ТЕХНИКАЛЫҚ ҒЫЛЫМДАР
ТЕХНИЧЕСКИЕ НАУКИ
TECHNICAL SCIENCE

CENTRAL ASIAN SCIENTIFIC JOURNAL
ISSN 3135-3061
2026. No. 4 (32), Vol. 2. pp. 424–434
DOI: https://doi.org/10.5281/zenodo.24314732
Journal website: https://cajournal.kz

UDC 004.8:004.415.53

HOW MUCH DO ADMISSION TESTS CATCH? A MUTATION STUDY OF GATES FOR LLM-GENERATED TOOLS FROM TWO MODELS

Kaisar Turysbek
International Information Technology University, Almaty, Kazakhstan

Kaisar Turysbek — 2nd-year master's student, Information Technologies programme (M094), International Information Technology University, Almaty, Kazakhstan.

Scientific supervisor: Olzhas M. Olzhayev — PhD, Assistant Professor, International Information Technology University, Almaty, Kazakhstan.

© Kaisar Turysbek, 2026

Abstract: This study measures how many defective tools two admission tests let into the tool library of an agent built on a large language model, using as test material the tools that language models actually wrote. From 90 tools written by one model (study V7) and 134 written by a second model (study V13) we generated 2,650 and 4,096 token-level mutants (one token or one adjacent token pair changed), ran seeded samples of 480 and 448 in a sandbox through a basic gate (shared execution checks plus a visibly valid result on the tool's original task) and a stronger gate (the basic gate plus five authored examples), and judged every mutant on nine authored test cases that no gate sees. In V7, 106 mutants passed the shared checks but failed a test case that their unmodified tool passed; the basic gate admitted 88 of them and the stronger gate 10, at the cost of 1,050 extra sandbox calls and 4 rejections among 123 mutants with no observed defect. In V13, among the mutants of tools that themselves passed both gates, the same counts were 92, 68 and 8. On these injected defects the stronger gate detected far more defective tools than the basic gate, and we consider its extra cost worth paying.

Keywords: large language model agents, generated tools, admission testing, mutation testing, test adequacy, sandboxed execution.

Introduction

Several recent agent designs let a language model write executable functions as tools, and most of them keep those functions and call them again when a similar task arrives [1, 2, 3, 4, 5]. Keeping functions changes what one generation error costs: a wrong answer stays with its task, but a wrong function in a library can be returned to every later task that matches its interface. The moment a function is accepted into the library, which we call admission, is therefore the last cheap place to stop such an error.

The evidence these systems use at admission is usually light. LATM has the tool-making model write unit tests from a few validation samples [1]. CRAFT keeps a tool only if it re-derives the answer to the problem it was created for [2]. TroVE relies on agreement among sampled solutions and on execution success [4], and Voyager asks a language-model critic whether a skill succeeded before storing it [5]. Whether these checks are adequate is rarely measured on its own, and a re-analysis of library-learning systems found that stored functions were rarely reused and that much of the reported gain came from other mechanisms [6]. Work on code generation shows why adequacy matters: stronger test suites for a standard benchmark exposed many programs that had

passed the original tests [7], and patches counted as resolved on SWE-bench were found to fail the developers' full test suite [8]. An admission test for generated tools faces the same question: of the wrong functions that reach it, how many does it stop?

In earlier work the author compared three conditions on authored data-processing tasks: no library, basic reuse and stronger admission [9]. In the longest live study of that series (V6), the stronger admission gate rejected none of the 48 candidates that reached it before the run stopped, and the shorter studies before it showed no rejection either. A gate that never rejects is compatible with two situations: every candidate was correct, or the gate could not see the defects that arrived. Natural runs cannot separate these, because the defects are not known in advance.

Mutation testing was designed for this situation. Small faults are injected into a program on purpose, and a test suite is judged by how many of the resulting mutants it detects [10, 11]. Tests that detect simple faults tend to detect more complex faults built from them [10], and detecting mutants is correlated with detecting real faults, although some real faults are not coupled to the usual mutation operators [12]. Some mutants are equivalent to the original program and can never be detected [11]. Mutation results therefore describe test sensitivity to a chosen family of faults, not how often faults occur in practice.

This paper applies mutation testing to the two admission gates of the earlier study, using the tools that language models actually wrote as the programs to mutate. A first study (V7) mutates the tools of one model; a second study (V13) repeats it unchanged on the tools that a second model from the same provider wrote in a later run on the same tasks. It asks four questions:

– **RQ1.** Among mutants that pass the execution checks shared by both gates and introduce a new detected defect, how many does each gate admit?

