Three steps for OCaml to crest the AI humps

Sadiq Jaffer[1], Jon Ludlam[1,2], Ryan T. Gibb[1], Thomas Gazagnaire[2] and Anil Madhavapeddy[1,2]
Department of Computer Science & Technology
[1] University of Cambridge
United Kingdom
[2] Tarides, France

Abstract

We discuss how OCaml could adapt to the fast-moving world of AI-assisted agentic coding. We first benchmark how well represented OCaml is in the large and diverse set of open weight models that can be run locally. We then consider what is unique about OCaml programming (in particular, modules and abstraction) that differentiates it in this space. We then consider the changes required in our ecosystem to work better with AI coding assistants.

1 Introduction

The rapid evolution of AI-assisted programming presents both opportunities and challenges for the OCaml community. Large language models (LLMs) have become increasingly capable of code generation, especially in the past year. This paper focuses on our efforts to understand how to position OCaml within this new landscape. Since this is an exceptionally fast-moving field, the specific model performance metrics and capabilities presented in this paper will likely be outdated by the time of the OCaml Workshop. However, the basic approach and questions that we raise about OCaml's integration with AI tooling remain crucial for a collective community conversation, and we will do our best to keep them updated. We pose these questions:

1. How well do locally-runnable AI models handle OCaml code generation?
2. What differentiates OCaml coding vs other languages?
3. Should OCaml develop new tools for agentic integration?

Many broader questions are being raised about the licensing legalities of these code models, the environmental footprint of their use, and the broader impact on the evolution of existing programming languages. We acknowledge these debates, but do not directly address them in this paper.

2 Three Questions for OCaml and AI

Our aim with these questions is to establish a framework for ongoing community discussion and action in this fast moving space, and to make our benchmarks openly available.

2.1 How well do locally-runnable AI models handle OCaml code generation?

A crucial step in evaluating AI models is first determining your use case and then assembling a private benchmark which reflects it. In our case, we seek to understand how well existing models perform with models that can run locally or on a shared local inference server. This is important for any use of OCaml in a teaching environment, where the provenance, tuneability, predictability and reproducibility of the environment are crucial to good learning outcomes.

As a benchmark dataset, we used the OCaml tutorial problems that first year Computer Science students at the University of Cambridge tackle in our "Foundations of Computer Science" course. These are interactive Jupyter notebooks where students are given a series of standalone OCaml problems related to beginner computer science, and they populate answer cells with their solutions being machine checked. An example given in figure 1. Each tutorial is known as a "tick" and has three or more questions. Students must complete earlier questions in order to progress to later questions. There are also "starred" ticks which are stretch goals and of higher difficulty. The benchmark consisted of 10 ticks with a total of 33 questions.

Since these problems require no use of the OCaml standard library or modules, they form a minimum bar for being able to produce more complex OCaml code that make use of advanced language features or ecosystem libraries.

Methodology. For evaluation, we attempted each question 3 times for each model and conducted 5 repetitions per model per exercise set. Models were accessed via the OpenRouter inference marketplace. The study encompassed a diverse range of models with varying parameter counts:

Model Family	Parameter Counts
Google Gemma3	12B, 27B
Deepseek-R1-Llama	70B
Meta Llama-3.1	8B, 70B
Microsoft Phi-4	14B
Mistral	8B, 12B (Nemo), 24B (Small)
Qwen2.5	7B, 72B (Coder variants)
Qwen3	8B, 14B, 32B, 30B-A3B[1]

Performance Results. There were significant performance variations across models (Figure 2). Qwen3-32B in thinking mode (94.2%) approached the same levels of performance as the frontier (closed-weight, not self-hostable) Claude-3.7 Sonnet model (96.4%). Qwen3 was also close to 50x more cost-effective than Sonnet via the OpenRouter API. Within a model family larger parameter counts resulted in better performance. Between model families and architectures there

