Genomics Communications

PERSPECTIVE

Open Access

https://doi.org/10.48130/gcomm-0026-0018

Genomics Communications **2026**, 3: e018

LLM-assisted development of Rust for high-performance bioinformatics software: practices, workflows, and boundaries

Zhou-Geng Xu[1*] and Guihui Qin[2]

[1] *Key Laboratory of Plant Carbon Capture, CAS Center for Excellence in Molecular Plant Sciences (CEMPS), Institute of Plant Physiology and Ecology (SIPPE), Chinese Academy of Sciences (CAS), Shanghai 200032, China*

[2] *Center of Reproduction, Development & Aging, and Institute of Translational Medicine, Faculty of Medicine, University of Macau, Taipa, Macao SAR 999078, China*

* Correspondence: zgxu2016@cemps.ac.cn (Xu ZG)

Abstract

As high-throughput sequencing datasets continue to grow, the dominant bottlenecks in bioinformatics software are increasingly engineering concerns, such as input/output throughput, compression and decompression, memory allocation, and long-term maintainability, rather than algorithmic complexity alone. Rust addresses these concerns through its ownership and borrowing system, zero-cost abstractions, and compile-time concurrency safety without garbage collection. At the same time, large language models (LLMs) and agentic coding assistants are substantially lowering Rust's historically steep adoption barrier. Drawing on the development of hickit_rs, an open-source high-performance Hi-C data processing tool, this Perspective discusses how to effectively combine artificial intelligence (AI) assistance with disciplined software engineering. Our analysis spans two dimensions: Engineering workflow design (a tiered framework of specification-driven design, test-driven development, and multimodel parallel exploration) and analysis of bioinformatics-specific failure modes (layer misidentification, semantic drift in script migration, and goal substitution in agentic loops). We conclude that AI is most effective as an accelerator within a rigorous engineering framework, not a substitute for scientific judgement, and that Rust's compile-time portability guarantee provides a structural benefit that compounds with AI-assisted migration.

Keywords: Rust, Large language model, AI-assisted programming, Bioinformatics software, High-performance computing, Hi-C, Software engineering, Test-driven development

Citation: Xu ZG, Qin G. 2026. LLM-assisted development of Rust for high-performance bioinformatics software: practices, workflows, and boundaries. *Genomics Communications* 3: e018 https://doi.org/10.48130/gcomm-0026-0018

Introduction

Modern bioinformatics pipelines face engineering challenges that are qualitatively different from algorithmic ones. Routine tasks, such as streaming FASTQ, Binary Alignment Map (BAM), Compressed Reference-oriented Alignment Map (CRAM), or Variant Call Format (VCF) files, as well as Hi-C contact pairs at scale, are constrained by large-file input/output (I/O), repeated decompression, heap allocation, and subtle format edge cases. However, most pipelines still delegate these to Bash, AWK, or Python scripts that behave inconsistently across operating systems, high-performance computing (HPC) environments, and container configurations. A more durable approach is to compile performance-critical logic into statically linked native binaries with no runtime interpreter or external dependency chain.

Rust solves these problems at the language level. Its ownership model enforces memory safety at compile time, its lack of garbage collection preserves throughput predictability, and its type-driven concurrency model makes data races structurally impossible[1]. Since the introduction of Rust-Bio in 2016[2], the community has produced infrastructure-grade tools covering BigWig/BigBed I/O[3], out-of-core genomic interval operations[4], whole-genome alignment manipulation[5], and high-performance genomic range queries[6], demonstrating that Rust has moved from feasibility demonstrations into direct upstream dependencies of widely used workflows.

At the same time, large language models (LLMs) and agentic coding assistants are changing how developers approach Rust. The language's historically steep learning curve—lifetimes, borrow checking, trait constraints, error handling—has long hindered its adoption. Artificial intelligence (AI) tools now allow newcomers to obtain natural language explanations of compiler errors, automatically generated boilerplate, and minimal viable prototypes seeded from existing scripts[7,8]. However, empirical studies show that without disciplined review practices, AI coding tools can degrade code quality[9,10], and that human translators outperform automatic tools where memory-safety semantics must be preserved across a translation boundary[11]. "Why now?" therefore has a dual answer: The Rust bioinformatics ecosystem is mature enough to build on, and AI is converting Rust from a high-barrier language into one that can be mastered through iterative, test-driven cycles—provided engineering discipline is maintained.

