Guarding LLM-aided Software Transformation Tasks via Component Exoskeletons

Evangelos Lamprou
vagos@lamprou.xyz
Brown University
USA

Christian Gram Kalhauge
chrg@dtu.dk
Technical University of Denmark
Denmark

Martin C. Rinard
rinard@csail.mit.edu
Massachusetts Institute of Technology
USA

Nikos Vasilakis
nikos@vasilak.is
Brown University
USA

Abstract

Large language models (LLMs) are achieving state-of-the-art results across a wide variety of software transformation tasks—including translating across languages and lifting opaque software components to high-level languages. Unfortunately, their results are often subtly incorrect, insecure, or underperformant—affecting the widespread deployment of these LLM-driven techniques in settings that go beyond the narrow scope of academic papers. This paper posits that such widespread deployment crucially depends on developing appropriate model guardrails for safeguarding the results of the transformation process. Such guardrails can be supported by *component exoskeletons*, tunable partial specifications extracted mostly automatically from the original, pre-transformed component. Exoskeletons serve as component projections that supplement, and often go through, the entire transformation process, confirming that the new, transformed component meets the original specifications. They show promise on several real-world scenarios and unearth exciting research directions.

CCS Concepts: • **Software and its engineering** → **Software development techniques**.

Keywords: large language models, software transformation, component exoskeletons

ACM Reference Format:
Evangelos Lamprou, Christian Gram Kalhauge, Martin C. Rinard, and Nikos Vasilakis. 2025. Guarding LLM-aided Software Transformation Tasks via Component Exoskeletons. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3766882.3767171

PACMI '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2205-9/25/10
https://doi.org/10.1145/3766882.3767171

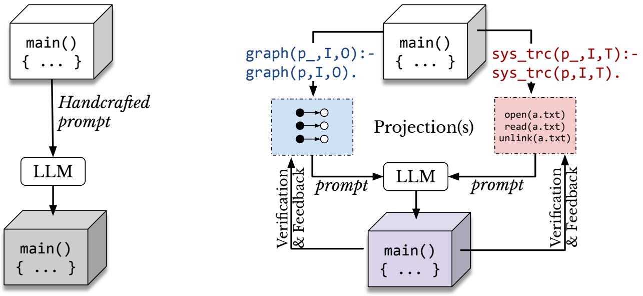

Fig. 1. Exoskeletons as component projections. In a typical LLM-assisted workflow (left), an LLM transforms a component using its source code and a handcrafted prompt. With exoskeletons (right), the component is projected—guided via a query—onto lightweight, verifiable properties (*e.g.*, input-output examples, system-call traces), which guard and support an LLM in producing the transformed component. A verification and feedback loop confirms the result satisfies the query, safeguarding transformations.

1 Introduction

Large language models (LLMs) are used pervasively to assist with software development and evolution tasks [8, 11, 13, 34]. A recent longitudinal study from GitHub indicates that LLMs are used by over 90% of U.S.-based developers [25] among other tasks to translate code between languages [8], refactor complex modules [34], and generate components given a few input-output examples [13].

Unfortunately, the broader adoption of LLMs remains challenging in tasks that target the transformation of software components, including component lifting, lowering, acceleration, optimization, and other improvement tasks. LLM outputs remain heavily influenced by surface-level patterns in their inputs [31], struggle to satisfy structural or negative constraints [14, 17], and often overfit to spurious cues [29, 30]. As a result, they often need steering through *ad hoc* heuristics, trial-and-error, and carefully curated examples [6, 16, 19, 21, 33].

The key thesis underlying this paper is that, as transformation tasks take existing software components inherently into account in order to improve them, LLM outputs can in practice be guarded by verifiable hypotheses extracted directly from the original components. These hypotheses form partial specifications in the form of input-output examples, system-call traces, generated tests, function and data-structure signatures, execution invariants, and other analysis facts that underpin any improvement, translation, transformation, or augmentation of an existing software component towards a new component. We call these partial specifications *exoskeletons*, as they are initially *extracted* from the original component before any improvement or transformation has been applied, they potentially undergo similar transformations, and then they are *applied externally* to the new, improved component—but remain tunable by component-external entities such as their users (§2).

