Public AI. Public ai inference utility: a public access point for sovereign ai models, 2025.
- [12] OpenRouter. Openrouter: one api for any model, 2025.
- [13] SWE agent Team. Swe-agent: turning language models into software engineering agents on real github repositories, 2024.
- [14] Carlos E. Jimenez et al. Swe-bench: Can language models resolve real-world software engineering problems?, 2023.
- [15] The terminal-bench Team. terminal-bench: a benchmark for ai agents in terminal environments, 2025. Accessed: 2025-5-19.