In this Perspective, we draw on hickit_rs to propose a tiered framework of specification-driven design (SDD), test-driven development (TDD), and multimodel parallel exploration, and analyse three bioinformatics-specific failure modes: Layer misidentification, semantic drift in script migration, and goal substitution in agentic loops.

AI-assisted Rust development in bioinformatics

Rust's growing role in bioinformatics infrastructure

The consistent pattern across these tools[2–6] is that Rust is most compelling where the scripted logic accumulates hidden fragility: High-throughput, I/O-bound, long-maintenance layers requiring performance, memory safety, and cross-platform determinism simultaneously, which is the layer that pipeline orchestrators depend on but cannot themselves fix.

https://doi.org/10.48130/gcomm-0026-0018

LLMs as a Rust adoption accelerator, and their limitations

Systematic reviews have identified two recurring LLM failure patterns: Unstable correctness across semantically equivalent prompts and loss of context in long conversations[7]. Empirical studies trace these to hallucination and incomplete understanding of projects' conventions[8]. A controlled experiment found that Copilot-assisted developers produced lower-quality outputs despite adding more code[10], and a user study showed human translators outperform automatic C-to-Rust tools where memory-safety semantics must be preserved[11].

These findings establish a core tension: AI reduces the friction of adopting Rust, but its failure modes are amplified in bioinformatics by domain-specific format complexity, such as multimember gzip, Blocked GNU Zip Format (BGZF) framing, and AWK boundary conventions, that are absent from generic benchmarks.

The case of Hi-C: when the script logic calls for a Rust rewrite

The Hi-C data processing ecosystem provides the driving application for hickit_rs. Pairtools[12] is the de facto standard for producing and manipulating the 4D Nucleome (4DN) .pairs format. The wider ecosystem includes Juicer[13], which introduced the merged_nodups format and is a widely adopted pipeline for loop-resolution contact analysis, and Hi-C-Pro[14], which is still widely used for allele-specific analyses. Despite this maturity, most Hi-C post-processing logic (map resolution estimation, region filtering, streaming statistics) remains implemented as Bash and AWK scripts. These scripts produce correct results in controlled environments but behave inconsistently across AWK dialects, gzip versions, and locale settings. Containerization mitigates some issues at runtime but does not eliminate the underlying fragility: The script logic itself remains sensitive to the shell interpreter, locale settings, and installed utility versions[15].

hickit_rs is an open-source command-line tool that performs Hi-C map resolution estimation tasks (via a Juicer-compatible algorithm), region-based filtering of merged_nodups contacts, and streaming contact statistics. It rewrites this layer as a typed, tested, statically linked Rust binary supporting both Juicer merged_nodups format and 4DN .pairs.gz format used by Pairtools. On an eight-core 32-GB workstation, it processes approximately 100 million pairs in about 30 s and roughly 1 billion pairs in about 5 min, with peak memory of approximately 300–350 MB, which is an 8–10× speedup compared with the reference Bash/AWK pipeline (Fig. 1). Numerical consistency has been verified across multiple real datasets on both input formats. Critically, cargo build --target … --release produces a statically linked binary with deterministic compilation semantics that behaves identically on a Linux HPC node, a macOS workstation, or a minimal Alpine container, which is a compile-time portability guarantee that containerization alone cannot provide. We verified the identical output on Ubuntu 22.04, macOS 14, and an Alpine 3.19 container, noting that cross-platform behavior can still depend on the target configuration and libc/musl choices; static linking mitigates but does not eliminate these factors.

Achieving this result required more than selecting the right language. The development process itself surfaced recurring points at which AI-assisted coding failed silently, and those failures motivated the engineering framework described in the next section.

A tiered engineering framework for AI-assisted development

The core principle is straightforward: Scale the engineering process to the risk. A one-line bug fix needs only a prompt and a test run; a greenfield compression-layer module demands a full specification, TDD-driven implementation, and multimodel comparison.

The guiding framework has four components (Fig. 2).