– **RQ2.** How are those admissions distributed across mutation operators and across the twelve task interfaces?

– **RQ3.** How much additional sandboxed execution does the stronger gate require, and how often does it reject mutants in which no defect was observed?

– **RQ4.** What are the same measures when the tools were written by a second model?

Both studies are controlled evidence about the gates. They make no model calls and say nothing about how often a model writes defects of these kinds. The section Materials and methods describes the design, which was frozen and reviewed before any mutant was generated. The section Results and discussion reports the results, counts before rates, with the two studies side by side and never pooled, followed by the discussion; the Conclusion closes the paper.

Materials and methods

Design and integrity rules

The study followed a written protocol that was frozen before any mutant was generated or executed. An independent static review of the protocol by a separate AI model (Codex) returned a pass with amendments, and amendment A1 was adopted before generation. It fixed the source deduplication rule, the sampling frame, the token-context rules of each operator, the host boundary, the classification strata and the decision to report no confidence intervals. Three rounds of independent code review by the same reviewer followed; amendment A2 made unit records durable, made the verifier independent of the runner's analysis code and made publication of the summary atomic, again before execution.

The run wrote to a fresh output directory owned by one process, and its results were append-only. Before the first sandbox call, a manifest pinned the SHA-256 hashes of the input event log, the source list, the complete mutant frame with every exclusion, the sampled mutants, the code, the runtime and the seed. No mutant, operator, interface or oracle case could be added or removed after a result had been seen. A read-only verifier then re-derived every gate decision, oracle failure, classification and total from the raw per-call records with its own code and compared them with the runner's summary; all numbers below come from that verification report. The run replays existing tools: it made no model, provider or network requests, and its monetary cost was not measured (recorded as unknown, not as zero).

© Central Asian Scientific Journal, 2026

Source tools

The programs to mutate are tools that a language model wrote during the main part of the earlier V6 study [9]. V6 covered twelve function contracts (interfaces) in six authored data-processing families (Table 1), all pure, stateless functions over rows of JSON-like records. The sources are every tool admitted to a library by either reuse condition while the four complete V6 blocks were building their libraries, deduplicated by interface and exact source text, so the same code with a different description counts once. Each tool's origin task is the earliest task for which it was written. This gave 90 distinct source tools.

Table 1 – Task families and interfaces of the source tools

Family	Interfaces
Record reshaping	*coalesce_fields* and *explode_values*
Interval algebra	*merge_intervals* and *intersect_intervals*
Integer aggregation	*group_statistics* and *rolling_sum*
Text parsing	*split_escaped* and *parse_ranges*
Relational multiplicity	*group_unique_values* and *left_expand_join*
Event sequences	*replay_updates* and *sessionize_events*

On the host the source text is handled as data only: the mutator, sampler, runner and verifier never parse, compile, evaluate or import it, and any judgement that depends on syntax happens inside the sandbox.

Mutation operators

Mutants are produced by the Python standard tokenizer and its inverse, as plain text manipulation. Each mutant changes exactly one token, or one short run of adjacent tokens, of one source. Table 2 lists the eight operator classes; their short names follow the mutation-testing literature [11]. Amendment A1 fixed the context rules: a + or - counts as binary only after a name, number, string or closing bracket; an *in* is a membership test unless a *for* occurs earlier on the same line at the same bracket depth; a *not* is removed only outside *is not* and *not in*; integer constants are plain decimal literals. Duplicate mutants within a source, and mutants identical to their source, are dropped.

Table 2 – Mutation operator classes

Id	Operator class	Change
ROR	Relational boundary	*< ↔ <=, > ↔ >=, == ↔ !=*
CRP	Integer constant	decimal literal *n → n+1* or *n-1* (result kept non-negative)
LCR	Logical connector	*and ↔ or*
NEG	Negation removal	delete a unary *not* (not in *not in* or *is not*)
NONE	None-check flip	*is None ↔ is not None*
MEM	Membership flip	*in ↔ not in* in comparisons (not in *for … in*)
AOR	Arithmetic	binary + ↔ - (unary minus excluded)
BOOL	Boolean constant	*True ↔ False*

A mutant is a changed program, not necessarily a wrong one: some are equivalent to their source, and others differ only on inputs that no case in this study exercises.

