CORRECTABLE MEANS CORRUPTIBLE: THE REASONING CHAIN OF VISION-LANGUAGE-ACTION POLICIES UNDER ADAPTIVE ATTACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-language-action policies map camera images and natural-language instructions to a robot's motor actions. Some of these policies reason before they act, writing a reasoning chain in text and decoding their actions conditioned on it. That chain is offered as an interface that a person can read to understand the robot's behavior and rewrite when the behavior is wrong. An attacker who reaches the chain can rewrite it too. We therefore ask whether the chain can be exposed as an interface and still be secured by monitoring it. We answer with two measurements on DeepThinkVLA and CoTinyVLA, two such policies that write their chains at different times, in four simulated task suites of the LIBERO benchmark. In the first measurement, a chain corrupted by renaming its objects lowers task success on every suite of both policies, even with an intact instruction. Conversely, injecting the clean chain with a corrupted instruction recovers part of what that instruction cost. On both policies, the chain that corrects the robot can also corrupt it. In the second measurement, two detectors that read only the chain catch the corrupted chains when the attacker does not know them. Against an adaptive attacker that knows what they check, neither detector scores above chance on either policy. In every case the attack keeps the corruption's harm to task success. Yet on every feature the detectors check, the attacked chains look at least as clean as the clean chains. We call this outcome disguised corruption. We argue that detection over features read from the reasoning chain fails as a class whenever an attacker can achieve disguised corruption against the features it reads. This argument instead points to a detector that reads a channel the attacker does not write. We therefore pre-register an action-only task-consistency detector with an attacker built against it.

1 INTRODUCTION

Vision-language-action policies build on vision-language models pretrained at internet scale, mapping camera images and a natural-language instruction to a robot's motor actions (Zitkovich et al., 2023; Kim et al., 2025; Black et al., 2025b). The pretrained model lets such a policy generalize beyond its robot training data, to new objects, to commands that data never contained and to simple semantic reasoning (Zitkovich et al., 2023). Several of these policies also generate an intermediate reasoning step before they act (Zawalski et al., 2025; Zhao et al., 2025b; Yin et al., 2026). When that step is written in text, it forms a reasoning chain that a person can read before the robot moves (Zawalski et al., 2025; Yin et al., 2026). The work that introduced such chains presents them as a way to understand the decisions a policy makes, including why it failed, and to correct its behavior by editing a faulty chain (Zawalski et al., 2025). The chain is thereby offered as an interface that an operator reads to see what the policy will do and rewrites when it is wrong.

An interface that an operator can rewrite is also one that an attacker can rewrite. Such an attacker needs only to control the text between the step that writes the chain and the step that acts on it. Against that attacker, the natural defense is to monitor the chain and to flag any chain that does not fit the task. Whether such a monitor can be evaded is disputed in studies of language models. On one side, when a harmful task cannot be done without reasoning, models evade a monitor only with substantial help (Emmons et al., 2025). On the other, rewriting the reasoning alone, with every action left unchanged, sharply lowers a monitor's catch rate wherever the reasoning is its only

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

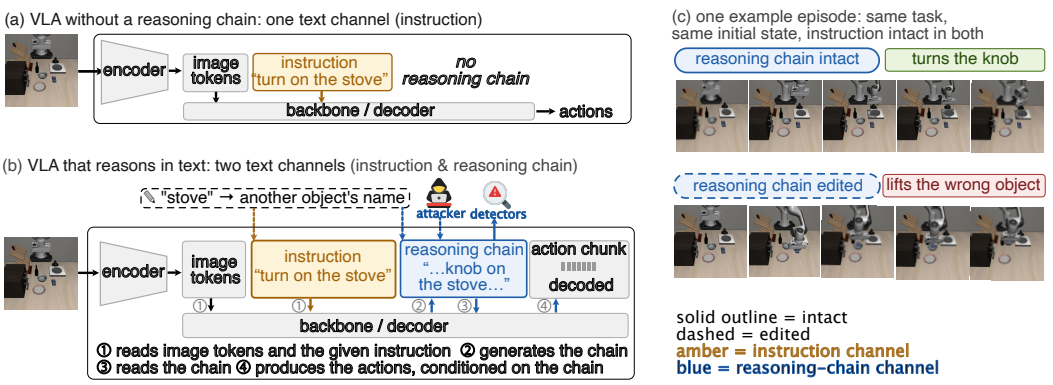


Figure 1: **(a)** A vision-language-action (VLA) policy without a reasoning chain reads one text channel, the instruction. **(b)** A VLA policy that reasons in text first writes a reasoning chain and then produces its actions conditioned on it, so a second text channel reaches the actions. Our entity swap can edit either channel. An attacker can rewrite the chain after the policy writes it, and both detectors read only the chain. DeepThinkVLA writes a chain per action chunk; CoTinyVLA writes one Plan per episode and a Think per chunk. **(c)** One example episode of DeepThinkVLA, run with the chain intact and with the chain edited by the entity swap, which here rewrites *stove* as *cream cheese box*; with the edited chain, the policy lifts the cream cheese box. Frames are rendered from the episode's stored states.

signal (Shiromani & Richter, 2026). Both studies concern language-model agents rather than robot policies. We therefore ask whether the reasoning chain of a vision-language-action policy can be exposed as an interface and still be secured by monitoring it.