Specification-driven design (SDD). Before coding a nontrivial module, a specification should capture the problem statement, I/O contracts enumerating all format variants (multimember gzip, BGZF,

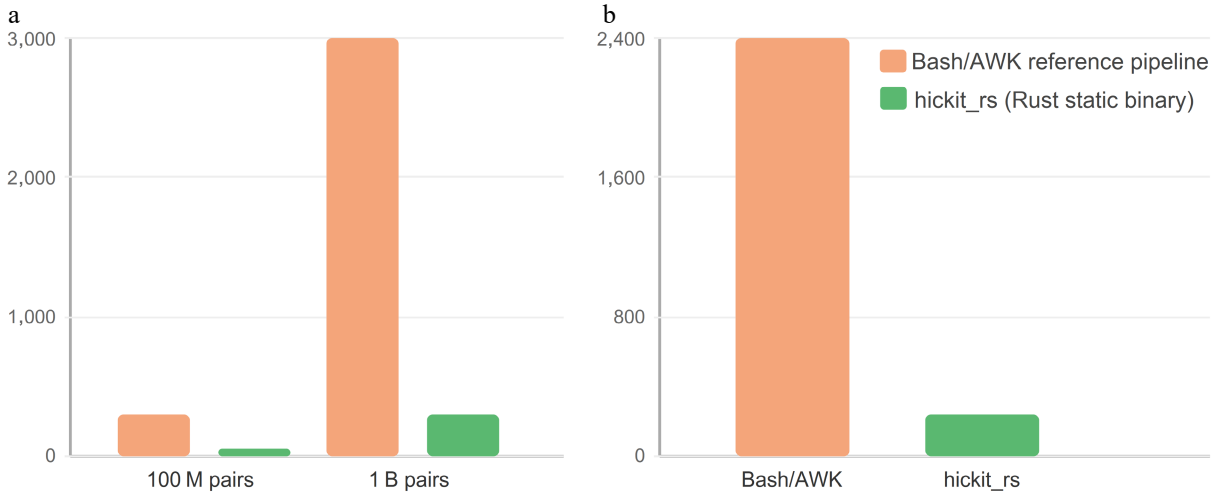

Fig. 1 Performance comparison between the Bash/AWK reference pipeline and hickit_rs on an eight-core 32-GB workstation. (a) Processing time for 100 million and 1 billion read pairs on a logarithmic scale; hickit_rs completes both scales in approximately 30 s and 5 min, respectively, whereas the Bash/AWK pipeline takes roughly 250 s and 3,000 s—an 8.3–10× speedup. (b) Peak memory consumption. The Bash/AWK pipeline peaks at approximately 2,400 MB, whereas hickit_rs uses roughly 300–350 MB, an 86%–88% reduction (7–8× lower). Benchmarks were performed on an eight-core (Intel i7-10700 @ 2.9 GHz) 32-GB RAM workstation running Ubuntu 22.04. Input data were derived from the publicly available human Hi-C dataset SRR1658573 (NCBI SRA), processed to 100 million and 1 billion valid read pairs in Juicer merged_nodups format (gzip-compressed). hickit_rs v0.4 was invoked with the default parameters (hickit resolution merged_nodups.txt.gz); the reference pipeline used Juicer's AWK-based calculate_map_resolution.sh. Wall-clock time and peak memory were measured using /usr/bin/time -v; each benchmark was run three times with the median reported.

dual-format pairs), quantitative performance targets, numerical consistency requirements with the legacy implementation, and known edge cases. This specification serves as both a communication contract with the AI and a reviewable artefact. Modern agentic tools support a brainstorming phase where the AI asks clarifying questions before proceeding.

Test-driven development (TDD). Implementation follows the RED–GREEN–REFACTOR cycle, with each failing test encoding one acceptance criterion. In the AI-assisted context, a failing test provides an unambiguous, machine-verifiable goal that is far more effective than prose instructions. For performance-sensitive code, one should first establish a single-threaded correctness baseline before introducing parallelism[16]. One rule is critical for agentic loops: Every iteration must introduce at least one new test constraint; reusing the same test suite creates conditions for goal substitution as discussed below. For Rust projects, Clippy (the official Rust linter) with the clippy::perf lint group should be integrated into the test cycle to ensure that AI-generated code adheres to performance-oriented idioms, which is particularly important when the AI references the original code being rewritten.

As a project grows across multiple modules, a comprehensive test suite becomes the only artefact giving a developer—or an AI assistant—confidence that a structural change has not silently broken an invariant. The test suite thus serves as a living specification, constraining every future AI-assisted modification to the same correctness envelope.

Multimodel parallel exploration. Because LLM outputs are stochastic, relying on a single model at critical junctures is inadvisable. A robust approach uses the git worktree to host multiple models simultaneously, each generating candidate implementations with the same test data. Unified benchmarks enable lateral comparison, transforming LLMs' stochasticity into a controllable search space. Each branch merges only after human review.