Sampling

Every such mutant of every source was enumerated, giving a frame of 2,650 mutants with 0 exclusions. For each interface the protocol draws min(40, N) mutants without replacement from the sorted frame, with a random generator seeded from the study name and the interface. Every interface had at least 40 mutants in the frame, so each contributed exactly 40, for 480 sampled mutants in total. Because the draw is over mutation opportunities, a source with more eligible tokens is more likely to appear, and an operator's share of the sample follows how often its pattern occurs in the code. Sampled mutants were never replaced, including those that later failed the common checks or looked equivalent.

The two admission gates

Both gates are the V6 implementations, imported unchanged. Each starts with the **common checks** that every condition of the earlier study applied: the implementation must load, its signature must match the interface, and a public smoke input must run without error, return output of the declared shape and leave its input unmodified.

– **Basic gate** (V6 basic reuse): the common checks pass, and the **origin check** passes: the tool's call on the arguments of its origin task returns, leaves its input unmodified and produces output that passes the interface's output validator. This uses no expected answer.

– **Stronger gate** (V6 stronger admission): the basic gate passes, and the tool reproduces five authored examples for its interface, two ordinary and three edge cases. All five are run even after one fails.

The stronger gate is finite example testing: it does not generate inputs, unlike property-based testing [13], and it does not repair a rejected tool, unlike execution-feedback repair [14].

Sandboxed execution

All mutants and sources ran in the project's sandboxed runner with a pinned runtime: Bubblewrap namespaces [15], a seccomp system-call filter [16], resource limits and no network, with no fallback to execution on the host. A failed runtime health check would have stopped the study before any unit, and a sandbox failure during the run would have stopped it without retry. The guest shares the host kernel, and some result details (exception and input-mutation flags) are packaged inside the guest, so they are not protected against deliberate forgery by the candidate.

Oracle cases

Each mutant was judged on nine oracle cases for its interface: the three V6 transfer tasks, compared with V6's correctness function, and six private audit cases. None of them is a gate input; the gates see only the smoke input, the origin task's arguments and the five examples. A report pinned before execution compared the arguments of every oracle case with every gate input, and an exact overlap would have stopped the study. The verifier re-derives decisions from the oracle's recorded results; it does not check the oracle itself. Passing all nine cases is recorded as "no observed defect", never as "correct".

Classification and controls

Each completed mutant falls into one class. **Common-check rejected** mutants are rejected by both gates by construction; they appear in the flow but not in the primary measures. A mutant is **defective** when it passes the common checks and fails at least one oracle case that its unmutated parent passes; **inherited only** when it fails only cases its parent also fails; and **no observed defect** when it passes the common checks and all nine oracle cases. As controls, every unmutated source first ran through the same pipeline and was expected to pass the common checks, both gates and all nine oracle cases.

Measures and reporting rules

The primary measures, conditional on the defective mutants, are the number admitted by the basic gate, the number admitted by the stronger gate, and the number admitted by the basic gate but rejected by the stronger one ("caught by the stronger gate only"). Each rejection of a defective mutant is attributed to the origin check (part of both gates) or to the stronger examples.

Three secondary measures follow the protocol. **Transfer failures under forced reuse** count, for each admitted mutant, how many of its interface's three transfer tasks it fails (zero to three); they measure exposure of admitted defects, not end-to-end accuracy. **Rejections among mutants with no observed defect** are reported with the category of each failed check; they are not a false-positive rate, because such a mutant may still break the contract on an admission input that the oracle does not contain. **Gate work** counts sandbox calls per completed mutant, including the origin call.

Results are counts by operator and by interface. The original protocol planned Wilson intervals; amendment A1 replaced them with counts and descriptive rates, because mutants cluster within sources and interfaces. No significance test is made, and a rate is printed only when its denominator is at least 10.

Second source set (V13)

V13 repeated this design on tools written by a second model. Its protocol was frozen before any mutant was executed and was reviewed by the same reviewer, who found no high-severity issue; the two medium-severity findings and one low-severity finding were fixed before execution. A digest of every interface declaration, smoke input, admission example, transfer task and audit case was identical in both studies' workspaces, and the gate code is byte-identical, so the gates and the oracle are V7's. Four things differ, all fixed before the run:

– **Sources.** The tools that *gpt-6-luna* wrote and a reuse condition stored while all six blocks of a later, complete live run on the same tasks (V9) built their libraries [9], deduplicated as before. The V7 sources were written by *gpt-6.1-sol*, from the same provider.