We answer this question with two measurements on two released policies that write their chains in text, DeepThinkVLA (Yin et al., 2026) and CoTinyVLA (Lee et al., 2026), across four task suites of the LIBERO benchmark (Liu et al., 2023). The two policies write their chains at different times, DeepThinkVLA each time it is queried for actions and CoTinyVLA once per episode, so a result found on both replicates across that difference in design. Both policies run in simulation, because only a simulator can start the runs the measurements compare from identical initial states. The first measurement corrupts either the instruction or the chain with an entity swap that renames every object the edited text mentions (Figure 1). Applied to each text in turn, the swap shows whether the chain can corrupt or correct the robot's actions. The second measurement asks whether a detector that reads only the chain can see an attacker who writes it. Its threat model comes from the first measurement rather than being assumed.

Together, the two measurements support three conclusions.

- Corrupting the chain under an intact instruction lowers task success on both policies and all four suites. Conversely, injecting the clean chain under a corrupted instruction recovers part of what that instruction cost, so whoever writes the chain steers the robot.

- Two chain-reading detectors catch the corruption, yet score no higher than chance against an adaptive attacker who knows them, with the harm retained.

- On every feature the detectors check, the adaptive attack's chains look at least as clean as clean chains while keeping that harm, an outcome we call disguised corruption. We argue that detection over features read from the reasoning chain fails as a class whenever an attacker can achieve disguised corruption against the features it reads. This argument instead points to a detector that reads a channel the attacker does not write, so we pre-register an action-only task-consistency detector with an attacker built against it.

2 RELATED WORK

Reasoning in vision-language-action policies. Reasoning before acting takes several forms in vision-language-action policies. Embodied chain-of-thought reasoning trains a policy to write the plan, subtasks and motions of a task in text before each action (Zawalski et al., 2025). Other policies reason in different forms, predicting future images as visual subgoals (Zhao et al., 2025b), reasoning over coarse action trajectories (Zhong et al., 2026), or conditioning a fast action module on a slower vision-language module (Chen et al., 2025; NVIDIA et al., 2025). Both policies we study write text (Yin et al., 2026; Lee et al., 2026), the form a person can both read and edit directly. Their authors already corrupt the chain, replacing it with random or contradictory text, and find that success drops (Yin et al., 2026; Lee et al., 2026). We add the opposite direction, a clean chain injected under a corrupted instruction, and compare runs in which the chain is the only thing we change.

Attacks that reach the reasoning chain. The attacks on robot policies reviewed here all enter through channels other than the reasoning chain. They perturb the camera image (Wang et al., 2025), add optimized text to the instruction (Jones et al., 2025), implant a backdoor during training (Zhou et al., 2025) or jailbreak robots controlled by language models through the user's commands (Zhang et al., 2024; Robey et al., 2025). One of them, TRAP, reaches the chain indirectly, through an adversarial patch in the scene (Huang et al., 2026). Pairing chains with mismatched instructions, its authors find that whether the actions follow the chain or the instruction depends on the model. Our first measurement asks the same question with a causal design. In language models, by contrast, the chain itself is the target, corrupted through the prompt, the input image or the training data (Xu et al., 2024; Kuo et al., 2025; Zhao et al., 2026; Wang et al., 2024; Zhao et al., 2025a; Chaudhari et al., 2026; Foerster et al., 2025). There, the consequence is a wrong answer in text rather than a robot's action.

Monitoring the reasoning chain. Reading a model's chain of thought (Wei et al., 2022) to catch misbehavior is the premise of chain-of-thought monitoring (Korbak et al., 2025; Baker et al., 2025), where the chain is called a reasoning trace. That premise needs the chain to reflect what the model computes. Stated reasoning does not always do so (Turpin et al., 2023; Lanham et al., 2023; Arcuschin et al., 2025). Even where it does, monitoring holds when the harmful behavior needs written reasoning (Emmons et al., 2025), and fails when a rewrite of the reasoning alone is enough (Shiromani & Richter, 2026). Pressure from the monitor can also break it, since a model trained against a monitor can learn to hide its intent in the chain (Baker et al., 2025). Korbak et al. (2025) accordingly describe the chain's monitorability as fragile. Yet a recent monitor built for reasoning safety, a language model that checks each reasoning step against a taxonomy of errors, is evaluated only against attackers with no access to it (Wang et al., 2026). Its authors leave robustness to adaptive attackers for future work. Our second measurement tests chain monitoring against an attacker who knows the monitor, on robot policies. Its detectors follow fixed rules, so a learned monitor like this one is the case it does not reach. Further work is discussed in Appendix G.