Several insights power exoskeletons. First, the availability of the original components prior to transformation: pre-transformed components can be used as (partial) behavioral references, analyzable using a variety of logical approaches—termed *exoskeleton extractors*—to extract valuable and unambiguous properties about the component and supplant statistical approaches such as LLMs. These properties can be further morphed—via *exoskeleton modifiers*—to fit a target domain or language, or remove unsafe or undesirable aspects of the representation. Second, these properties, viewed as *component projections*, can be fed to one or more (and potentially different) models, independently from the path the original source takes through the model—effectively constraining statistical outputs, minimizing the risk of overfitting to the source representation, and enabling independent checks. And third, due to their inherently logical structure, exoskeletons remain a fine target for declarative human input via declarative task- and domain-specific languages—including logical constraints as shown in this paper.

Concretely, the exoskeleton approach analyzes a component to extract relevant facts and feasible properties informed by a logical query. It allows morphing and filtering of those extracted facts and properties guided by the same query. It then prompts an LLM instance, providing it only with the extracted exoskeletons, to generate a new (potentially transformed) component, which it will then check against these properties. Exoskeletons treat component extraction as a projection of the original component over language- and system-level dimensions of facts, leveraging the LLM's high-dimensional internal representation [12, 18, 26] to lift the projection back to a concrete implementation (Fig. 1). The approach allows for tunable exoskeleton quality based on coverage metrics [32].

The paper presents use cases that illustrate the promise of exoskeletons in software trasnformation scenarios like component hardening, lifting, translating, and offloading (§2), describes the key components of an early exoskeleton prototype alongside preliminary results (§3), and concludes with a discussion of possible directions to scale the presented techniques beyond this paper (§4). For reference, Fig. 2 summarizes key terms used in this paper.

component projection: A component representation via a set of lightweight, verifiable properties (*e.g.*, input-output examples or traces).
component exoskeleton: A tunable, partial specification of a component; it is extracted before transformation, encodes a single invariant, and serves to guard and support the transformation process.
exoskeleton extractor: Any analysis or reasoning that derives verifiable properties from a component to construct its exoskeleton.
exoskeleton modifier: A function that reshapes an exoskeleton to fit a new domain or language, or drops parts of it that are unsafe or undesirable to transfer.

Fig. 2. Glossary. Summary of key terms used in this paper.

```
#define MAX_C 1024
char *Cmd_Args(int argc, char **argv) {
 static char args[MAX_C];
 for (int i = 1; i < argc; i++) {
  strcat(args, argv[i]);
  if (i < argc - 1) strcat(args, " "); }
 return args; }
```

Fig. 3. Source component. A string formatting function that concatenates command-line arguments into a single string.

2 Use Cases

This section considers a variety of transformation tasks, ranging from security hardening to kernel offloading and a variety of target domains, ranging from string manipulation to packet filtering. Using exoskeletons over declarative constraints safeguards the transformation of all components. The outputs shown are drawn from exo, an early exoskeleton prototype under heavy development (§3).

Buffer-overflow hardening: Consider a C utility function that handles string parsing (Fig. 3). It accepts an array of strings, along with its size, and combines its elements into a single string separated by spaces. Such string manipulation code often results in bugs or buffer overflow vulnerabilities [1, 2]. A refactoring may involve hardening this function against such vulnerabilities.

Some realizations: the function is pure, meaning its behavior can be fully described by input-output examples; moreover, if a buffer overflow exists, there must be inputs that trigger it. Below is a query written in the proposed DSL that hardens the function against such inputs. The query requires the regenerated component p_ to exhibit the same input-output behavior on representative inputs, but have non-faulty behavior on inputs that cause a segmentation fault to the original component p.

```
fsignt(p_, S) :- fsignt(p, S).
graph(p_, I, O) :- graph(p, I, O), { crash(p, I); }.
#not crash(p_, I), graph(p_, I, _).
```