– **Seed.** The same draw rule, with the new study name as the seed label.

– **Controls by outcome.** In V9 the stronger gate had rejected tools for accepting invalid input, so some unmutated sources were expected to fail it; every control is reported with its outcome, and none is dropped.

– **Strata by parent status.** Each mutant takes the status of its unmutated parent. The **primary stratum** holds mutants whose parent passes both gates and all nine oracle cases, as every V7 parent did; it is the one comparable with V7, because when the stronger gate already rejects the parent, its decision on a mutant says nothing about the mutation.

The measures are computed on the primary stratum, which gives the primary result, and over all sampled mutants. V13 is never pooled with V7: the studies are compared only by placing their counts side by side, with no pooled number and no test.

Results and discussion

Controls and completion

All 90 unmutated source tools ran as controls and 90 completed; 0 failed any expectation (the common checks, either gate or any of the nine oracle cases). Every sampled mutant therefore has a parent that passed everything (480 mutants; 0 with a failing parent, 0 with no usable parent). The run completed 480 of 480 sampled mutants, so the results describe the complete pre-registered sample.

From sample to classes

Table 3 traces the sample. Of the 480 mutants, 251 failed the common checks that both gates share, all of them at the smoke call. Of the 229 that passed, 106 were defective and 123 showed no observed defect. No mutant was inherited only or unclassifiable.

Table 3 – Flow of the sampled V7 mutants

Stage	Mutants
Enumerated frame	2,650
Sampled (40 per interface)	480
Completed	480
Rejected by the common checks	251
Passed the common checks	229
— defective	106
— no observed defect	123
— inherited only	0
— unclassifiable	0

Admissions of defective mutants (RQ1)

Of the 106 defective mutants, the basic gate admitted 88 (83.0%) and the stronger gate 10 (9.4%); the stronger gate caught 78 (73.6%) that the basic gate admitted. By stage, the origin check rejected 18, the five stronger examples rejected 78 more, and 10 passed both gates (18 + 78 + 10 = 106). The common checks rejected none of these mutants by definition; their contribution is the 251 rejections in Table 3, which apply equally to both gates.

© Central Asian Scientific Journal, 2026

By mutation operator (RQ2)

Table 4 breaks the primary measures down by operator class. The operators were sampled very unequally, following how often each pattern occurs in the code, and the share that turned out defective differed widely: NEG 1 of 164, LCR 3 of 103, CRP 39 of 80, ROR 19 of 55, NONE 35 of 39, MEM 3 of 33, AOR 3 of 3 and BOOL 3 of 3. Only ROR, CRP and NONE reached ten defective mutants, so only those rows carry rates. All 10 mutants admitted by the stronger gate came from CRP (7) and ROR (3).

Table 4 – V7 defective mutants and gate admissions by operator class

Operator	Sampled	Defective	Admitted by basic	Admitted by stronger	Caught by stronger only
ROR (Relational boundary)	55	19	15 (78.9%)	3 (15.8%)	12 (63.2%)
CRP (Integer constant)	80	39	35 (89.7%)	7 (17.9%)	28 (71.8%)
LCR (Logical connector)	103	3	2	0	2
NEG (Negation removal)	164	1	1	0	1
NONE (None-check flip)	39	35	28 (80.0%)	0 (0.0%)	28 (80.0%)
MEM (Membership flip)	33	3	3	0	3
AOR (Arithmetic)	3	3	2	0	2
BOOL (Boolean constant)	3	3	2	0	2
Total	**480**	**106**	**88 (83.0%)**	**10 (9.4%)**	**78 (73.6%)**

By interface (RQ2)

Every interface contributed 40 sampled mutants, but the number that turned out defective ranged from 2 to 17 (Table 5); five of the twelve interfaces reached the threshold for rates. The basic gate admitted at least one defective mutant in every interface. The stronger gate admitted defective mutants in two interfaces, *parse_ranges* (8) and *rolling_sum* (2), and none in the other ten. Figure 1 shows the same counts by outcome.