3 TWO MEASUREMENTS OF THE REASONING CHAIN

Our question is whether the reasoning chain can be exposed as an interface and still be secured by monitoring it. Answering it takes two measurements on the same policies. The first measures whether a corrupted chain lowers task success under an intact instruction, and whether a clean chain recovers it under a corrupted one. The second measures whether detectors that read only the chain can tell clean chains from chains that an attacker has written.

3.1 CORRUPTION AND CORRECTION THROUGH THE CHAIN

Policies and suites. Both measurements need a policy that writes its reasoning chain in text, which a person can read and edit before the actions are decoded. We study two such policies, which differ in when they write the chain. DeepThinkVLA (Yin et al., 2026) writes a new chain each time it is queried for actions, then decodes the next chunk of actions from the camera frames, the instruction and that chain. CoTinyVLA (Lee et al., 2026) writes a Plan, an ordered list of subgoals, when an episode begins, then a short Think before each chunk of actions. It then decodes the chunk from the frames, the instruction, the Plan and the Think. The Plan is CoTinyVLA's reasoning chain throughout the paper, and the text we edit. Because the two policies write their chains at different

instruction *turn on the stove* left intact by both attacks

reasoning chain, written at the first query of a clean episode

<think>This is the initial frame. The **stove** (right-middle) is on the table. Its knob (right-middle, on **stove**) needs to be manipulated to turn the **stove** on. The robot arm (top-center) is positioned near the **stove**, ready for interaction. The immediate goal is to rotate the knob. Turn the knob on the **stove**.</think>

non-adaptive attack (condition B's swap): every **stove** → **cream cheese box**

adaptive attack: the swapped chain, with one sentence appended: "The target object is the **stove**."

Figure 2: The two attacks on one reasoning chain, written at the first query of the episode in Figure 1(c). The non-adaptive attack renames the stove at every mention; the adaptive attack keeps that swap and appends one sentence naming it, so the entity check finds every object it looks for.

times, a result found on both holds across that difference, so we read the second as a replication of the first, never as a comparison. Both were trained on the four task suites of the LIBERO benchmark, or on variants of them (Liu et al., 2023). Each suite, Spatial, Object, Goal or Long, has 10 tasks. We run the policies' released checkpoints on these suites without further training, so a change in success reflects our edit rather than an unfamiliar task. Each policy's inputs, the episode protocol and the swap's name pool are in Appendix A.

The entity swap and conditions A to D. Our edit can reach either of the two text channels, the instruction the policy reads or the chain it writes. Such an edit must change which object a channel names, so that the text describes a different task, and it must fit both channels. The instruction, a short sentence, is the harder channel to fit. Reordering or perturbing its words breaks it rather than redirecting it, whereas replacing an object's name leaves it fluent and wrong only about the object. The entity swap makes that replacement, mapping each object a channel mentions to another benchmark object's name, the same at every occurrence, and leaving the rest unchanged (Figures 1 and 2). The swap defines four conditions. Condition A leaves the instruction and the chain intact. Condition B keeps A's instruction but corrupts the chain. A minus B, the drop in success that we call corruption, is what a corrupted chain costs. Condition C corrupts the instruction, and with it the chain the policy writes. A minus C, the cost of the corrupted instruction, is what a clean chain could recover. Condition D keeps C's instruction but gives the policy a clean chain in place of the one it would write. D minus C, which we call correction, is the part of that cost the clean chain does recover. We report correction beside the cost and, on CoTinyVLA, also as a share of that cost.

Why D minus C measures the chain. Correction reads as the chain's effect only if C and D differ in the chain and in nothing else. By construction they do, since they share the corrupted instruction byte for byte, the initial states and the query schedule. The one part that differs, D's injected clean chain, is the policy's own text, written from the clean instruction, so it stays grounded in the scene where it is used. If the chain merely restated what the cameras show, injecting it would change nothing, so D would sit at C's level. If the actions depend on what the chain says, D would move back toward A. Goal separates these two predictions most sharply. Its tasks share one scene, so what the cameras show is the same for every goal. Once the instruction is corrupted, the injected clean chain is the only input we supply that still states the intended goal. Goal is therefore where a corrupted instruction should cost most, leaving the chain the most to recover. The other three suites, where the scene narrows the task, measure how far the effect extends beyond it.