The approach starts by generating potential exoskeletons—extracting the function signature S and a set of input-output pairs I, O from the function. The signature is extracted by the fsignt plugin, by parsing the original library's source code, and input-output pairs by the graph plugin, which prompts an LLM instance to generate values that exercise the function. The crash plugin signals graph's input generator to provide inputs that induce segmentation faults, and filters those that actually do. This information around the component populates a fact database, which contains all extracted facts about the component or given by the user through their query. The generated set of input-output pairs and the function's signature are then used to prompt an LLM instance which in turn generates a transformed, behaviorally equivalent component. A verification phase confirms that (1) the input-output behavior of the transformed component on the original set of inputs I remains the same and does *not* crash on the crash-inducing inputs and (2) the function signature is preserved. The relevant fragment of the transformed component looks as follows:

```
// ...
for (i = 1; i < argc; i++) {
  strncat(args, argv[i], remaining);
  remaining = MAX_C - strlen(args) - 1;
  if (i != argc - 1 && remaining > 0) {
    strncat(args, " ", remaining);
    remaining = MAX_C - strlen(args) - 1;
  } }
  return args;
```

Cross-language lifting: Building on the same component, consider the scenario of cross-language translation, where the task is to lift the C component to JavaScript.

The following query specifies that the transformed component p_ must be implemented in JavaScript and must preserve observable behavior over a representative set of inputs.

```
language(p_, js).
graph(p_, Ijs, Ojs) :- graph(p, Ic, Oc),
                       tr(Ic, Ijs), tr(Oc, Ojs).
```

To bridge differences in representation between JavaScript and C, the query includes the tr predicate to convert each input and output into a practically equivalent one in the target language. The corresponding tr plugin invokes a fine-tuned transformer-based model which excels in this domain [27]. Errors are detected during verification when the transformed component is run on Ijs and its outputs compared to Ojs. Based on the query, graph invokes tr to produce corresponding JavaScript representations of each example so that the translated function will reproduce identical string-formatting behavior. Prompting an LLM with the translated input-output pairs and the target language results in the following code:

```
function cmdArgs(argv) {
    return argv.slice(1).join(" ");
}
```

The regenerated component has a slightly different signature than the original, since JavaScript arrays include built-in information about their length, which makes the argc argument obsolete. The synthesis backend had the freedom to make this change since the query did not specify preserving the function's signature.

```
float Q_rsqrt(float number) {
 long i; float x2, y;
 const float threehalfs = 1.5F;
 x2 = number * 0.5F; y  = number;
 i = * ( long * ) &y; // evil bit level hack
 i = 0x5f3759df - ( i >> 1 );
 y = * ( float * ) &i;
 y = y * ( threehalfs - ( x2 * y * y ) );
 return y; }
```

Fig. 4. Source component. An unidiomatic C implementation of the fast inverse square root function popularized by Quake III [7].

Idiomatization: Consider the fast inverse square root routine, originally developed for the Quake III engine [7]. The implementation exploits type punning to manipulate IEEE floats at the bit level—a technique that relies on undefined behavior and reduces portability. The transformation goal here is to replace this implementation with a more idiomatic one. The following query uses lines of code as a coarse metric of idiomaticity, where idiomatic patterns are considered to produce more concise code.

```
graph(p_, I, O) :- graph(p, I, O).
#min N: len(p_, N).
```

The system will regenerate the component, minimizing this metric while preserving observable behavior over a set of nominal inputs. The properties passed to the synthesis engine are then ones extracted from graph and len. In this case, where the goal is to optimize an aspect of the component—its length—the system re-prompts the model to regenerate the component until it can make no further improvements after two consecutive transformations:

```
#include <math.h>
float Q_rsqrt(float number) {
    return 1.0f / sqrtf(number); }
```

The synthesized implementation is easier to verify, portable across compilers, and avoids undefined behavior.