Table 5 – V7 defective mutants and gate admissions by interface

Interface	Sampled	Defective	Admitted by basic	Admitted by stronger	Caught by stronger only
coalesce_fields	40	3	3	0	3
explode_values	40	5	3	0	3
merge_intervals	40	8	5	0	5
intersect_intervals	40	8	7	0	7
group_statistics	40	9	8	0	8
rolling_sum	40	10	8 (80.0%)	2 (20.0%)	6 (60.0%)
split_escaped	40	6	4	0	4
parse_ranges	40	17	15 (88.2%)	8 (47.1%)	7 (41.2%)
group_unique_values	40	12	11 (91.7%)	0 (0.0%)	11 (91.7%)
left_expand_join	40	12	10 (83.3%)	0 (0.0%)	10 (83.3%)
replay_updates	40	2	2	0	2
sessionize_events	40	14	12 (85.7%)	0 (0.0%)	12 (85.7%)
Total	**480**	**106**	**88 (83.0%)**	**10 (9.4%)**	**78 (73.6%)**

© Central Asian Scientific Journal, 2026

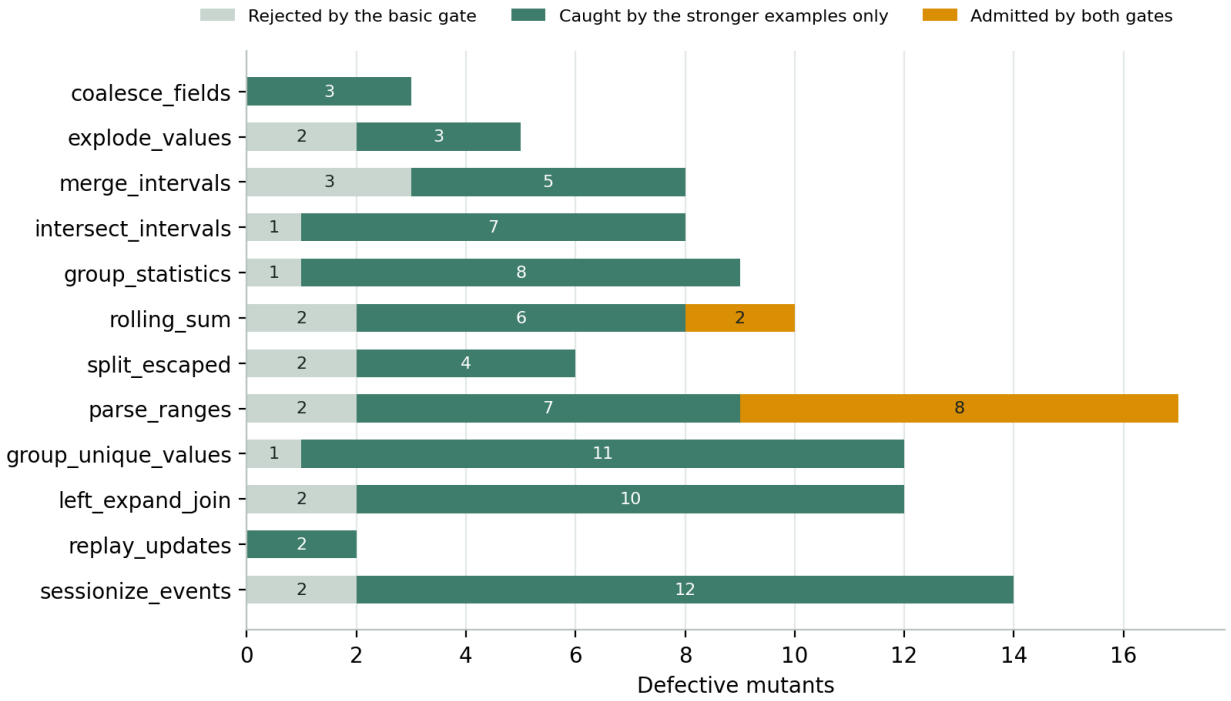


Figure 1 - V7 defective mutants per interface by the gate that stopped them

Transfer failures under forced reuse

If every admitted defective mutant had been reused on its interface's three transfer tasks, those admitted by the basic gate would have failed 227 transfer cases in total (2.14 per defective mutant over all 106), and those admitted by the stronger gate 10 (0.09). Per admitted mutant (our division of the same totals), this is 2.58 failed transfer cases for the 88 mutants the basic gate admitted and 1.00 for the 10 the stronger gate admitted, out of a possible three. These totals are not accuracy figures for an agent, since a rejected tool would have been replaced by a fresh generation whose correctness this study does not observe.