3.2 Detection under attack

The two detectors. The second measurement asks whether a detector that reads only the chain can see an attacker who writes it. We test two detectors, instruments for this question rather than proposed defenses, built from rules fixed before any episode ran and fitted to no data. The entity check scores one feature, the share of the instruction's objects that the chain also names. The fixed four-feature consistency detector averages that feature with three more. Direction checks each direction word in a motion phrase, such as *move left*, against the way the arm actually moves on that axis. Gripper checks the chain's first gripper word, such as *grasp* or *release*, against whether the gripper actually closes or opens. Parseability is the share of the chain's sentences that contain a word the detector can read. A feature with nothing to check in a chain scores clean and attacked chains alike,

Table 1: Task success, in points, under conditions A to D on both policies and four suites, with corruption (A minus B) and correction (D minus C) in the last two rows. The two policies are replications and are not ranked against each other.

			DeepThinkVLA				CoTinyVLA			
	instruction	chain	Spatial	Object	Goal	Long	Spatial	Object	Goal	Long
A	intact	intact	95.0	99.0	94.8	94.0	94.7	99.7	99.2	92.7
B	intact	corrupted	89.0	94.3	75.7	84.8	27.8	52.2	15.8	18.2
C	corrupted	written from it	60.8	74.5	12.5	64.7	10.2	36.2	7.5	5.3
D	corrupted	clean	88.8	95.2	62.3	85.8	76.5	99.0	96.3	83.5
A−B	corruption		6.0	4.7	19.2	9.2	66.8	47.5	83.3	74.5
D−C	correction		28.0	20.7	49.8	21.2	66.3	62.8	88.8	78.2

so we report, for each feature, how often it had something to check. We call these four the checked features. The exact rule for each feature, attack and verdict is in Appendix C.

The two attacks. The checked features follow fixed rules, so an attacker can know them. Such an attacker is the standard by which defenses against adversarial examples are judged (Carlini & Wagner, 2017; Carlini et al., 2019; Tramer et al., 2020). We therefore test both detectors against two attacks. The non-adaptive attacker knows nothing of the detectors and applies the swap of condition B. The adaptive attacker knows both detectors. Its attack, also a fixed rule, keeps that swap and adds only the text each checked feature needs in order to read clean. In its basic form, the adaptive attack appends one sentence naming the instruction's objects, so the entity check finds every object it looks for (Figure 2). A second form of it, the adaptive attack with the direction flip, also flips each direction word in a motion phrase that disagrees with the arm's motion under the clean chain. An attacker can always hide by abandoning the harm, so every detector figure is read beside the attack's task success.

How a detector is read. Each detector is read by its area under the receiver operating characteristic curve (AUROC) between clean and attacked episodes, computed per task and averaged over tasks, with an interval from a bootstrap over tasks (Efron & Tibshirani, 1994). The AUROC that counts as chance is read from a null, the same detector scored on two clean runs of the policy, rather than assumed. The verdict words for each detector's AUROC under an attack were fixed before any score existed. Under the non-adaptive attack, a detector is caught when the lower end of its interval reaches 0.90, partly caught when its interval clears the null but not that bound, and not caught when it overlaps the null. Under the adaptive attack, an interval that overlaps the null reads chance, and one wholly above it reads not evaded. One wholly below it reads inversion for the fixed four-feature consistency detector and at or below chance for the entity check. Beside these verdicts, we read the attack's harm from its task success. It is retained when the interval of that success overlaps the non-adaptive attack's, lost when it overlaps the clean episodes', and reduced in between.

4 Experiments

Task success is read in both measurements. Section 4.1 reports how far corruption and correction move it on the two policies. That result gives the threat model under which §4.2 tests the two detectors, reading each beside the attack's task success. Section 4.3 then states the pre-registered action-only test, whose detector reads a channel the attacker does not write.

4.1 Corruption and correction on both policies

DeepThinkVLA. On DeepThinkVLA, corrupting the chain under an intact instruction (condition B) lowers task success on every suite, from 4.7 points on Object to 19.2 on Goal (Table 1). Each of these drops, reported without a test, has an interval above zero. In the opposite direction, we corrupt the instruction and give the policy a clean chain (condition D), in place of the chain it would write from that instruction (condition C). The clean chain raises success on every suite, from 20.7 points on Object to 49.8 on Goal. The rise means that the actions depend on what the chain says, since a chain that only restated what the cameras show would have left D at C's level. Nor is the rise

carried by a few tasks. We fixed the rescue test before the runs to ask exactly this, with Goal as its confirmatory case. It finds D above C on every Goal task (exact one-sided $p = 0.0010$). Applied to each suite in an order also fixed before the runs, it finds the rise on all 4. However, the clean chain does not recover the whole cost. D stays below A, with instruction and chain intact, on every suite, and furthest on Goal. The chain therefore moves the actions without carrying all of what the instruction does. We call that correction partial. Appendix B gives the test order, the per-suite statistics and the size of each gap.