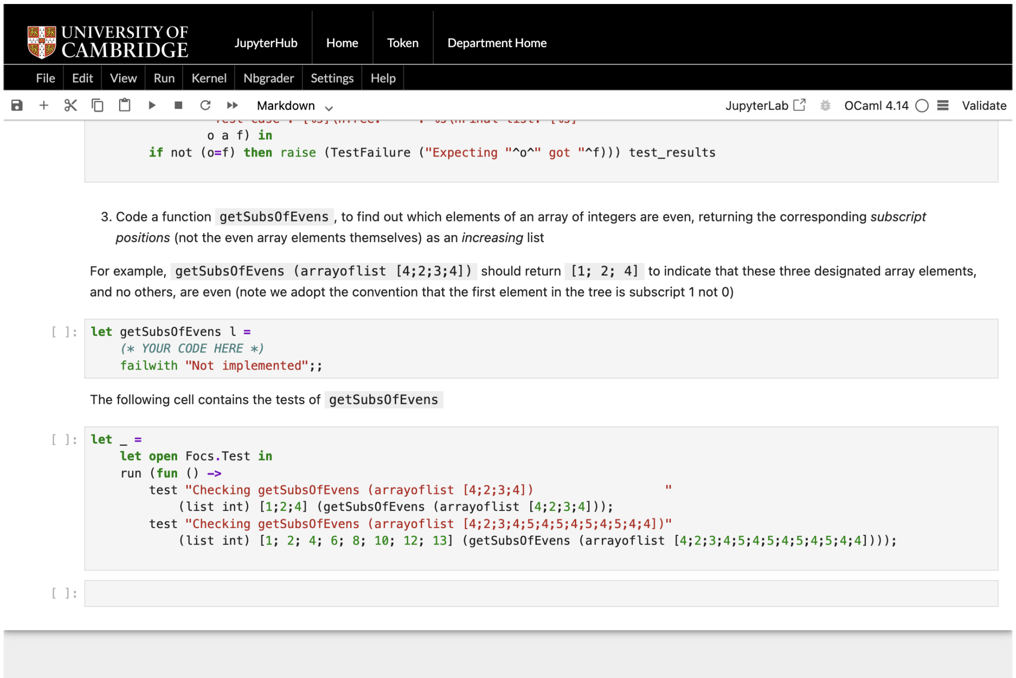

Figure 1. Example of the interactive Jupyter notebook used by students

was significant variation in performance even for similar parameter counts.

Thinking Mode Advantage. Some models have a "thinking mode", where models show their reasoning process before answering, consistently outperformed their standard counterparts. For instance, Qwen3 32B improved from 62.4% to 95.2% when thinking mode was enabled. Whilst this leads to a significant performance improvement, it comes at the cost of increased latency and expense.

Qwen 3 models performed very well. The best performing model was qwen3-32b in thinking mode at 95.2%. This is very close to Anthropic's Claude 3.7 Sonnet with thinking. The Qwen3 32B model is Apache2 licensed and so can be self-hosted. In addition, all Qwen3 models in non-thinking mode outperformed comparably-sized models from other families. While not in the Qwen3 family, the Qwen2.5-Coder-32B model was the best non-thinking model (77.6%). In every case allowing Qwen3 to think improved performance. The qwen3-8b model with thinking on was very close in performance to llama-3.3-70b, a model almost an order of magnitude larger. As noted earlier, this comes at the expense of increased latency and cost. In most cases, thinking added 2-3k tokens of reasoning before the model produced its final answer; for Qwen3 32B on OpenRouter that is roughly a minute of thinking for an average model. There were cases where the model entered a loop and kept reasoning until it hit the token limit. Models employing reasoning are more likely to be used for asynchronous coding agents *vs.* latency-sensitive completion tasks.

What Went Wrong in the OCaml code? Code generated by models exhibited consistent failure patterns: syntax errors (missing `rec` keywords, incorrect operators), type confusion (mixing integer/float operations), failing to parenthesise nested match statements, and hallucination of nonexistent functions (e.g., `List.sub`, assuming the Jane Street Core library had been opened earlier in the module).