Rejections of mutants with no observed defect (RQ3)

Of the 123 mutants with no observed defect, the basic gate rejected 1 and the stronger gate 4: at the origin check, 1 that raised an error; at the examples, 3 that returned a result for an input that the contract declares invalid. A mutant of the second kind passes all nine oracle cases but does not meet the contract on an admission input, so these rejections are not counted as false positives.

Gate work (RQ3)

Across the 480 completed mutants (controls excluded), the basic gate used a mean of 2.48 sandbox calls per mutant and the stronger gate 4.66; the five examples added 1,050 calls, 2.19 per mutant. Over the 480 completed mutants this is about 1,189 sandbox calls for the basic gate and 2,239 for the stronger gate. The 1,050 example calls are exactly five for each of the 210 mutants that passed the basic gate (229 that passed the common checks, minus 18 defective and 1 other mutants rejected at the origin check), consistent with the examples running only after the basic gate passes. Reporting this work follows the recommendation that agent evaluations state their costs next to their accuracy [17].

Second source set (V13, RQ4)

All 134 unmutated V13 sources ran as controls and 134 completed; 130 passed both gates and all nine oracle cases. The other 4, all *coalesce_fields* tools, were rejected by the stronger gate while passing all nine oracle cases; at the examples they returned a result for an input that the contract declares invalid. The frame held 4,096 mutants (0 exclusions); the *coalesce_fields* sources yielded only 8, so the sample held 448: 40 for each of the other eleven interfaces and 8 for *coalesce_fields*. All 448 completed. The common checks rejected 204, all at the smoke call; of the 244 that passed, 100 were defective and 144 showed no observed defect. By parent status, 440 mutants are in the primary stratum and 8 have a parent rejected by the stronger gate.

In the primary stratum, 236 mutants passed the common checks and 92 were defective. The basic gate admitted 68 (73.9%) and the stronger gate 8 (8.7%), so the stronger gate caught 60 (65.2%) that the basic gate admitted. The origin check rejected 24, the examples 60, and 8 passed both gates. Over all 448 sampled mutants, 100 were defective, of which the basic gate admitted 74 (74.0%) and the stronger gate 8 (8.0%). In the primary stratum the stronger gate's escapes came from CRP (6) and ROR (2) (Table 6) and from the interfaces *parse_ranges* (4), *rolling_sum* (2) and *sessionize_events* (2).

Under forced reuse, the admitted defective mutants of the primary stratum would fail 174 transfer cases with the basic gate and 11 with the stronger gate. Of the 144 mutants with no observed defect, the basic gate rejected 0 and the stronger gate 6: at the origin check, none; at the examples, 6 that returned a result for an input that the contract declares invalid. The examples added 1,060 sandbox calls in the primary stratum, 2.41 per completed mutant. Table 7 places the V7 and V13 counts side by side; it has no pooled number and no test, and "–" marks a measure that has no stratum or is not reported.

Table 6 – V13 primary stratum by operator class

Operator	Sampled	Defective	Admitted by basic	Admitted by stronger	Caught by stronger only
ROR	49	16	12 (75.0%)	2 (12.5%)	10 (62.5%)
CRP	83	44	33 (75.0%)	6 (13.6%)	27 (61.4%)
LCR	126	1	0	0	0
NEG	135	1	1	0	1
NONE	23	21	14 (66.7%)	0 (0.0%)	14 (66.7%)
MEM	18	3	2	0	2
AOR	5	5	5	0	5
BOOL	1	1	1	0	1
Total (primary stratum)	**440**	**92**	**68 (73.9%)**	**8 (8.7%)**	**60 (65.2%)**

Table 7 – V7 and V13 side by side

Measure	V7 (*gpt-6.1-sol* tools)	V13 primary stratum	V13 all sampled
Unmutated source tools	90	–	134
Sources failing a gate or an oracle case	0	–	4
Enumerated frame	2,650	–	4,096
Sampled mutants	480	440	448
Passed the common checks	229	236	244
Defective	106	92	100
Admitted by basic	88 (83.0%)	68 (73.9%)	74 (74.0%)
Admitted by stronger	10 (9.4%)	8 (8.7%)	8 (8.0%)
Caught by stronger only	78 (73.6%)	60 (65.2%)	66 (66.0%)
Transfer failures under forced reuse, basic	227	174	192
Transfer failures under forced reuse, stronger	10	11	11
No observed defect, rejected by stronger	4 of 123	6 of 144	–
Example calls per completed mutant	2.19	2.41	–

Limits that apply to every number above

– **Mutants of one source are not independent.** The 480 V7 mutants come from 90 sources and the 448 V13 mutants from 134, and a source with more eligible tokens contributes more mutants, so counts are not independent trials and no confidence intervals are given. Most operator and interface cells hold fewer than ten defective mutants and carry counts only.

