V1.1 · AUGUST 2026

The shell, not the system

The same model on Linux and Windows. Which of the two the work depends on, and the six commands that succeed while doing something else.

Nikolaos Broikos · broikos.gr August 2026 76 commands · 4 shells · 28 Sep 2026

71 points

BETWEEN TWO SHELLS ON ONE MACHINE

p = 0.497

FOR THE OPERATING SYSTEM DIFFERENCE

0

COMMANDS THAT FAILED LOUDLY ON WINDOWS

17 sessions

AND THE AGENT ROUTED AROUND EVERY TRAP TESTED

01 What this is

> Discharges claims G001 to G009, G046 to G048, G053, G058, G061, G071.

A coding agent runs the same model whether it is working on Linux, macOS or Windows. The weights do not change. Whatever else varies between those machines, intelligence is not one of the variables, and that makes a question answerable that usually is not: **when the same model does worse work on one platform, what exactly is doing the damage?**

This paper measures the part of that question you can answer without a model at all.

The claim in one paragraph

Running the same 76 commands, of the shape a model emits by default, through four environments: Linux bash executes 76 of 76 correctly, Git Bash on Windows 74 of 76, PowerShell 7.6.6 on that same Windows machine 34 of 76, and the Windows PowerShell 5.1 that the agent actually gets 20 of 76.

The shell gap is large enough to be beyond argument. The operating system gap is not detectable in this measurement at all. 76 of 76 against 74 of 76 gives a two sided Fisher exact p of 0.497, which is no evidence of a difference. The honest statement is not that the operating system contributes 2.6 points. It is that **the command survival metric cannot see an operating system effect at this sample size**, while it sees the shell effect immediately.

That is not a null result, because a second instrument looked somewhere else and found one. With the shell held constant, six commands out of twenty four did something different on Windows and every one of them reported success. **The operating system effect is real and it lives entirely outside the metric the study was built to measure it with.**

That paragraph is the paper.

What was measured, and what was not

Two instruments, neither of which invokes a language model:

The command oracle. 76 shell commands with a known POSIX result, run through each environment's own shell and compared against what the command means. This measures how much of a model's default vocabulary survives contact with the environment. It is a floor, not an outcome: it establishes what *can* execute, not what a model would choose to write.

The silent probe. 24 commands run with the shell held constant on both platforms, so that anything differing belongs to the operating system and its filesystem rather than to a dialect. Nothing was hardcoded as an expected answer. Each platform recorded what actually happened, and Linux was treated as ground truth afterwards.

Time was not measured, and its absence is deliberate. The two machines have different processors and different disks. A calibration run established that their single core speeds agree within 3%, which is enough to say the machines are comparable and not enough to turn a millisecond into a statement about an operating system. Slow is not wrong. This paper is about wrong.

The terminal emulator was excluded by construction rather than controlled for. Tool output reaches a model through a pipe. No terminal is in the loop at any point, which the instruments demonstrate rather than assume: they capture through pipes, touch no terminal, and still reproduce every difference reported here. A terminal cannot change an exit code or a byte.

The environments

ARM	SYSTEM AND FILESYSTEM	SHELL
L-bash	Debian 13, kernel 6.12.95, ext4	bash
W-gitbash	Windows 11 Pro 10.0.26200, NTFS	Git Bash, POSIX sh
W-ps7	Windows 11 Pro 10.0.26200, NTFS	PowerShell 7.6.6
W-ps51	Windows 11 Pro 10.0.26200, NTFS	Windows PowerShell 5.1.26100.9444

Three of the four arms share one machine, which is what makes the shell comparison free of any hardware objection: same disk, same processor, same hour.

The toolchain, because "Git Bash" is not a version and a paper that omits this cannot be reproduced:

	L-BASH	W-GITBASH	W-PS7 / W-PS51
kernel or build	Linux 6.12.95	MINGW64_NT 10.0 26200, msys 3.6.6	Windows 11 10.0.26200
bash	5.2.37	**5.2.37**	n/a
sed	4.9	**4.9**	n/a
grep	3.11	**3.0**	n/a
git	2.x	2.53.0.windows.2	same
locale	en_US.UTF-8	LANG unset, LC_CTYPE C.UTF-8	code page 437

Two of those rows are confounds rather than context. grep is not the same version on the two POSIX arms, which costs one of the six findings in section 05 its attribution. The locales differ on all three, and none of them is the same as another. Both are recorded in section 07 and neither was controlled, because neither was noticed until after the runs.

macOS is absent. There is no Mac here, and a virtualised one would confound the single arm it was meant to measure. Nothing in this paper describes macOS, and the title says two platforms rather than implying three.

What this paper does not claim

It does not claim that an agent produces worse work on Windows. It claims the environment executes less of the default command vocabulary and diverges silently in six specific places. Those are properties of the ground the agent walks on, measured without a model.