CoTinyVLA. CoTinyVLA shows the same pattern in both directions (Table 1). Corrupting its Plan under an intact instruction (condition B) lowers success on every suite, from 47.5 points on Object to 83.3 on Goal. Pooled, this drop is 68.0 points [60.6, 75.1]. Conversely, the clean Plan injected under a corrupted instruction (condition D) raises success over the Plan the policy writes from that instruction (condition C) on every one of the 40 tasks. The rescue test, fixed before this policy's runs, confirms that the rise spans its tasks. As a share of the cost, the clean Plan recovers 0.906 [0.851, 0.952] pooled, with a per-suite interval above zero on all 4 suites. We read these figures as a replication of DeepThinkVLA's pattern, not a comparison of sizes, because the two policies write their chains at different times.

What the measurement implies. Both results rest on comparisons paired by task within one set of runs, because two identical clean runs can end differently on some episodes (Appendix A). On both policies, the chain moves the robot's actions in both directions. Whoever writes the chain therefore steers the robot. The second measurement's attacker follows from this result rather than being assumed. It controls the text between the step that writes the chain and the step that acts on it.

4.2 Detection under the two attacks

The non-adaptive attack. The non-adaptive attacker knows neither detector. Its attack is the same swap as condition B, so the detectors face the corruption the first measurement studied. By the rules fixed before any score, both detectors catch it on every suite of both policies (Table 2). Pooled, the fixed four-feature consistency detector scores an AUROC of 0.997 on DeepThinkVLA and 0.998 on CoTinyVLA, far above the null in the table. The attack they catch still does harm, lowering pooled task success from 95.7 to 86.1 points on DeepThinkVLA and from 96.5 to 28.5 on CoTinyVLA. Both detectors therefore work against this corruption. If they fail under the adaptive attack, it will not be because they could never catch a corrupted chain. A deployer who stopped here would conclude that a corrupted chain is visible to a detector that reads it.

The adaptive attack. The adaptive attacker tests that conclusion with knowledge of both detectors. Its attack keeps the same swap and adds only the text each checked feature needs to read clean. Under it, neither detector scores above chance on any suite of either policy (Table 2), where chance means that the detector cannot tell attacked chains from clean ones. The fixed four-feature consistency detector does worse than chance on DeepThinkVLA. It rates the attacked chains as more consistent than the clean ones on every suite, a reading we call inversion, with a pooled AUROC of 0.228 [0.195, 0.263] against a null of 0.491. On CoTinyVLA the same detector reads chance on Goal, Object and Long, and inversion on Spatial. Meanwhile the attack's task success stays at the non-adaptive attack's level on every suite of both policies, at 87.3 points pooled on DeepThinkVLA and 38.6 on CoTinyVLA. So harm is retained in every cell, three of them on DeepThinkVLA by a tie-break added after the run (Appendix C). Adding the direction flip changes no verdict on DeepThinkVLA. On CoTinyVLA it produced the same text as the adaptive attack on every episode, so the two forms are one attack there. Both detectors thus lose their discriminative power under an attacker built against them, while the attack keeps the harm it was built to carry.

Which features had something to check. Which features had something to check sets how far the next result reaches, so we report it descriptively, as §3.2 committed, in Table 12 in Appendix C. On DeepThinkVLA the entity check and parseability had something to check on nearly every query, gripper on most, and direction on almost none. On CoTinyVLA gripper and parseability also had something to check on every Plan, yet scored the clean and the attacked Plan of a task alike, parseability excepting one Object task. Direction had something to check on one Long task only.