Periodic human review. After a sustained phase of AI-assisted coding, a codebase can accumulate dead code, duplicated logic, inconsistent abstractions, and creeping architectural drift that no individual commit introduced but that compound over time. Scheduling a human-led review after every major feature addition to remove dead paths, consolidate duplicated utility code, and realign modules' boundaries, etc., prevents the codebase from becoming unmaintainable and keeps the AI's context window focused on the current design rather than on archaeology. This checkpoint also serves as the natural moment to audit active tooling: Model Context Protocol (MCP) servers and skill plugins can expose domain-specific capabilities (format validators, crate documentation lookup, persistent memory of design decisions) directly to the agent's context window. A small, well-chosen set of plugins covering only the current bottleneck is more effective than a large general-purpose toolkit; excessive plugin availability expands the agent's option space and raises the risk of overengineered solutions. The practical rule is to start with no plugins, add one when a recurring manual step becomes the limiting factor, and review the active set at each periodic human checkpoint.

In the development of hickit_rs, these four components were applied in sequence and exposed the failure modes detailed below: The SDD phase surfaced gzip-layer complexity that a prompt-only approach had silently missed; TDD fixtures drawn from legacy Bash/AWK output detected boundary-semantic drift on real datasets; and multimodel branches on separate git worktree instances identified goal substitution before it reached the main branch. Automated C-to-Rust translation frameworks such as Syzygy[17] apply a similar test-equivalence philosophy at larger scale. We have since applied the same workflow to two additional migrations, namely sradb_rs (https://github.com/AI4S-YB/sradb_rs), a Rust port of pysradb for Sequence Read Archive (SRA) metadata queries, and fastqc-rs (https://github.com/AI4S-YB/fastqc-rs), a 1:1 rewrite of FastQC v0.12.1. In both, the SDD phase and the TDD oracle strategy were most directly responsible for catching regressions.

The agentic loop: designing for AI's self-iteration

A natural extension of the SDD + TDD framework is to let the AI iterate on its own output autonomously, using the test suite as the feedback signal. This pattern, often called an *agentic loop*, is not a novel concept; its essence is simply a well-designed test harness that the agent can execute, observe, and respond without human intervention at every step. What makes it a recent development is not the idea itself but the maturity of the tooling: Modern agentic integrated development environments (IDEs) and code agents can now run a full test suite, parse failures, propose a targeted patch, apply it, and re-run the code, closing the edit–test cycle in seconds rather than minutes[18]. This is a meaningful contrast with the earlier, simpler pattern of pasting code into a web chat interface: agents operating through a command-line interface (CLI) actively retrieve the relevant context, build and execute the test harness, and consume the compiler and runtime feedback autonomously, handling the majority of compilation and logic errors without manual re-prompting. The context problem, in other words, has largely been solved by the tooling itself.

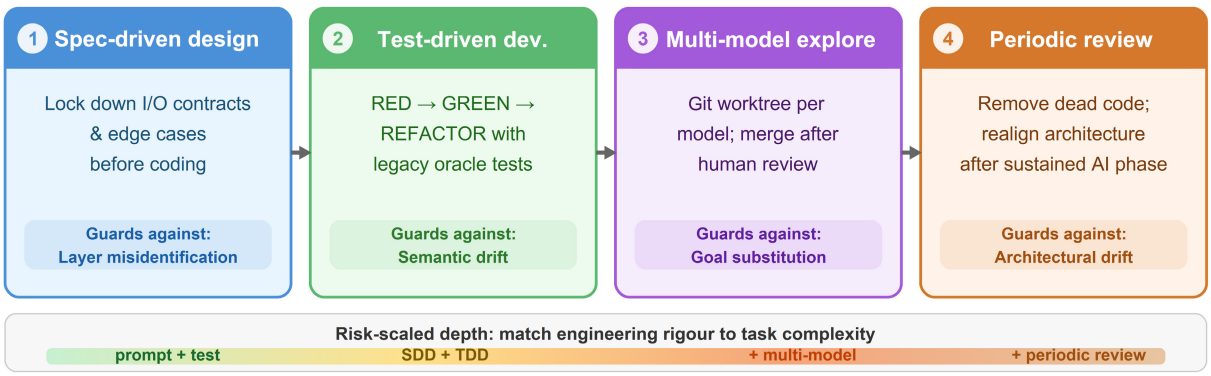

Fig. 2 A tiered engineering framework for AI-assisted development with four components: ① Specification -driven design (SDD), ② test -driven development (TDD), ③ multimodel exploration, and ④ periodic review. These are applied sequentially, with depth scaled to the task's complexity (bottom bar). Each component's primary defence against the three bioinformatics-specific failure modes discussed in the main text is annotated within its panel.