Whether an agent actually falls in is asked separately, in section 08, and the answer on the three traps tested is no. Seventeen sessions across two models and both platforms routed around every one of them. The environment is hostile and the agent compensates, and a paper reporting only the first half would be the more

dramatic one and the less true.

Where that distinction is load bearing, section 09 says so explicitly rather than leaving the reader to infer it.

A note on how this went

The paper was planned as a study of operating systems. The first instrument was built before any model session because it consumes nothing, and it contradicted the plan on the same day the plan was written. The framing changed to follow it.

Two measurement errors were caught and are reported in place rather than tidied away: a calibration figure that was physically impossible and four probes whose expectations were wrong. Both are in section 09, because a paper that reports only the instruments that worked is describing a process that did not happen.

02 The instrument, and why it cost nothing

> Discharges claims G009 to G016.

Both instruments are scripts. Neither calls a model, neither needs a key, and both can be run to exhaustion by anybody who wants to dispute a number in this paper. That is not a virtue claim, it is the reason the study exists at the size it does: the findings that follow came out of an afternoon of shell commands rather than out of a budget.

The corpus

76 commands across twelve categories: redirection, exit codes and chaining, text processing, command substitution and variables, quoting, paths, case behaviour, globbing, find and xargs, line endings, tool availability, and write-then-read coherence.

They were chosen to be **the vocabulary a language model reaches for by default**, because that vocabulary is overwhelmingly POSIX. A model's training data is full of grep -n, sed -i, find . -name, 2>/dev/null, cmd && cmd and $(...). When it is dropped into a shell that does not speak that language, the mismatch is not exotic. It is the first thing that happens.

Each probe carries its expected stdout and expected exit code, defined as what the command means on POSIX, because that is the contract the model is writing against.

The classification, and why the middle codes matter

```
ok              exit code and stdout both as expected
crlf            correct once line endings are normalised
output-differs  exit code right, output wrong
exit-differs    exit code wrong, including a failure reported as success
not-found       the tool is not there
timeout         did not return
```

crlf exists as its own code because it is not a crash and it is not a success. The command ran, the exit code was right, and the text came back different. That is precisely how an exact string match silently fails to find a line it is looking at.

exit-differs covers the worst case in the set: a command that failed while reporting success. A missing tool announces itself. A wrong exit code tells the agent something untrue and the agent proceeds on it.

Running the reference arm first, and what it caught

Linux bash ran first, and scored **94.7%**, on four failures.

All four were defects in the probes. An outer shell expanded a variable before the inner shell saw it. An expected exit code was taken from the wrong end of a pipe. A semicolon in find -exec was unescaped. A cut range was off by one.

None was an environment failure, and that is the point of running the reference first: **any probe that fails on plain bash is a broken probe, because plain bash is the contract the corpus is written against.** Corrected, the reference arm scores 100.0%, and every number measured against it afterwards means something. Had the corpus gone to Windows uncorrected, four probe bugs would have been distributed across four arms and read as environment differences.

The uncorrected run is published alongside the corrected one rather than deleted.

The second instrument, and the decision not to guess

The silent probe holds the shell constant, POSIX on both platforms, so any difference belongs to the operating system rather than to a dialect. 24 commands covering case behaviour, symbolic and hard links, permissions, reserved names, trailing characters, open file handles, line endings, deep paths, unicode names and binary through a pipe.

Nothing in it is hardcoded as an expected result. For several of these the honest position was that the answer was unknown, and writing down a guess as an expectation would have been a guess dressed as a control. Each arm records what happened; Linux is taken as ground truth afterwards; the comparison is made in analysis where it can be re-done differently by anybody who disagrees with the choice of ground truth.

The classification that comes out of it has three values, and the middle one is the finding:

```
match     same result on both platforms
silent    exit 0 on both, different result
loud      non-zero exit on Windows, so the agent can see it
```

What is stored

Every probe keeps its full command, stdout, stderr and exit code. A classification is a claim; the captured bytes are the data. This programme lost a 900 site crawl once by recording a derived verdict and discarding the evidence it came from, and the rule that came out of it is applied here from the first run rather than after the seventh.

What the instruments cannot do

They execute commands. They do not choose them.

That boundary is the single most important limitation in this paper and it is stated here rather than only in section 09. The oracle can prove that sed -i behaves differently on two platforms. It cannot tell you whether a model would use sed -i, whether it would notice the difference, or whether it would recover. **It measures the floor, and the distance between the floor and the outcome is the entire remaining study.**

03 The shell is the whole story

> Discharges claims G002 to G008, G017 to G023, G061 to G063.