Table 2: AUROC of both detectors per suite and pooled under the non-adaptive and the adaptive attack, beside the null of two clean runs. DeepThinkVLA carries the adaptive attack with the direction flip; CoTinyVLA does not, because the flip produced the same text as the adaptive attack. The attack's task success is beneath each policy's block. Verdicts were fixed before any score; a dagger marks a pooled cell whose suites disagree. Per-suite intervals are in Appendix C.

	attack	Spatial	Object	Goal	Long	pooled
DeepThinkVLA						
fixed four-feature	non-adaptive	0.998 caught	0.999 caught	0.997 caught	0.995 caught	0.997 [0.995, 0.999]
fixed four-feature	adaptive	0.228 inversion	0.218 inversion	0.250 inversion	0.217 inversion	0.228 [0.195, 0.263]
fixed four-feature	adaptive + flip	0.230 inversion	0.192 inversion	0.246 inversion	0.161 inversion	0.207 [0.172, 0.245]
fixed four-feature	null	0.476	0.500	0.489	0.502	0.491 [0.478, 0.505]
entity check	non-adaptive	0.997 caught	1.000 caught	1.000 caught	1.000 caught	0.999 [0.998, 1.000]
entity check	adaptive	0.346 at or below chance	0.498 chance	0.497 chance	0.394 at or below chance	0.434† [0.397, 0.465]
entity check	adaptive + flip	0.346 at or below chance	0.498 chance	0.497 chance	0.394 at or below chance	0.434† [0.397, 0.465]
entity check	null	0.476	0.500	0.500	0.505	0.495 [0.486, 0.504]
task success	clean	95.0	99.0	96.2	92.7	95.7
task success	non-adaptive	88.3	94.3	77.0	84.8	86.1
task success	adaptive	89.2 harm retained	94.8 harm retained	79.8 harm retained	85.3 harm retained	87.3
task success	adaptive + flip	89.7 harm retained	94.8 harm retained	80.0 harm retained	85.3 harm retained	87.5
CoTinyVLA						
fixed four-feature	non-adaptive	0.996 caught	1.000 caught	0.997 caught	1.000 caught	0.998 [0.997, 1.000]
fixed four-feature	adaptive	0.150 inversion	0.450 chance	0.400 chance	0.460 chance	0.365† [0.290, 0.430]
fixed four-feature	null	0.500	0.500	0.500	0.500	0.500 [0.500, 0.500]
entity check	non-adaptive	0.996 caught	1.000 caught	0.997 caught	1.000 caught	0.998 [0.997, 1.000]
entity check	adaptive	0.150 at or below chance	0.500 chance	0.400 chance	0.450 chance	0.375† [0.312, 0.438]
entity check	null	0.500	0.500	0.500	0.500	0.500 [0.500, 0.500]
task success	clean	94.7	99.7	99.2	92.7	96.5
task success	non-adaptive	27.8	52.2	15.8	18.2	28.5
task success	adaptive	46.2 harm retained	61.2 harm retained	24.3 harm retained	22.8 harm retained	38.6

Why the weighting does not matter. Where the attacked chain matches or beats the clean chain on every checked feature, the failure does not depend on how a detector weights those features. To state this, take a set F of checked features, each scored in $[0, 1]$ with 1 read as clean, and pair each attacked episode of a task with each clean one. On DeepThinkVLA, whose episodes have many queries, an episode's value on a feature is the feature's mean over those queries. Let s be the share of pairs (a, c) with $\phi(a) \geq \phi(c)$ for every $\phi \in F$. *Proposition.* Any detector that never rates an episode as less clean when one of the features in F rises, which includes every non-negative weighting of those features, has on that task

$$\mathrm{AUROC} \leq 1 - s/2. \tag{1}$$

On a pair in which the attacked episode matches or beats the clean one on every feature of F, such a detector rates the attacked episode at least as clean. The AUROC counts that pair as zero, or as half on a tie, and every other pair as at most one. When $s = 1$ on every task of a suite, the bound equals chance, so no such detector can score above chance there. We state the proposition only in that