Offloading packet filtering to eBPF: Consider a utility library that filters incoming packets by destination port (Fig. 5). To avoid the overhead of user-kernel crossings, a transformation from a userspace component to one that runs in the kernel using eBPF [28] has the potential to offer significant performance benefits. The query below offloads this function

```
bool filter_packet(const struct packet *pkt) {
  return (pkt->port == 80 || pkt->port == 443); }
```

Fig. 5. Source component. Userspace packet filter.

to an eBPF component attached to the socket.

```
graph(p_, Ipkt, Opkt) :- graph(p, I, O),
                         tr(I, Ipkt), tr(O, Opkt).
ebpf(p_, socket).
```

The graph plugin extracts the input-output behavior of the original component, while tr translates the inputs I and outputs O to kernel-compatible representations. During verification, the ebpf plugin compiles and loads the regenerated component, attaches it to a raw socket using libbpf, and runs it under controlled traffic replay to eventually result in the following code (simplified):

```
SEC("socket")
int filter_prog(struct __sk_buff *skb) {
 // ...
 return (tcp->dest == __constant_htons(80) ||
 tcp->dest == __constant_htons(443)) * pkt_size; }
```

Contrary to plain LLM assistance, the exoskeleton approach compares the results to those of the original user-space filter on the same packet traces, runs the eBPF component against the eBPF verifier to confirm it passes, and updates the application to load and attach the eBPF program.

3 Key Elements & the Exo Prototype

To handle the diversity of transformation tasks and domains in which LLMs are used, an exoskeleton system combines (1) domain- and task-specific extensibility, (2) a high-level declarative language for querying facts about components, and (3) direct LLM interaction through an increasingly constrained feedback loop. The exo prototype implements simple versions of all three elements (Fig. 6), including seven extensions. It has been applied to a small collection of transformation tasks to gather preliminary results.

Key elements: First, an exoskeleton system needs to be decoupled into a core and several extensions collecting facts related to the domain and the task at hand.

These extensions are structured in a way that combines (1) *extraction* which obtains facts about the original component via combinations of static, dynamic, or hybrid analyses (*e.g.*, input-output examples or system-call traces) or transforms an extracted fact to a new equivalent fact about the target component; (2) *checking* that validates the transformed component against task-specific properties by applying the corresponding analyses to the transformed component and comparing the results; and (3) *collection* of all the extracted and confirmed facts into a query-able database for intra- and inter-extension consumption. While the space of possible combinations poses an intractability challenge, the focus on practically significant subsets combined with additional crowdsourcing from several communities is promising, supported by exo's preliminary results.

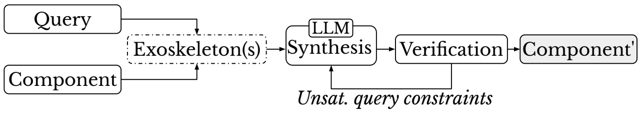

Fig. 6. High-level architecture. A component and a declarative query serve as input. Exoskeletons are extracted from the component, guided by the query, and passed to an LLM-based synthesis engine. After a verification and feedback loop, the engine produces the transformed component.

The database of analysis facts is accessible by a high-level declarative query language injecting reasoning logic that is agnostic to the transformation back-end. This Datalog-like query language allows describing transformation intent as logical constraints over component properties. These queries are resolved against—but also feed back to—the database of extracted facts, representing a space of behaviors that must be preserved, eliminated, or optimized.

Finally, a transformation-and-validation driver interacts with the LLM to explore transformed components that better match the required properties. A failure triggers a retry, augmenting the LLM prompt with additional information about the properties that were not satisfied. The system either converges towards success or issues an error.

The* exo *prototype: These elements inform the design of a preliminary exoskeleton prototype, exo. The exo prototype is developed in Python v3.13, uses pluggy [20] to define pluggable hooks, compiles queries to Clingo [9], interfaces with the clasp solver [10], and uses GPT-4o [23] as the synthesis back-end.

It implements seven pluggable exoskeleton extractors and modifiers written in Python, covering the aforementioned use cases (§2) as well as additional ones seen in the evaluation: graph, tr, len, sys_trc, crash, fsignt, and ebpf. The exo prototype is 364 LoC; its extensions between 9–143 LoC. A more sophisticated prototype is under development.

Preliminary results: A small-scale evaluation indicates feasibility and trade-offs. The evaluation consists of 37 transformation tasks spanning language translation, de-obfuscation, and security hardening (Tab. 1). The target components are drawn from sources such as the Rosetta Code project [24], popular npm and PyPi packages [3, 4], high-profile supply-chain attacks [5, 22], and obfuscated programs from the IOCCC [15]. A transformation is considered successful when the target component (1) satisfies all specified properties, (2) passes all developer tests, and (3) is deemed correct via manual inspection.