– **The oracle is authored.** Its nine cases per interface are disjoint from the gate inputs but share an author with the gate examples and exercise related rules, so they are not independent evidence of correctness. A mutant that passes them may still be wrong.

– **Injected defects are not natural defects.** Single-token changes are a narrow fault family; the results describe the gates' sensitivity to them and say nothing about how often a model writes such defects, or about defects that need larger edits.

– **One task bank, two models from one provider.** Both source sets are tools for the same twelve authored, pure data-processing interfaces, written by two models of one provider; other tasks, languages, model families or code styles may behave differently.

– **A thin interface in V13.** *coalesce_fields* contributes 8 mutants, none of them in the primary stratum.

– **No model calls.** Both studies replay existing tools; they do not measure what an agent would have done after a rejection.

Discussion

Points the results support (facts only):

– V7: of 106 defective mutants, the basic gate admitted 88 and the stronger gate 10; the origin check rejected 18 and the examples 78.

– V13 primary stratum: of 92 defective mutants, the basic gate admitted 68 and the stronger gate 8.

– Stronger-gate escapes by operator: V7 CRP (7) and ROR (3); V13 CRP (6) and ROR (2).

– Rejections among mutants with no observed defect, stronger gate: V7 4 of 123, V13 6 of 144; extra example calls: V7 1,050, V13 1,060.

The stronger gate was much better at detecting defective tools: in V7 it admitted 10 of the 106 defective mutants where the basic gate admitted 88. The second study points the same way: among the mutants of tools written by a second model that themselves passed both gates (the V13 primary stratum), it admitted 8 of 92 where the basic gate admitted 68.

The 10 defective mutants that passed the stronger gate in V7 were all boundary or constant changes, and 8 of them were in one interface. To catch such defects we would add test inputs that sit exactly on the boundaries, random or property-based tests alongside the hand-written examples, and extra tests for an interface where the shared five examples are not enough. None of these additions was tested in this study.

The stronger gate has a price: 1,050 extra sandbox calls in V7 and 4 rejections among 123 mutants in which no defect was observed. We consider this price worth paying, because a defect is cheaper to find at admission than after the tool has been reused. This is our judgement; the study did not measure the later cost of a defect.

Three things should not be concluded from this study. The defects were injected, so the results do not show how often a model writes such defects itself. Both models come from one provider, so other model families may behave differently. The tasks were authored for the study and are simple data-processing functions, not real projects.

Conclusion

The stronger gate detected far more defective tools than the basic gate in both studies (10 against 88 of 106 in V7; 8 against 68 of 92 in the V13 primary stratum), and we consider its extra cost worth paying because defects are cheaper to catch early; that cost comparison was not measured here. The defects it missed were boundary and constant changes, which points to boundary tests, random or property-based tests and interface-specific tests as the next additions. These results hold for injected defects, two models from one provider and simple authored tasks.

AI-assistance disclosure

An AI coding assistant was used to implement the study software, to review code and to draft the factual parts of this article (Materials and methods, and the results). Every number in the abstract, results, tables, figure, fact list, discussion and conclusion is generated by a script from the read-only verified V7 and V13 results files; none was typed by hand. The protocol and code reviews were performed by a separate AI model. The research questions, the approval of the protocols and their amendments, and the interpretation of the results are the author's own: the discussion, the conclusion and the abstract's last sentence come from the author's own answers in an interview with the AI assistant, which edited the grammar and added verified figures and scope qualifiers that the author approved.

© Central Asian Scientific Journal, 2026

Data availability

The protocols with their amendments and reviews, the verified results and the scripts that generate this article are available from the author on request. The V7 numbers come from the verification report with SHA-256 *4b34df4d6545685e9154994b8ddc04984698a5488596a531d8039be69a4420a9* and the V13 numbers from the report with SHA-256 *85fc075c98eeab7b24f655e445b60dbb0eb28db3dd99806f49541028cbadf351*.