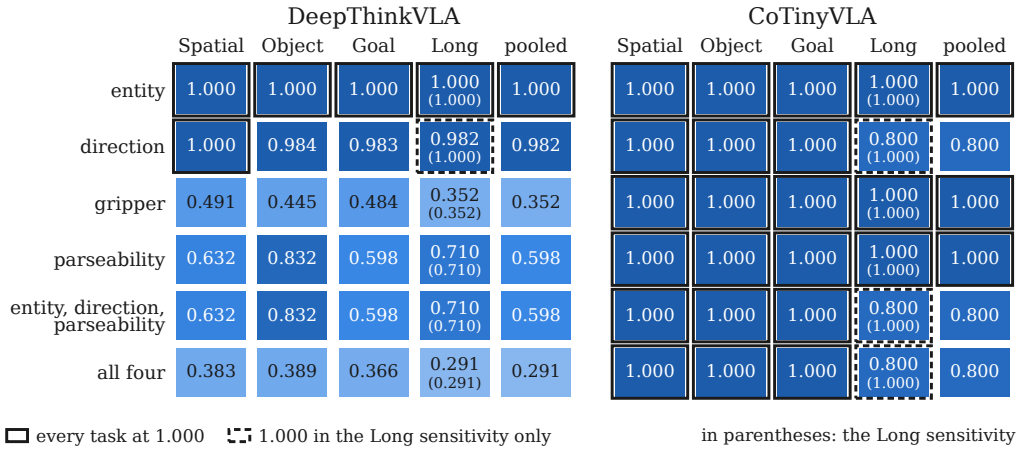


Figure 3: In each cell, the share of pairs in which the attacked chain matches or beats the clean chain on every feature of the set, taking the lowest share among the tasks in its column. The attacked chains come from the adaptive attack with the direction flip. A solid outline marks a cell where every task reaches 1.000. A dashed outline marks a Long cell that reaches 1.000 only in the Long sensitivity of Appendix C, whose value is in parentheses; every feature set is in Appendix D.

case, a rule corrected and then amended after the runs (Appendix D). Elsewhere, Equation 1 at the smallest s over the tasks of a suite, or of all suites pooled, gives a ceiling the detector cannot exceed. Figure 3 gives these shares under the adaptive attack with the direction flip. The proposition holds on CoTinyVLA over all 4 checked features on Goal, Object and Spatial, and on Long in the sensitivity form of Appendix C. On DeepThinkVLA no set of three or four features reaches $s = 1$, so the best set of three cannot exceed an AUROC of 0.701 pooled, as Appendix D derives with every feature set.

Beyond the checked features. On the features the detectors check, both adaptive attacks produced an outcome we call disguised corruption, the evasion that adaptive attacks on detectors aim for (Carlini & Wagner, 2017). The attacked chains keep the corruption's harm to task success, yet on every feature a detector reads they look at least as clean as clean chains. Read alone, no checked feature flagged the attack on any suite (Appendix C). Any feature read from a channel the attacker writes is one the attacker can control. We therefore argue that detection over features read from the reasoning chain fails as a class whenever an attacker can achieve disguised corruption against the features it reads. A learned monitor of the chain, such as the reasoning-safety monitor of Wang et al. (2026), falls under this argument but not under the proposition, and is untested here. The argument instead points to a detector that reads a channel the attacker does not write.

4.3 A PRE-REGISTERED ACTION-ONLY TEST

Such a detector is the subject of the test we pre-register here. The text of this subsection and of Appendix E is the registration. Its rules are fixed, no episode has run, and any later change will be reported as a dated deviation.

The detector, its attacker and the readings. The detector is an action-only task-consistency detector, which reads only the robot's executed actions, never the chain. It is fitted per task to clean episodes and flags an episode by its most unusual step. The attacker built against it is white-box, knowing the detector's reference and alarm threshold. Its attack decodes each candidate chain in advance and commits to the most corrupting one whose actions stay under that threshold (Appendix E). Table 3 fixes three readings of how the detector fares against that attack, each taken against a null of clean episodes from the same launch, with the attack's task success beside it. Each reading says what that outcome would mean for the claim, so no result can later be read into a different conclusion.

Table 3: The three readings of the pre-registered action-only test. No episode has run.

Outcome	What would be observed	What it would mean for the claim
Holds	The detector separates attacked from clean episodes above the null while the attack still does harm, or the attacker can hide only by giving up the harm.	A detector on a channel the attacker does not write survives an attacker built against it; we state which form held.
Fails	It does not separate them above the null while the attack keeps its harm.	Monitoring on the reasoning chain cannot secure it; integrity has to be enforced where the reasoning chain is transported.
Conditional	It holds only at a false-alarm rate no operator would accept.	A defense only under that condition, stated with the result.

5 Discussion

Deployment. Whatever the action-only test finds, a deployer who monitors the chain should not read a clean score as evidence of a clean chain when the attacker may know the monitor. Under the adaptive attack, the chains that scored as clean still carried the corruption's harm to task success. A clean score therefore shows only that the chain passed the features the monitor reads. An attacker who knows those features can write the chain to pass them.

Generality. Beyond the two policies, we argue that whenever an operator can correct a policy through its chain, for errors the policy cannot catch from its other inputs, an attacker can also corrupt it through that chain. From the chain alone, such a policy cannot tell the operator's edit from an attacker's. A defense therefore has to draw on something outside the chain, which is the channel the pre-registered action-only test reads. The argument covers every policy that meets its condition and says nothing about how many do. The two policies we measured show that the condition can be met.

Limitations. The construct controls pre-registered in Appendix F bound a limit, that the corruption we measure is a synthetic entity swap. The class argument, in turn, reaches as far as its condition. Its open test is a learned monitor of the chain under an attacker built against it. The reasoning-safety monitor of Wang et al. (2026) is one such monitor, evaluated so far only against attackers without access to it. The measurements themselves come from two policies in simulation. The policies were chosen because they write their chains at different times, so a result found on both replicates across that difference. The effect sizes are theirs, while the question and the design that answers it can be put to any policy that writes its chain in text. Simulation was chosen because only a simulator can start the compared runs from identical initial states. Even there, rollouts vary from run to run under GPU rendering, so we report that drift as a measured parameter (Appendix A).