Overall, exo succeeds in all but two cases: (1) a utility function that modifies its arguments in-place—something not reflected in its input-output examples and thus not preserved after transformation; (2) an obfuscated prime-checking program whose complex output format was not fully captured,

Tab. 1. Benchmark summary. The table summarizes the number of components (col. 2), their range in lines of code (col. 3), and successes using exo or an LLM-only approach (cols. 4–5).

Benchmark	n	LoC	$\checkmark_{LM}$	$\checkmark_{\text{exo}}$	Purpose	Sources
RosettaCode	16	1–155	16	16	Translation	[24]
Utilities	13	2–577	13	12	Parity	[3, 4]
SSCA	3	30–214	0	3	Purification	[5, 22]
IOCCC	2	13–214	0	1	Lifting	[15]
Other	3	16–280	1	3	multiple	*Cf.* §2
Total	37		30	35		

with the primality test itself correctly regenerated. It succeeds in all other cases, including an impressive supply-chain security case where exoskeletons drive the LLM to push malicious side-effects out of the transformation scope, while preserving legitimate functionality. Runtime ranges between 25.9–64.2s (avg. 43.63s), with 1–3 synthesis attempts (avg. 2). The verification loop caught mistakes related either to incorrect function input/output types, or missed edge-cases.

Compared to `exo`, an LLM-only approach that prompts a model (in the experiments, GPT-4o) with each component's source code and a natural-language description of each task faces significant difficulties beyond simple translations and refactoring. For example, transforming the `leetlog` component [22] maintains a malicious credential exfiltration behavior that looks as if it is part of the intended behavior. On the `Q_rsqrt` component [7], the generated output overfits, transferring details such as bit manipulation to the transformed component.

4 Discussion

Scaling exoskeleton techniques beyond the cases discussed earlier requires solving several challenges.

A transformation system supported by component exoskeletons is unable to offer partial success in cases when no satisfactory candidates have been found. While all-or-nothing semantics is useful in cases at the "all" end of the binary spectrum, degradation might not be graceful enough for practical LLM scenarios.

Going beyond pure, stateless components will require balancing query verbosity (*e.g.*, declaring statefulness), while also balancing the expressiveness of potential exoskeletons under the LLM's limited context. With similar trade-offs, addressing non-deterministic components may require exoskeletons that encode probabilistic guarantees, offering partial but useful constraints on behavior.

In terms of scalability, in its current form, the `exo` prototype must be used as a surgical tool, applied at function level. Straightforward infrastructure could allow pointing `exo` to a larger module and iteratively regenerating it piece-by-piece, but inter-procedural properties and interactions would require more sophisticated exoskeletons and reasoning.

A significant fraction of the techniques behind exoskeletons are language-agnostic, including the logic-based query engine and exoskeletons such as input-output pairs and traces. Other techniques, *e.g.*, the extraction and checking phases of `exo`'s extensions are language-aware. This language awareness complicates interactions with the target components in ways that LLMs fundamentally avoid. Fortunately, adding support for most languages is surmountable as long as analyses exist that can be used in a pluggable fashion—by invoking them as opaque functions applied to the target components. Language translation is a promising approach towards this direction, further capitalizing on the techniques described so far: an analysis that operates on one language is applicable on other languages by performing an inverse translation, extracting information about the translated component's properties in the latter domain, and then applying other stages on the original component. The effectiveness of this approach depends on how well properties translate across languages—for example, memory safety is a property that is meaningful across only some languages. Fortunately, many properties that end up being useful in practice such as commutativity, termination, idempotence, and purity can be encoded across many languages.

Acknowledgments

We are thankful to the anonymous PACMI reviewers for their input. This material is based upon research supported by NSF awards CNS-2247687 and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] 2006. CVE-2006-3400. National Vulnerability Database (NVD), U.S. NIST. https://nvd.nist.gov/vuln/detail/CVE-2006-3400

[2] 2019. CVE-2019-1010043. National Vulnerability Database (NVD), U.S. NIST. https://nvd.nist.gov/vuln/detail/CVE-2019-1010043

[3] 2025. npm. https://www.npmjs.com.