https://doi.org/10.48130/gcomm-0026-0018

The critical design decision is therefore what the agent iterates against. If the feedback signal is shallow—a fixed set of tests on a fixed dataset—the loop converges rapidly to a solution that satisfies those tests and nothing else. Empirical work on feedback-loop-based code generation has shown that a loop's effectiveness decays exponentially once the agent has exhausted the information available in a static test suite[19]. The same countermeasure applies here directly: Each iteration of the agentic loop must introduce at least one fresh constraint, either a new test case or a new input fixture, so that the agent is always working toward a generalizable solution rather than overfitting to the existing evaluation set.

Multiagent architectures extend this further. A closed-loop system in which separate agents handle test generation, execution, failure analysis, and patch review can achieve substantially higher test coverage and lower invalid test rates than single-agent baselines[20]. The Agentic Agile-V framework formalizes this as a specify–constrain–orchestrate–prove–evolve–verify cycle, demonstrating that the central challenge in the development of agentic software has shifted from prompt engineering to process engineering[21]. In bioinformatics, where input formats carry domain-specific edge cases that no generic benchmark covers, the "constrain" and "prove" steps are especially valuable: They are the mechanism by which domain knowledge—format specifications, numerical consistency oracles, real dataset fixtures—enters and governs the agentic loop.

Potential and pitfalls of AI assistance in bioinformatics

The tiered framework of described above reduces but does not eliminate AI's failure risk. Even within its boundaries, three classes of failure remain a persistent risk in bioinformatics. Each is specific to or substantially amplified by the bioinformatics context because the domain formats carry semantic complexity that does not appear in generic software engineering benchmarks; each has a symptom, a root cause, and a concrete countermeasure. These three failure modes were observed across the development of hickit_rs, sradb_rs, and fastqc-rs as described above, and are recurring rather than isolated incidents.

Failure mode 1: layer misidentification

Symptom

The parsed record count is far lower than expected; all sanity checks on the FASTQ/pairs parser pass.

Root cause

The input file is a multimember gzip archive (common in Pairtools output and split-lane FASTQ files), but the program used a single-member gzip decoder that halts after the first member without error. The BGZF format[22]—a deliberate extension of multimember gzip for random access—is treated as a simple gzip stream, silently discarding all but the first block. The symptom presents in the parsing layer; the cause lies one abstraction level below in the compression stack. AI, pattern-matching to surface symptoms, consistently proposed parser-level fixes, none of which addressed the real issue. This failure was first encountered during the development of hickit_rs when processing. pairs.gz output from Pairtools, and was subsequently confirmed in the sradb_rs migration.

Countermeasure

Introduce at least one new input fixture from a different real dataset at every agentic iteration. When using an agentic IDE that exposes the model's reasoning, phrases such as "to satisfy this test" are early warning signals of goal substitution.

Failure mode 2: semantic drift in script migration

Symptom

The Rust reimplementation passes all unit tests but produces subtly different aggregate statistics from the reference Bash/AWK pipeline on edge-case inputs.

Root cause

AWK's NR, FS, and blank-line handling semantics do not map directly to Rust's iterator-based parsing. When AI translates a Bash/AWK script into Rust, it preserves the structure of the logic but silently drops the boundary semantics, leading to empty-line treatment, field-separator edge cases, and off-by-one behavior at stream boundaries. These discrepancies are invisible on clean test data but emerge on real datasets containing header anomalies, trailing whitespace, or non-standard line endings. This pattern was observed repeatedly during migration of the AWK-based map-resolution script (calculate_map_resolution.sh) in hickit_rs, particularly for trailing whitespace and empty-line handling at stream boundaries.

Countermeasure

Use the legacy Bash/AWK output as an oracle to build a cross-format test set: For each real dataset and edge-case variant, capture the legacy output and assert numerical equality in the Rust test suite. This transforms the legacy implementation from "a thing being replaced" into a continuous correctness specification.

Failure mode 3: goal substitution in agentic loops

Symptom

After several agentic iterations, all tests pass. However, when the test dataset is replaced with a different real sample, the solution fails.