ARM	OK OF 76	SURVIVAL	WILSON 95% CI	WRONG EXIT	WRONG OUTPUT	CRLF ONLY	NOT FOUND
L-bash, Debian	76	100.0%	95.2 to 100.0	0	0	0	0
W-gitbash, Windows	74	97.4%	90.9 to 99.3	0	2	0	0
W-ps7, Windows	34	44.7%	34.1 to 55.9	17	18	3	4
W-ps51, Windows	20	26.3%	17.7 to 37.2	34	14	4	4

Three of those four rows are the same computer. Same disk, same processor, same hour, same files, same bash 5.2.37 where bash is involved. The only thing that changed was which program was asked to interpret the command.

The shell intervals do not overlap and are not close to overlapping. 90.9 to 99.3 against 17.7 to 37.2 is not a result that needs a significance test to be believed.

The two operating system intervals overlap almost completely, 95.2 to 100.0 against 90.9 to 99.3, and a two sided Fisher exact test on 76 of 76 against 74 of 76 gives p = 0.497. There is no evidence here of an operating system effect on command survival. Section 05 finds one elsewhere, which is the paper's argument rather than a rescue: **the metric that answers the shell question is the wrong instrument for the platform question.**

A denominator caveat, applied to the worst affected arm. Nine of the 76 probes check tool presence by discarding output with > /dev/null, which is not valid PowerShell, so they measure redirection rather than availability. Excluding all nine, PowerShell 5.1 scores 20 of 67 (29.9%) and PowerShell 7.6.6 scores 33 of 67 (49.3%). Both figures are quoted throughout as the contaminated ones, which is the conservative direction for the claim being made about Git Bash, and the corrected pair is stated here so a reader can use either.

Why this is not the expected result

The question in circulation is whether Windows is a worse place to run a coding agent. The implied model is that Windows is a rougher environment: different filesystem, different kernel, different conventions, and the agent suffers accordingly.

The data says the filesystem and the kernel barely register. Git Bash on NTFS, on Windows 11, on the same hardware that scores 26.3% under PowerShell, runs the model's default vocabulary at 97.4%. The operating system underneath it did not change between those two numbers.

> What changes is whether the environment speaks the language the model writes in.

That reframes the practical question completely. "Should I develop on Linux or Windows" has a 71 point answer hiding inside it that has nothing to do with the choice being asked about, and a 2.6 point answer to the question actually asked.

Where PowerShell loses them

Windows PowerShell 5.1, by category:

CATEGORY	OK OF TOTAL	WHAT BREAKS
quoting	6/6	Intact. The one clean category
redirection	3/6	/dev/null, 2>&1 against native commands
text processing	5/12	Coreutils resolve inconsistently, pipelines behave differently
glob	1/3	Expansion is not the shell's responsibility here
line endings	1/3	CRLF, including one case correct only after normalising
exit codes	**1/8**	$?, &&, ` , pipefail`
paths	2/8	Separators and traversal
find and xargs	0/5	Absent as the model knows them
coherence	0/7	Write then read back, in the POSIX idiom
variables	1/6	export, backticks, inline assignment
case	0/3	The filesystem behaviour in section 05
tools	0/9	**See the caveat below**

One of those rows is not what it looks like. The tools probes check that a binary exists by running it and discarding output with > /dev/null, which is not valid PowerShell. They are measuring the redirection, not the tool. Git and curl are present on this machine. The corpus conflates two causes in that row and the row should be read as evidence about redirection, not about tool availability. It is left in rather than quietly dropped, because a corpus written in one dialect will always have some of this and pretending otherwise would overstate the result.

The category that matters most is not the worst one

exitcode 1/8 deserves more weight than find 0/5.

A missing command announces itself. find: command not found is an unambiguous signal, the agent sees it, and the worst case is a wasted turn. That is friction, and friction is recoverable.

A wrong exit code is not friction. It is misinformation. A shell that reports success for a command that failed has told the agent something untrue at exactly the moment the agent is deciding what to do next, and the agent has no reason to doubt it. Every subsequent decision rests on it.

Seven of the eight exit code probes were wrong under 5.1. That is the single most consequential cell in the table, and it is not the one a survival percentage draws your eye to.

The honest size of the Git Bash result

97.4% is two failures, and both are the same underlying cause: filesystem case insensitivity. They are examined in section 05 because they belong to the operating system rather than to this section's argument.

What matters here is the shape. **The POSIX vocabulary is essentially portable to Windows provided something is there to interpret it**, and the two exceptions are not dialect problems. A shell cannot fix them and section 05 shows that none of the three Windows shells does.

04 What a version buys, and why it is mostly one operator

> Discharges claims G004, G024 to G030.

The 26.3% figure belongs to Windows PowerShell 5.1, which is the version that ships with Windows and the version the agent on this machine actually invokes. Quoting it alone would be unfair to the platform, so PowerShell 7.6.6 was installed and the corpus re-run against it with nothing else changed.

Survival goes from 26.3% to 44.7%. Fourteen commands recovered, 18.4 points, from a shell version.