The dramatic improvement with thinking modes indicates that step-by-step reasoning significantly enhances OCaml code generation, possibly compensating for limited OCaml representation in training datasets.

Future Benchmarking Directions. These are only introductory tasks aimed at beginning Computer Science students, and so are not representative of the complexity of larger OCaml codebases. To properly assess AI models capabilities on OCaml tasks, we need comprehensive benchmarks that test more language features and typical developer workflows such as: *(i)* multi-module project completion with complex build configurations; *(ii)* advanced type system usage (GADTs, first-class modules, polymorphic variants); *(iii)* PPX and preprocessor integration tasks; and *(iv)* adding features to existing large OCaml codebases.

Mining public open source projects codebases has yielded large benchmarking codebases for languages like Python [4, 8] and Java[5]. The relatively small number of public OCaml

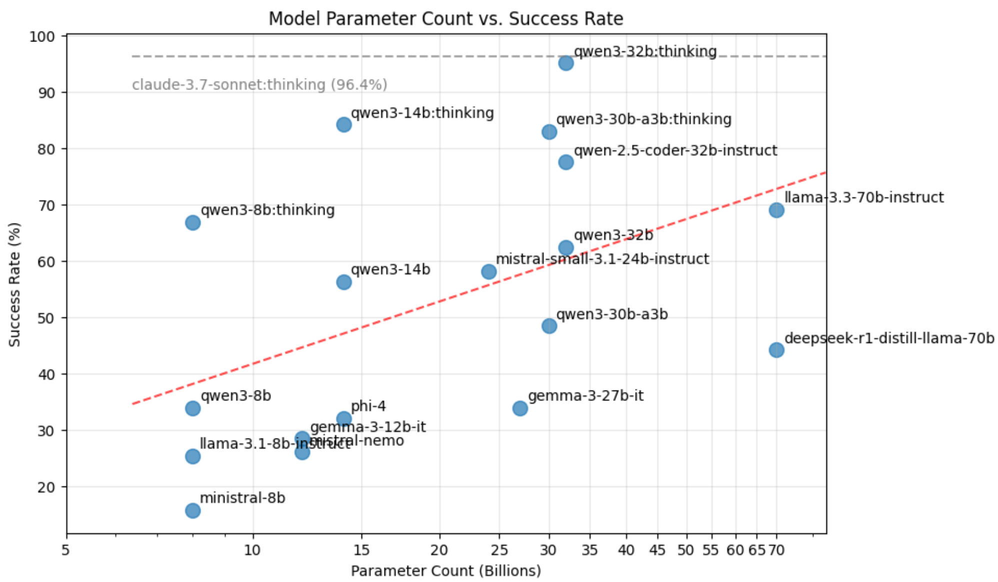

Figure 2. Performance of language models on OCaml exercises plotted against model parameter count (log scale). The graph shows a general trend of improved performance with larger models, with notable outliers in the Qwen3 family achieving superior performance at lower parameter counts. Note also the frontier (closed-weights) Claude-3.7 Sonnet model in grey.

projects likely require using synthetic defect generation approaches [9] to obtain suitable coverage of the language and tooling.

2.2 What differentiates OCaml agentic coding?

OCaml's module system and approach to separate compilation provides an opportunity to adopt a rather different approach to agentic programming. The benchmark task that we use in this question is to begin with a textual Request For Comments (RFC) that defines a protocol, and then attempt to synthesise a working OCaml implementation that can communicate using that protocol (e.g. JMAP[2], MCP[3] or FastCGI[4]). In an OCaml project, this development can occur in three separate phases.

Interface file generation. Firstly, a code agent can be prompted to generate *only* the `mli` files for a library. When the agent cannot find an appropriate external type, it simply leaves it abstract. In this phase, the interface structure of the library is established across multiple separate interface files and potentially even libraries. Crucially, it is possible to type-check these for consistency without having an implementation, for example via `dune build @check`. This phase fixes type errors and allows for odoc HTML documentation for human checks on whether the interface is correct.