[4] 2025. Python Package Index (PyPI). https://pypi.org.

[5] Iosif Arvanitis, Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. 2022. A systematic analysis of the event-stream incident. In *Proceedings of the 15th European Workshop on Systems Security* (Rennes, France) *(EuroSec '22)*. Association for Computing Machinery, New York, NY, USA, 22–28. https://doi.org/10.1145/3517208.3523753

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[7] Wikipedia contributors. 1999. Fast Inverse Square Root Algorithm. https://en.wikipedia.org/wiki/Fast_inverse_square_root.

[8] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with LLMs: A study of translating to Rust. *arXiv preprint arXiv:2405.11514* (2024).

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2021. Clingo= ASP+ control. *Preliminary report* (2021).

[10] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 260–265.

[11] Arjun Guha, Ben Zorn, Carolyn Jane Anderson, Molly Q Feldman, Sumit Gulwani, and Gabrielle Allen. 2025. The Future of Programming in the Age of Large Language Models. *Future* (2025).

[12] Minyoung Huh, Brian Cheung, Tongzhou Wang, and Phillip Isola. 2024. Position: The Platonic Representation Hypothesis. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 20617–20642. https://proceedings.mlr.press/v235/huh24a.html

[13] Nam Huynh and Beiyu Lin. 2025. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications. arXiv:2503.01245 [cs.SE] https://arxiv.org/abs/2503.01245

[14] Kyomin Hwang, Suyoung Kim, JunHoo Lee, and Nojun Kwak. 2024. Do not think about pink elephant! arXiv:2404.15154 [cs.CL] https://arxiv.org/abs/2404.15154

[15] IOCCC Organizers. 2024. The International Obfuscated C Code Contest. https://www.ioccc.org.

[16] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* (July 2025). https://doi.org/10.1145/3747588

[17] Yibo Jiang, Goutham Rajendran, Pradeep Kumar Ravikumar, and Bryon Aragam. 2024. Do LLMs dream of elephants (when told not to)? Latent concept association and associative memory in transformers. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=WJ04ZX8txM

[18] Charles Jin and Martin Rinard. 2024. Emergent representations of program semantics in language models trained on programs. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 891, 25 pages.

[19] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *The Twelfth International Conference on Learning Representations.*

[20] Holger Krekel and pytest-dev team. 2025. *pluggy: A minimalist production-ready plugin system.* https://github.com/pytest-dev/pluggy

[21] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv:2409.02977 [cs.SE] https://arxiv.org/abs/2409.02977

[22] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer.

[23] OpenAI. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[24] Rosetta Code contributors. 2024. Rosetta Code: Programming Tasks. https://rosettacode.org

[25] Inbal Shani and GitHub Staff. 2023. *Survey reveals AI's impact on the developer experience.* Updated February 7, 2024.

[26] Chongyang Tao, Tao Shen, Shen Gao, Junshuo Zhang, Zhen Li, Zhengwei Tao, and Shuai Ma. 2024. Llms are also effective embedding models: An in-depth overview. *arXiv preprint arXiv:2412.12591* (2024).

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

[28] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[29] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. 2023. DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions. arXiv:2312.04730 [cs.CR] https://arxiv.org/abs/2312.04730

[30] Xilie Xu, Keyi Kong, Ning Liu, Lizhen Cui, Di Wang, Jingfeng Zhang, and Mohan Kankanhalli. 2023. An LLM can Fool Itself: A Prompt-Based Adversarial Attack. arXiv:2310.13345 [cs.CR] https://arxiv.org/abs/2310.13345

[31] Rem Yang, Julian Dai, Nikos Vasilakis, and Martin Rinard. 2025. Evaluating the Generalization Capabilities of Large Language Models on Code Reasoning. arXiv:2504.05518 [cs.SE] https://arxiv.org/abs/2504.05518

[32] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997), 366–427.

[33] Yangtian Zi, Harshitha Menon, and Arjun Guha. 2025. More Than a Score: Probing the Impact of Prompt Specificity on LLM Code Generation. *arXiv preprint arXiv:2508.03678* (2025).

[34] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating code at scale with LLMs at google. *arXiv preprint arXiv:2504.09691* (2025).