The installation is worth describing because it bears on the recommendation: a portable per-user build, a zip extracted into a user directory and added to the user PATH. No administrator rights, no installer, no service, no reboot, and reversible by deleting one folder.

Where the fourteen came from

CATEGORY	5.1	7.6.6	CHANGE
coherence, write then read back	0/7	**6/7**	+6
text processing	5/12	9/12	+4
exit codes	1/8	3/8	+2
tools	0/9	1/9	+1
find and xargs	0/5	1/5	+1
quoting, paths, glob, variables, line endings, case	unchanged	unchanged	0

The coherence row was never about the filesystem

Six of the fourteen recovered commands are in one category, and the category is misleadingly named. The coherence probes write a file and immediately read it back: echo v > f && cat f, touch fresh.txt && find . -name "fresh.txt", echo v > src && mv src dst && cat dst.

They were meant to test whether a write becomes visible to the next command. Under 5.1 they scored 0 of 7, which looked like a serious filesystem or caching problem.

It was not. **&& is not an operator in Windows PowerShell 5.1.** It is a parse error. Pipeline chain operators arrived in PowerShell 7. Every one of those probes was failing at the syntax, before the filesystem was ever reached. Under 7.6.6, six of the seven pass unchanged.

This is worth dwelling on for a reason beyond PowerShell. **A whole category of a benchmark scored zero for a reason that had nothing to do with what the category was named after**, and the only way that surfaced was by varying one thing and watching which cells moved. A single arm study would have published "write then read back fails on Windows" and been confidently, reproducibly wrong.

What a version does not buy

Paths, globbing, variables, line endings and case behaviour are identical between 5.1 and 7.6.6. Zero commands recovered in any of them.

That boundary is informative. **A newer shell fixes syntax, not semantics.** && becomes legal; /dev/null does not become a device, export does not become a keyword, and two filenames differing only in case do not become two files. The things a version upgrade cannot touch are exactly the things that belong to the operating system and to the dialect gap rather than to the interpreter's age.

The conclusion survives the fairness check

44.7% is a much better number than 26.3% and it does not change the finding.

On the same machine, on the same 76 commands, Git Bash runs **97.4%** against a current PowerShell's **44.7%**. More than twice as much of the default vocabulary survives, and the comparison is between two programs installed side by side on one Windows box.

So: keep PowerShell current. It is free, it needs no administrator, it halves the failure rate of the commands a model writes by default, and && and || are how everybody writes shell anyway.

Then read section 06, because on the machine measured here that upgrade does not reach the agent at all.

05 The silent six

> Discharges claims G008, G011 to G012, G023, G031 to G041, G058, G064 to G066, G068.

Hold the shell constant, run a POSIX shell on both platforms, and ask what the operating system does differently. 24 commands.

Identical on both platforms	18
Diverged silently, exit 0 on both sides	**6**, of which 5 are unconfounded
Failed loudly on Windows	**0**

These are counts and must not be read as a rate. The 24 probes were chosen by looking where divergence was suspected, so the denominator is an artefact of where the author looked. "6 of 24" says nothing about how often an arbitrary command diverges, and no such figure is offered anywhere in this paper.

The zero is the finding, and the zero does not depend on the sampling. However the probes were chosen, not one of them produced a non-zero exit code on Windows.

Not one of the twenty four commands failed on Windows. Not one returned non-zero, printed to stderr, or gave an agent any signal that something had gone differently. Six of them did something other than what they did on Linux and reported success.

Every measure the study was originally designed around, failure rate, corrective reissues, turns to completion, would have scored these twenty four probes clean on both platforms.

The six

> 1. Two files become one, and the first one's contents are gone.

`echo one > Beta.txt ; echo two > beta.txt ; cat Beta.txt`
Linux prints one and counts two files. Windows prints two and counts one. The second write destroyed the first, because on NTFS they are the same file. Exit code 0, empty stderr.

An agent that ran this believes it created two files with different contents. Every later decision that depends on that belief is built on it.

> 2. cd into the wrong case succeeds.

mkdir RealDir then cd realdir. Linux refuses. Windows enters.

A path an agent is holding can be wrong and keep working indefinitely, until the same string reaches somewhere case matters: a Docker image, a CI runner, a deployment target, a case sensitive volume. The failure then appears a long way from the thing that caused it.

> 3. ln -s does not make a link. It makes a copy.

Linux reports symlink. Windows reports copy. The command returns 0 on both.

The agent believes a link exists, and therefore believes an edit to the target will be visible through it. It will not. Two files now drift apart, both plausible, both readable, one stale, with nothing anywhere recording that they were ever supposed to be the same file.

> 4. chmod is a no-op, and ls -l confirms the lie.

`chmod 600 f ; ls -l f`
Linux reports -rw-------. Windows reports -rw-r--r--. chmod returned 0.