Test generation with linking errors. While the interface above may type check, it may still leave too much abstract for it to be a usable interface. This phase generates binary executables that link against the library above, and exercise the actual usage of the library. The binaries generated should pass type checking, but will still fail with a linking error due to a missing library implementation. The model can be prompted to view the linking error as "success" in this phase, with any type errors needing to be corrected. By the end of this phase, the interface proposed for implementation should be verified as broadly acceptable by the human programmer.

Implementation generation. Now that the library interface and test cases both typecheck, the agent can be prompted to proceed to generate implementations for each interface file, one at a time. This constrains code generation errors to just one `ml` module, and makes unit test generation much more reliable. It is also straightforward to prompt the agent to generate module-specific test cases to ensure code coverage, which would be more difficult with dozens of files being generated in one pass.

OCaml's abstraction edge. This approach for OCaml code generation is in contrast with dynamically-typed languages where AI models must load significant proportions of the codebase into the context window in order to reason about implicit interfaces. Models can also generate plausible-looking code that may fail at runtime. In OCaml, the type checker acts as a gatekeeper, rejecting syntactically correct but semantically incorrect suggestions. This further relieves the AI model and user of needing to generate tests which check for adherence to existing interfaces. The fast compilation speed of the OCaml compiler and support for separate

[2]https://tangled.sh/@anil.recoil.org/ocaml-jmap
[3]https://tangled.sh/@anil.recoil.org/ocaml-mcp
[4]https://tangled.sh/@anil.recoil.org/ocaml-fastcgi

compilation also makes the generate-compile-repair iterative loop a more practical prospect.

The specifics of how to generate prompts for this workflow are still changing rapidly, as it depends on how the agentic environment sets up the model context. Not needing to fully load implementations into the context window is generally proving beneficial, as recent results [10] indicate that frontier models struggle to make use of their long context capabilities, so minimising context use is beneficial.

This OCaml "types-first" approach described here is what we traditionally teach our students to do when writing OCaml code, but an informal survey of the local programmers revealed that this is not widely adopted practise!

2.3 Should OCaml develop new tools for agentic integration?

Relative to mainstream languages, there is little OCaml code in pre-training corpuses. The open code pre-training corpuses in "The Stack v2" [7] consists of no more than 0.003% OCaml code by size, and this is after incorporating the synthetically generated OCaml dataset from MultiPL-T [1]. Given this, we cannot rely solely on model parametric knowledge of OCaml as we might with Python or JavaScript.

However, there are several other ways we could support coding models, especially those embedded in agentic workflows that can execute external tools.

LLM-friendly documentation. Rather than depending on parametric model memory (which is only as up-to-date as when the model was trained), we instead should provide access to LLM-friendly formats that minimise context usage. This typically means using simple textual formats such as Markdown rather than HTML or PDFs. The `llms.txt` [3] project proposes having a convention where Markdown files should be located. There are also efforts underway[5] to ensure odoc can emit Markdown directly. This can then be used in `ocaml-docs-ci`, the system that currently generates the central ocaml.org documentation for all versions of all packages in opam, to produce Markdown documentation in addition to the current HTML.

Command-line error prompting. Continuing efforts to improve the error messages of the OCaml tooling will benefit both humans and LLMs. For example, errors could provide links to relevant resources and documentation as part of the output, which can be consumed by agents as a "next-step" hint.

Model-Context Protocol (MCP). The MCP protocol is emerging as the standard for agents to perform external function calls to tools to navigate, understand, and manipulate OCaml codebases. The OCaml ecosystem would benefit from hosting the following MCP integrations, some of which can be run centrally, while others should be run locally:

- Natural-language search and navigation of OCaml libraries
- Merlin integration for type throwback, jump-to-definition to help navigate the code, "find occurrences" to locate uses of types and values and other tools
- Integration with build systems like Dune and ocamlbuild
- Sherlodoc-powered search, both generic, searching the latest versions of packages, and specific, searching the versions used by your current project
- Error message interpretation and fix suggestions
- Module system navigation and interface extraction
- Provide support when PPXs are in use, by giving a concise guide to the PPX to the LLM, or by showing the post-processed code

These tools would bridge the gap between generic AI capabilities and OCaml-specific development workflows by invoking real tools within the inference flow, rather than guessing from the parametric model knowledge.