Root cause

We observed this pattern directly in the development of hickit_rs using a commercial agentic IDE (Claude Code). Inspection of the model's visible reasoning chain revealed the following: Given a failing test and iteration pressure, the model identified a narrow code path that satisfied the specific test on the specific input file. The fix was logically incomplete: It addressed the test artefact rather than the underlying algorithm. This is *goal substitution*, in which the declared goal ("make the program correct") is silently replaced by the operationalized subgoal ("make this test pass on this input").

Countermeasure

The test-driven development principle described above applies directly: Introduce at least one new input fixture drawn from a different real dataset at every agentic iteration. When using an agentic IDE that exposes the model's reasoning (e.g., extended thinking or scratchpad outputs), scan the chain at each iteration. Phrases such as "to satisfy this test" or "for this specific input" are early warning signals of goal substitution in progress.

Broader implications

These three failure modes share a common structure: AI operates on observable surface features while the true problem lies one level below, reflecting that current LLMs match text patterns rather than reason about layered abstractions. Bioinformatics amplifies this because the domain formats carry semantic complexity that is absent from generic benchmarks. AI's long-term value is thus capability

transfer. By explaining compiler errors and comparing crate designs, AI accelerates the developer's own learning. This is maximized when the developer maintains ownership of the specification, the test oracle, and the final merge decision.

This Perspective does not advocate rewriting all tools in Rust. Python and R remain ideal for exploratory analysis and orchestration; Rust is suited to high-throughput, performance-critical core modules, and AI's contribution is to lower the cost of that targeted migration.

This work complements orchestration agents such as AutoBA[23] and BioAgent Bench[24]: They automate multi-step analyses by calling existing tools, whereas we address the task of developing the performance-critical tools they depend on.

Conclusions

Rust is becoming an important option for high-performance bioinformatics software, and LLMs are changing how Rust is used. The former provides performance, memory safety, and compile-time portability; the latter lower the language barrier and accelerate prototyping and refactoring. Together, they create real opportunities for modernizing the performance-critical layers of the bioinformatics stack.

However, the three failure modes documented here share a common structure: AI operates on surface features while the true problem lies one level below. The tiered SDD + TDD + multimodel framework, extended by a structured agentic loop, addresses this by ensuring domain knowledge enters the feedback signal before the AI iterates. The bottleneck is no longer the language barrier but the engineering discipline, which cannot be delegated to the agent.

Within this framework, AI serves as an accelerator, not the judge. Only when human judgement governs the specification, the test oracle, and the final merge decision can AI-assisted Rust development truly serve the goals of high-performance, reproducible, and maintainable scientific software.

Ethical statements

During the preparation of this manuscript, the authors used SciMaster Vibe Writing (Bohrium; accessed in July 2026) solely for language refinement and for reviewing the logical flow of the manuscript. The underlying language model and version were not publicly disclosed by the service provider. The tool was not used to generate original scientific ideas, data, analyses, interpretations, or conclusions. All AI-assisted output was critically reviewed, verified, and edited by the authors, who take full responsibility for the accuracy, integrity, and originality of the final manuscript. No AI tool is listed as an author.

Author contributions

The authors confirm their contributions to the paper as follows: study conception and design, software development and code, draft manuscript preparation: Xu ZG; testing and validation, manuscript review and editing: Qin G; analysis and interpretation of results: Xu ZG, Qin G. All authors reviewed the results and approved the final version of the manuscript.

Data availability

No new datasets were generated during this study. The benchmark results reported in Fig. 1 were obtained using the publicly available human Hi-C dataset SRR1658573 (NCBI SRA). All software described is openly available in the repositories listed below.

The case study software hickit_rs is publicly available at https://github.com/AI4S-YB/hickit_rs. Additional tools developed with the same SDD + TDD framework are also publicly available: sradb_rs (https://github.com/AI4S-YB/sradb_rs); *fastqc-rs* (https://github.com/AI4S-YB/fastqc-rs).

Acknowledgments

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Conflict of interest

The authors declare that they have no conflict of interest.