The verification command reports the old mode, so an agent that checks its own work is told the check passed. This is the one to worry about wherever an agent writes a key, a token, a credentials file or a config it intends to restrict.

> 5. sed -i rewrites every line ending in the file.

A CRLF file of six bytes, after sed -i "s/a/x/", is six bytes on Linux and **four on Windows**. The edit asked for one character. It also converted the line endings of the entire file.

On a real repository that is a diff touching every line, attributed to a one character change, and it will be committed by an agent that has no reason to think it did anything else.

> 6. The same grep returns the opposite answer, and this one is confounded.

grep -c "abc$" against a line ending CRLF returns **0 on Linux and 1 on Windows**.

A follow up test isolates where the difference lives. The file is byte identical on both platforms, five bytes, b'abc\r\n', confirmed by reading it in binary. So nothing about the filesystem or the write path differs. The divergence is entirely inside grep.

> And the two arms do not run the same grep: 3.0 under Git Bash against 3.11 on Debian.

This paper cannot separate a version difference from the MSYS2 layer's text handling, because the versions were not pinned and matching them was not attempted. **Divergence six is therefore reported as a tool difference of unknown cause, not as an operating system behaviour**, and it should not be counted toward the platform claim.

It is left in because the practical consequence survives the confound: on these two machines, as an agent would actually find them, the same anchored match inverts. The cause matters for attribution and not for the person it happens to.

The other five are not confounded this way. sed is 4.9 on both arms and bash is 5.2.37 on both, so divergences one to five rest on identical tool versions.

Three of the six corrupt something

The case overwrite destroys a file's contents. chmod leaves a file more permissive than the agent believes. sed -i rewrites the whole file's line endings. The symlink leaves behind a copy that will keep being read as though it were live.

cd and grep do not corrupt anything directly. They feed the agent a false premise, which is arguably worse, because the damage is done later by code that was reasoning correctly from a wrong fact.

Windows is not broadly strange

The eighteen that matched are worth as much as the six that did not, and they are published in full rather than summarised away: trailing dots and trailing spaces in filenames, Windows reserved device names, colons in filenames, deleting a file while a handle is open, overwriting a file while a handle is open, a twenty level deep path, unicode filenames, hard links, the executable bit, removing a read only file, and binary data through a pipe. All identical.

The platform is not erratic. It is different in six specific places, and it is quiet in every one of them. That is a much more actionable finding than a general warning, and it is also a much harder one to notice, which is presumably why it is not already common knowledge.

What this does not establish

These are properties of the environment, measured without a model. They do not show that an agent falls into any of them.

A capable model may check ls -l after chmod and catch the discrepancy. It may know that MSYS2 copies rather than links. It may avoid case variants entirely, or write sed output to a new file instead of editing in place. Whether it does any of that is the question the next stage of this work exists to answer, and it is the only remaining question that costs anything to ask.

06 The gap between what you install and what the agent gets

> Discharges claims G024, G030, G042 to G045, G060.

Section 04 ends with a clean recommendation: update PowerShell, gain 18.4 points, free, no administrator rights. It is the obvious reading of the data and on the machine measured here it does not work.

Two binaries, by design

PowerShell 7 does not replace Windows PowerShell 5.1. It installs alongside it, under a different executable name, deliberately: powershell.exe is 5.1 and pwsh.exe is 7. Microsoft chose different names precisely so that upgrading could not break anything depending on the old one.

The consequence for an agent harness is direct. **A harness that invokes powershell.exe by name keeps getting 5.1 forever, no matter what else is installed.** Putting pwsh on the PATH changes what a human gets at a prompt and what scripts resolve; it does not redirect a program that asked for powershell.exe.

On the Windows system, after installing 7.6.6 and adding it to the user PATH, the session's own environment continues to report its PowerShell edition as Windows PowerShell 5.1, and a direct query confirms it resolves to C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe, version 5.1.26100.9444.

> So the operative figure for the agent as configured is 26.3%, not 44.7%.

Why this matters more than the eighteen points

A reader who takes section 04 at face value installs PowerShell 7, sees pwsh on their PATH, confirms the version, and reasonably concludes the problem is solved. Nothing visible contradicts them. The agent carries on emitting POSIX into a 5.1 interpreter and failing three commands in four, and the improvement they paid attention to never reached the thing it was meant to help.

That is the same shape as the six divergences in section 05: an action that appears to succeed, with verification that appears to confirm it, and no signal anywhere that the intended effect did not happen. **It is the paper's own finding, applied to the paper's own recommendation.**

The recommendation that actually follows

On a Windows machine where a POSIX shell is already present, the fix is not to improve the PowerShell path. It is to stop using it.

Git Bash on this machine runs 97.4% of the default vocabulary against PowerShell 7's 44.7% and 5.1's 26.3%. An agent that routes shell work through the POSIX tool is, on the evidence here, working in an environment statistically indistinguishable from Linux, on the same hardware, with no installation of any kind.