References:

1. Cai T., Wang X., Ma T. et al. Large Language Models as Tool Makers [Electronic resource] // arXiv preprint arXiv:2305.17126v2. – 2023 (rev. 2024). – URL: https://arxiv.org/abs/2305.17126v2 (accessed: 17.11.2026).

2. Yuan L., Chen Y., Wang X. et al. CRAFT: Customizing LLMs by Creating and Retrieving from Specialized Toolsets [Electronic resource] // arXiv preprint arXiv:2309.17428v2. – 2023 (rev. 2024). – URL: https://arxiv.org/abs/2309.17428v2 (accessed: 17.11.2026).

3. Qian C., Han C., Fung Y. R. et al. CREATOR: Tool Creation for Disentangling Abstract and Concrete Reasoning of Large Language Models [Electronic resource] // arXiv preprint arXiv:2305.14318v3. – 2023 (rev. 2024). – URL: https://arxiv.org/abs/2305.14318v3 (accessed: 17.11.2026).

4. Wang Z., Neubig G., Fried D. TroVE: Inducing Verifiable and Efficient Toolboxes for Solving Programmatic Tasks // Proceedings of Machine Learning Research. – 2024. – Vol. 235. – P. 51177–51191. – URL: https://proceedings.mlr.press/v235/wang24az.html (accessed: 17.11.2026).

5. Wang G., Xie Y., Jiang Y. et al. Voyager: An Open-Ended Embodied Agent with Large Language Models [Electronic resource] // arXiv preprint arXiv:2305.16291v2. – 2023. – URL: https://arxiv.org/abs/2305.16291v2 (accessed: 17.11.2026).

6. Berlot-Attwell I., Rudzicz F., Si X. Library Learning Doesn't: The Curious Case of the Single-Use "Library" [Electronic resource] // 4th MATH-AI Workshop at NeurIPS 2024; arXiv preprint arXiv:2410.20274. – 2024. – URL: https://arxiv.org/abs/2410.20274 (accessed: 17.11.2026).

7. Liu J., Xia C. S., Wang Y., Zhang L. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation [Electronic resource] // arXiv preprint arXiv:2305.01210v3. – 2023. – URL: https://arxiv.org/abs/2305.01210v3 (accessed: 17.11.2026).

8. Wang Y., Pradel M., Liu Z. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study [Electronic resource] // arXiv preprint arXiv:2503.15223v2. – 2025. – URL: https://arxiv.org/abs/2503.15223v2 (accessed: 17.11.2026).

9. Turysbek K. Validating before reuse: a controlled study of admission gates for LLM-generated tool libraries // Студенческий вестник: электрон. научн. журн. – 2026. – № 42(422). – URL: https://www.internauka.org/journal/stud/herald/422 (accessed: 17.11.2026).

10. DeMillo R. A., Lipton R. J., Sayward F. G. Hints on Test Data Selection: Help for the Practicing Programmer // Computer. – 1978. – Vol. 11, No. 4. – P. 34–41. – DOI: 10.1109/C-M.1978.218136.

11. Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing // IEEE Transactions on Software Engineering. – 2011. – Vol. 37, No. 5. – P. 649–678. – DOI: 10.1109/TSE.2010.62.

12. Just R., Jalali D., Inozemtseva L. et al. Are Mutants a Valid Substitute for Real Faults in Software Testing? // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). – 2014. – P. 654–665. – DOI: 10.1145/2635868.2635929.

13. Claessen K., Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs // Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00). – 2000. – P. 268–279. – DOI: 10.1145/351240.351266.

14. Chen X., Lin M., Schärli N., Zhou D. Teaching Large Language Models to Self-Debug [Electronic resource] // arXiv preprint arXiv:2304.05128v2. – 2023. – URL: https://arxiv.org/abs/2304.05128v2 (accessed: 17.11.2026).

15. Bubblewrap: Low-Level Unprivileged Sandboxing Tool [Electronic resource] // GitHub, containers project. – URL: https://github.com/containers/bubblewrap (accessed: 08.11.2026).

16. Seccomp BPF (SECure COMPuting with Filters) [Electronic resource] // The Linux Kernel Documentation. – URL: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html (accessed: 08.11.2026).

17. Kapoor S., Stroebl B., Siegel Z. S. et al. AI Agents That Matter [Electronic resource] // arXiv preprint arXiv:2407.01502v1. – 2024. – URL: https://arxiv.org/abs/2407.01502v1 (accessed: 17.11.2026).