Dates

Received 29 June 2026; Revised 28 July 2026; Accepted 30 July 2026; Published online 27 August 2026

References

[1] Bugden W, Alahmar A. 2022. The safety and performance of prominent programming languages. *International Journal of Software Engineering and Knowledge Engineering* 32:713–744

[2] Köster J. 2016. Rust-Bio: a fast and safe bioinformatics library. *Bioinformatics* 32:444–446

[3] Huey JD, Abdennur N. 2024. Bigtools: a high-performance BigWig and BigBed library in Rust. *Bioinformatics* 40:btae350

[4] Wiewiórka M, Khamutou P, Zbysiński M, Gambin T. 2025. polars-bio—fast, scalable, and out-of-core operations on large genomic interval datasets. *Bioinformatics* 41:btaf640

[5] Wei W, Gui S, Yang J, Garrison E, Yan J, et al. 2025. *wgatools*: an ultra-fast toolkit for manipulating whole-genome alignments. *Bioinformatics* 41:btaf132

[6] Buffalo V. 2024. GRanges: a Rust library for genomic range data. *bioRxiv* 1–5

[7] Hou X, Zhao Y, Liu Y, Yang Z, Wang K, et al. 2024. Large language models for software engineering: a systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33:1–79

[8] Zhou X, Liang P, Zhang B, Li Z, Ahmad A, et al. 2025. Exploring the problems, their causes and solutions of AI pair programming: a study on GitHub and Stack Overflow. *Journal of Systems and Software* 219:112204

[9] Zhang B, Liang P, Zhou X, Ahmad A, Waseem M. 2023. Practices and challenges of using GitHub Copilot: an empirical study. *arXiv* Preprint: 2303.08733

[10] Imai S. 2022. Is GitHub copilot a substitute for human pair-programming? An empirical study. *Proceedings of the 44th International Conference on Software Engineering: Companion Proceedings, Pittsburgh, PA, USA, 22–27 May, 2022*. New York, USA: ACM. pp. 319–321 doi: 10.1145/3510454.3522684

[11] Li R, Wang B, Li T, Saxena P, Kundu A. 2025. Translating C to Rust: lessons from a user study. *Proceedings 2025 Network and Distributed System Security Symposium, San Diego, CA, USA, 24–28 February 2025*. USA: Internet Society. pp. 1–18 doi: 10.14722/ndss.2025.241407

[12] Open2C, Abdennur N, Fudenberg G, Flyamer IM, Galitsyna AA, Goloborodko A, et al. 2024. Pairtools: from sequencing data to chromosome contacts. *PLoS Computational Biology* 20:e1012164

[13] Durand NC, Shamim MS, Machol I, Rao SSP, Huntley MH, et al. 2016. Juicer provides a one-click system for analyzing loop-resolution Hi-C experiments. *Cell Systems* 3:95–98

[14] Servant N, Varoquaux N, Lajoie BR, Viara E, Chen CJ, et al. 2015. HiC-Pro: an optimized and flexible pipeline for Hi-C data processing. *Genome Biology* 16:259
[15] Grüning B, Chilton J, Köster J, Dale R, Soranzo N, et al. 2018. Practical computational reproducibility in the life sciences. *Cell Systems* 6:631−635
[16] Nanthaamornphong A, Carver JC. 2017. Test-Driven Development in scientific software: a survey. *Software Quality Journal* 25:343−372
[17] Shetty M, Jain N, Godbole A, Seshia SA, Sen K. 2024. Syzygy: dual code-test C to (safe) Rust translation using LLMs and dynamic analysis. *arXiv* Preprint: 2412.14234
[18] Maddila C, Tait A, Chang C, Cheng D, Ahmad N, et al. 2026. Agentic program repair from test failures at scale: a neuro-symbolic approach with static analysis and test execution feedback. *IEEE Transactions on Software Engineering* 52:2446−2462
[19] Palavalli MA, Santolucito M. 2024. Using a feedback loop for LLM-based infrastructure as code generation. *arXiv* Preprint: 2411.19043
[20] Naqvi S, Baqar M, Ali Mohammad N. 2026. The rise of agentic testing: multi-agent systems for robust software quality assurance. *arXiv* Preprint: 2601.02454
[21] Koch C. 2026. Agentic Agile-V: from vibe coding to verified engineering in software and hardware development. *arXiv* Preprint: 2605.20456
[22] Li H. 2011. Tabix: fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics* 27:718−719
[23] Zhou J, Zhang B, Li G, Chen X, Li H, et al. 2024. An AI agent for fully automated multi-omic analyses. *Advanced Science* 11:2407094
[24] Fa D, Čuljak M, Pandža B, Čupić M. 2026. BioAgent Bench: an AI agent evaluation suite for bioinformatics. *arXiv* Preprint: 2601.21800