Three things follow, in descending order of effect:

1. **Prefer the POSIX shell tool for shell work.** This is worth 71 points and costs nothing. 2. **Keep PowerShell current anyway**, for everything that is not the agent: your own prompt, your scripts, your habits. && and || are worth having. 3. **If the harness must use PowerShell, point it at pwsh.exe explicitly.** The 18.4 points are real, and they are only collected by naming the binary.

The limit of this section

This is one machine and one harness configuration, observed on one date. Another setup may resolve PowerShell differently, and a harness may allow the interpreter to be configured, in which case point 3 is a setting rather than an obstacle.

What generalises is not the path. It is the question: **when a tool reports which shell it uses, check it against what you installed, because the two are allowed to disagree and nothing will tell you when they do.**

07 Limitations, and two measurements that were wrong

> Discharges claims G013 to G014, G016, G021, G046 to G047, G049 to G059, G061, G064 to G070, G081 to G082.

The limitation that bounds everything else

Sections 02 to 06 run no model. Those instruments execute commands; they do not choose them, so everything in them is a property of the environment rather than of an agent.

Section 08 closes part of that gap with seventeen sessions, and its own limits are stated there: three of the six divergences tested, three tasks, two models, no forced shell, and an existence proof rather than a rate. **The other three divergences remain untested against a model**, and nothing here supports a claim about how often an agent falls in, only that on these three it did not.

Where the paper speculates about what an agent would believe, as it does in sections 05 and 06, it is describing what the transcript would contain, not what was observed.

The rest, in order of how much they bound the result

One machine per platform. All Windows arms share one Windows system and the Linux arm is one Linux system. Three of the four arms sharing a machine is what makes the shell comparison strong; it also means the Linux to Windows comparison rests on two specific machines, two specific installations, on one date.

The corpus is written in one dialect. Probes are POSIX by design, because that is what a model emits, but it means PowerShell is being measured on how well it runs somebody else's language rather than on whether it can do the job. It can: every intent in the corpus is expressible in PowerShell. A native idiom corpus per shell would separate "the environment cannot" from "the model chose badly", and it has not been built. The tools 0/9 row in section 03 is where this bites hardest, and it is flagged in place.

The silent probe is 24 commands, not a census. Six divergences were found by looking in places where they were suspected. There is no basis for a claim about how many exist in total, and the sampling was directed rather than random.

Linux as ground truth is a choice, not a fact. Where the platforms disagree, Linux is called correct. For the case overwrite that is uncontroversial. For grep -c "abc$" against a CRLF line it is genuinely arguable. The raw results are published per platform so the comparison can be redone with the other convention.

No macOS. There is no Mac here. Nothing in this paper describes it.

No WSL2. The arm that would have run a Linux kernel on the Windows hardware, removing the machine from the comparison entirely, is absent because WSL is not installed. Its absence is why the Linux to Windows numbers rest on a calibration showing the two processors agree within 3%, which is good evidence and is not the same as no hardware difference at all.

Single run for the command corpus, and no variance figure. The survival percentages come from one pass per arm. The commands are deterministic and repeated running did not change them, but that is an assertion rather than a measurement, and no arm was repeated to produce an interval from the data itself. The Wilson intervals quoted in section 03 are binomial intervals on a single pass, which is a statement about sampling 76 commands, not about run to run stability.

The operating system effect on command survival is not statistically significant. 76 of 76 against 74 of 76 is a two sided Fisher exact p of 0.497. Section 03 states this rather than reporting 2.6 points as though it were a result, and the platform finding rests on the silent probe instead. A reader who wants the survival metric to settle the platform question needs a corpus an order of magnitude larger, and it would still be looking in the wrong place.

Toolchain versions were not pinned, and one of them matters. bash is 5.2.37 and sed is 4.9 on both POSIX arms, so most of section 05 is clean. **grep is 3.0 under Git Bash and 3.11 on Debian**, which means divergence six cannot be attributed to the operating system at all, and it is withdrawn from the platform claim in place. This was

found after the runs, by collecting version metadata that should have been captured by the instruments themselves. It now is.

Locale and code page were not controlled and differ on every arm. Debian is en_US.UTF-8, Git Bash has LANG unset with LC_CTYPE=C.UTF-8, and PowerShell reports code page 437. Character handling, collation and the unicode probes all depend on this. No probe result here is known to be caused by it, and none is known not to be.

One coder, no second reader. Every classification was applied by the author. Paper 01 of this programme reported an inter-rater reliability of 0.93 after a failed first round; this paper has no equivalent figure because nobody else has coded it. The mitigation is that the classifications are mechanical and the raw bytes are published, so disagreement can be checked rather than argued.

Two measurements that were wrong

Reported here because a paper that describes only the instruments that worked is describing a process that did not happen.