Embedding the opam ecosystem. The models would also benefit from being able to figure out when to use an existing ecosystem library rather than coding it up from scratch. To this end, we have been prototyping[6] an approach by producing natural language summaries of all OCaml modules on opam. We recursively combine summaries up to the library level, and then provide tools that enable AI models to search for specific functionality across the opam universe.

This package embedding approach offers a promising direction for improving OCaml's discoverability in AI systems. By creating high-quality embeddings of OCaml packages, documentation, and code examples, we can enhance model understanding and generation capabilities. This approach would require systematic indexing of the entire OCaml ecosystem, creating searchable repositories that AI models can reference during code generation. Such infrastructure would address the current limitation where models struggle to find appropriate OCaml libraries and idioms. However, questions of user agency and licensing are also key to ensuring that such large-scale scraping does not split our community, and that clear opt-out mechanisms are present for those who do not wish to participate (for any reason).

Improving integration with external dependencies. OCaml applications do not exist in a vacuum, and often bind to external libraries written in other languages. We have also been investigating a more systematic approach to dependency management *across* multiple package managers [2]. By creating a dynamically updated embedding across Linux distributions, language-specific package managers and adhoc code repositories, we hope to empower AI models to make coding architecture decisions that do not duplicate the hard work already committed to version control. This

[5] See https://github.com/ocaml/odoc/pull/1341 for odoc PR#1341

[6] https://github.com/sadiqj/odoc-llm

code reuse should increase the overall code quality of AI-driven code generation, and also reduce the environmental cost of inference by reducing the amount of code generation required.

3 Conclusions

For many aspects of agentic coding, functionality that is useful for agents is also often useful for humans, and vice versa. For example, pure functional interfaces combined with effectful layers [6] make reasoning about logic and managing context better for both LLMs and humans. This extends to tooling as well; building tools that are easy and useful for agents to use looks a lot like building tools that are easy and useful for humans to use. With the efforts described in this paper, we hope to see a coevolution in OCaml tooling towards AI-assistance improving the overall quality of code, and not drowning in "AI slop".

References

[1] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 295 (Oct. 2024), 32 pages. doi:10.1145/3689735

[2] Ryan Gibb, Patrick Ferris, David Allsopp, Michael Winston Dales, Mark Elvers, Thomas Gazagnaire, Sadiq Jaffer, Thomas Leonard, Jon Ludlam, and Anil Madhavapeddy. 2025. Solving Package Management via Hypergraph Dependency Resolution. arXiv:2506.10803 [cs.SE] https://arxiv.org/abs/2506.10803

[3] Jeremy Howard. 2024. The /LLMS.TXT file – LLMS-txt. https://llmstxt.org/

[4] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[5] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[6] David Kaloper-Mersinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-Quite-So-Broken TLS. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 223–238. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kaloper-mersinjak

[7] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[8] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training Software Engineering Agents and Verifiers with SWE-Gym. In *ICLR 2025 Third Workshop on Deep Learning for Code*. https://openreview.net/forum?id=lpFFpTbi9s

[9] Minh V. T. Pham, Huy N. Phan, Hoang N. Phan, Cuong Le Chi, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. SWE-Synth: Synthesizing Verifiable Bug-Fix Data to Enable Large Language Models in Resolving Real-World Bugs. arXiv:2504.14757 [cs.SE] https://arxiv.org/abs/2504.14757

[10] Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. LongCodeBench: Evaluating Coding LLMs at 1M Context Windows. arXiv:2505.07897 [cs.CL] https://arxiv.org/abs/2505.07897