The calibration reported 81,000 durably written files per second. No storage device does that. The number reproduced perfectly across repeats, which is exactly why reproducibility is not validity. A direct test settled it: writing 300 files with fsync and without it took the same time, a ratio of 0.99, so the flush was never reaching the device and the measurement was timing the page cache.

The danger was not the wrong number, which was never published. It was that **this behaviour is not uniform across platforms.** Had the instrument gone on unchanged, a platform whose flush is real would have been compared against one whose flush is a no-op, and the difference would have surfaced as a large filesystem gap between operating systems. A fabricated headline, produced by the calibration instrument whose job is to prevent fabricated headlines. The disk measures no longer flush on any arm, and a diagnostic records per platform whether durability was measurable at all. It is reported, not corrected: whether a machine honours a flush is a property of that machine.

The reference arm scored 94.7% on its first run. Four failures, all four defects in the probes rather than in the environment, described in section 02. Running Linux first is what caught them.

Both errors were caught by looking at an output and asking whether it was believable, not by reading code. In both cases the code looked correct throughout, and in both cases the result reproduced.

The conflict of interest

The author uses the tool under test daily, and this paper and the programme it belongs to were produced with it. The object of study and the instrument of production are the same product family.

That is disclosed here rather than in a footnote, and it is also the reason both instruments are scripts with published source and published raw output. Nothing in the results depends on the author's judgement of a transcript, and anyone who suspects the conclusion can re-run seventy six commands and check.

08 The paid stage: does the agent fall in?

> Discharges claims G054, G058, G060, G071 to G082.

Everything up to here is the floor: what the environment executes, measured without a model. The question a reader actually has is whether an agent walks into the six divergences or routes around them, and that question costs model time to answer.

Seventeen sessions answer it for three of the six. Each ran in a fresh directory, capped at fourteen turns, on a pinned model, given a task phrased the way a user would phrase it and never mentioning the trap.

ARM	MODEL	SESSIONS
Windows, Git Bash	Haiku 4.5	8
Linux, bash, control	Haiku 4.5	6
Windows, Git Bash	Opus 5, effort low	3

The result

TASK	WINDOWS, HAIKU	LINUX, HAIKU	WINDOWS, OPUS LOW
Two filenames differing only in case	2 of 2 **noticed the collision** and reported it unprompted	2 of 2 produced two files with distinct contents	1 of 1 noticed
Restrict a secrets file so only the owner reads it	4 of 4 **restricted it**, with icacls	2 of 2 restricted it, mode 0o400	1 of 1 restricted it, with icacls
Edit a CRLF file in place, change nothing else	2 of 2 **kept all three CRLF pairs**	2 of 2 kept them	1 of 1 kept them

Not one session typed chmod on Windows. The agent reached for the platform's own tool without being told to, and the access control list verifies as restricted to the owner with no broad principal. On the case task it checked its own work closely enough to see a file count that did not match its intent, and said so.

The Linux control does what a control should: the tasks are trivially solvable where the traps do not exist. On Windows the same outcomes were reached by different means.

So the floor is hostile and the agent does not stand on it. That distance is the finding of this section, and it runs in the environment's favour.

What it does not show

Three tasks is an existence proof, not a benchmark. Seventeen sessions cannot estimate a rate and none is offered here. It shows the traps are navigable by these two models on these tasks, not that they are always navigated, and it says nothing at all about the other three divergences, which were not tested.

No shell was forced. The agent chose its own tools, which is the realistic condition and also means this does not isolate Git Bash from PowerShell. An agent pushed into PowerShell might fare very differently, and that is untested.

Effort was low for the Opus sessions, so nothing here compares models at their strongest, and the sample could not support that comparison anyway.

The instrument fell into its own trap

Four measurement errors in this stage, and one of them is the paper in miniature.

The remote arm measured the file transfer. The Linux sessions were originally inspected by copying the finished directory back to Windows. Alpha.txt and alpha.txt then collided *during the copy*, and the permission check ran icacls against a Linux run. The Linux arm faithfully reported Windows behaviour, with an exit code of zero and a plausible-looking table.

That is divergence one, happening to the instrument built to measure divergence one. Silently, in the same direction, producing a result that looked right. State is now read on the machine that produced it and the first Linux numbers are void.

The benchmark was contaminated by its own author. The pilot session answered correctly and cited a note in the machine's CLAUDE.md telling it that Windows filenames are case insensitive, which had been written an hour earlier after reading the oracle's output. The benchmark was measuring whether the agent could read a summary of the answer. Every session reported above ran with that section removed and restored afterwards, by the runner, in a finally.

The permission check read the wrong thing. The first chmod runs scored 0 of 2 on POSIX mode bits reading 0o666, which on Windows they always do whatever the access control list says. The agent was right and the check was wrong. The tell was a transcript saying icacls beside a score saying failure.

The Linux arm was unauthenticated for six sessions, which returned Failed to authenticate and a non-zero exit. Those were recorded rather than quietly dropped, and re-run after the session was restored.

None of the four was caught by reading code. All four were caught by looking at an output and asking whether it could be true.

What this changes about the rest of the paper

Nothing in sections 03 to 06 moves. A default POSIX vocabulary still executes at 26.3% under the PowerShell the tool invokes, and six operations still return success while doing something else. Those are properties of the environment and they were measured without a model.

What changes is the conclusion a reader should draw from them. **The hostility of the environment is not, on this evidence, transmitted to the work.** A capable model does not blindly emit the default vocabulary on Windows. It adapts, and on the three traps tested it adapted every time.

The recommendation in section 06 stands anyway, because it costs nothing: routing shell work through the POSIX tool removes the friction rather than relying on the model to compensate for it every time.

09 What follows

> Discharges claims G054, G058, G060.

For anyone running a coding agent on Windows, today

1. Route shell work through the POSIX shell. On the machine measured here that is 97.4% against 26.3%, it is worth more than every other recommendation combined, and it requires installing nothing.

2. Check which shell you are actually getting. Not which one you installed. They are allowed to disagree, powershell.exe and pwsh.exe are different binaries by design, and nothing announces the mismatch. Section 06 is an entire section about an improvement that never reached the thing it was meant to help.

3. Keep PowerShell current regardless. 18.4 points, free, no administrator rights, and && is worth having for its own sake.

> 4. Treat six specific operations as unreliable on Windows, none of which will tell you.

Filenames differing only in case, cd into a path whose case is wrong, ln -s, chmod, sed -i against a CRLF file, and end anchored grep against CRLF. Each returns success.

5. Do not choose an operating system on this evidence. The platform difference is 2.6 points of command survival and six quiet behaviours. If you are choosing for other reasons, choose for those reasons.

For anyone benchmarking an agent

Build the control before the first result. Both instruments here are controls: a reference arm that must score 100% before any other number means anything, and a shell held constant so that a difference can be attributed. The reference arm caught four defects that would otherwise have been distributed across four platforms and read as environment differences.

Ask whether a number is believable before asking whether it reproduces. Both errors in section 07 reproduced perfectly. One of them was physically impossible.

Vary one thing and watch which cells move. A whole category scored zero under PowerShell 5.1 for a reason that had nothing to do with what the category measured. Only the version comparison revealed it. A single arm study would have published "write then read back fails on Windows" and been reproducibly wrong.

Count what does not fail. The most valuable measurement in this paper is a zero: no command failed loudly. A study instrumented only for failures, reissues and turns would have recorded those twenty four probes as clean on both platforms and concluded there was nothing there.

What this study still owes

The model has not run. Everything here is the floor. The open question is whether an agent walks into these traps or routes around them, and it is the only remaining question that costs anything. The design for it exists: a small task suite, a fixed cheap model, a hard budget governor, and a family of tasks that walks an agent into the case collision and the CRLF edit deliberately and grades whether it noticed rather than whether it succeeded.

The silent class needs its own detector. Not a failure rate. An acceptance script that reads what is on disk and compares it against what the transcript claims was put there, and a count of the disagreements. Nothing else in the conventional set of agent metrics can reach it.

Two arms are missing. WSL2 would remove the hardware question entirely by running a Linux kernel on the Windows machine. A native idiom corpus per shell would separate an environment that cannot do something from a model that asked for it the wrong way. Both are cheap and neither has been done.

The one line version

The operating system is worth 2.6 points and the shell is worth 71, so the question almost everybody asks has a much smaller answer than the question almost nobody asks. What the operating system does contribute does not appear in a failure rate at all: six commands that returned success and did something else, three of which corrupt a file, and none of which any retry counter, success rate or completion time could ever see.

ABOUT THE RECORD

Two instruments, neither of which invokes a language model. A corpus of 76 commands with known POSIX results, and 24 probes run with the shell held constant. Every command, exit code, stdout and stderr is published.

WHAT IT SHOWS

The shell decides almost everything and the operating system is not visible in the failure metric at all. What the platform does contribute returns exit code zero: a file silently overwritten, a symlink that is a copy, a chmod that does nothing and a sed that rewrites every line ending. Given those traps unannounced, two models routed around all three that were tested, so the environment is hostile and the agent compensates.

CORRECTIONS AND CONFLICTS

The draft claimed a 2.6 point platform effect that a Fisher exact test does not support, and attributed one divergence to Windows when it belongs to a grep version difference. Both are corrected in the body. The author uses the tool under test daily and this paper was produced with it.

© 2026 Nikolaos Broikos. Text and data under CC BY 4.0, code under MIT. The data, code and claim ledger are in data/paper-12/ so any figure here can be recomputed, checked or disputed.