6 Conclusion

On both policies, the reasoning chain changes what the robot does in both directions, so whoever writes it steers the robot. Beyond these two policies, we argue that whenever an operator can correct a policy through its chain, for errors the policy cannot catch from its other inputs, an attacker can also corrupt it through that chain. Two detectors that read only the chain catch its corruption, yet lose their discriminative power once the attacker is built against them, while the attack keeps its harm. Their failure does not depend on how the features they check are weighted, wherever the attacked chain matches or beats the clean chain on every feature in every pair, as on most of CoTinyVLA. On DeepThinkVLA a better weighting could help, but it cannot exceed a ceiling (§4.2). Whatever the weighting, both adaptive attacks produced disguised corruption, keeping the corruption's harm while looking at least as clean as clean chains on every feature a detector reads. We also argue that detection over features read from the reasoning chain fails as a class whenever an attacker can achieve disguised corruption against the features it reads. Whether a detector that reads a channel the attacker does not write can hold against such an attacker is the question our pre-registered test asks.

AI USE STATEMENT

We used generative AI tools for two of the tasks whose disclosure is required, designing experiments and implementing methods. For the first, the tools proposed experiments that could support or falsify our hypotheses. We chose which of those experiments to pursue. For the second, the tools wrote the patches to the evaluation code and the analysis scripts. We tested that code with unit tests. We did not use the tools to generate synthetic data sets or to help develop theoretical models or conceptual frameworks. Nor did we use them to formulate mathematical claims, to provide critical ingredients for proving them, or to assist in writing proofs. The tools also had no part in proposing or refining hypotheses, cleaning or reformatting a dataset, supporting qualitative or thematic data analysis, or interpreting results. Translation, the last task on that list, does not apply to this work. Among the tasks whose disclosure is recommended, we used the tools to suggest experimental parameters, to create and edit software code, to draft parts of the paper, and to edit it for readability. We also used them to search for information, to identify and summarize relevant literature, and to propose a title. We reviewed all AI-assisted work. We revised every drafted passage before it entered the paper, and we read each cited work in its source. The title, likewise, is our own wording, written after we weighed the tools' proposals. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies two released vision-language-action policies in simulation. It involves no human subjects and no personal data.

On both policies, whoever writes the reasoning chain steers the robot. Two detectors that read only the chain score no higher than chance on either policy against an adaptive attacker who knows what they check. In every case, the attack keeps the corruption's harm to task success. These findings could inform an attacker as well as a defender, so we weighed what the attack adds. The attack keeps the entity swap of the first measurement and appends a sentence that names the instruction's objects. It needs write access to the text between the step that writes the chain and the step that acts on it. This work provides no means of gaining that access. Its episodes all ran in simulation, with no physical robot involved.

We judge that the value of these findings to defenders outweighs the help they could give an attacker. The chain is already offered as an interface that an operator can rewrite, so the access an attacker would use exists independently of this work. What the work adds is a warning for deployers. A deployer who monitors the chain should not read a clean score as evidence of a clean chain when the attacker may know the monitor. For defenders, we also pre-register a test of a detector that reads the robot's executed actions and never the chain (§4.3). No episode of that test has run.

We will share the anonymized code with the reviewers and area chairs through a comment on the discussion forum once it opens. The attack code is therefore not public during review. After the decision, we will release the code publicly. The release will contain no policy weights. Users obtain the weights from each policy's own distribution, under its authors' terms.

REPRODUCIBILITY STATEMENT

The policies, the benchmark and the simulator we use are all publicly released (Section 3.1; Appendix A). Appendix A also gives each policy's inputs, the episode protocol, the seeds and the rule behind the entity swap, together with the run-to-run drift we measured. For the first measurement, Appendix B gives the rules of both rescue tests and their per-suite statistics. For the second, Appendix C states the exact rule of each feature, attack and verdict, and reports every detector result with its interval. The assumptions of the proposition and the derivation of its ceiling follow in Appendix D. The two tests that have not yet run, the action-only test and the construct controls, are registered in full in Appendices E and F. We will share the anonymized code with the reviewers and area chairs through a comment on the discussion forum once it opens, and we will release it publicly after the decision.

REFERENCES

Christopher Agia, Rohan Sinha, Jingyun Yang, Zi-ang Cao, Rika Antonova, Marco Pavone, and Jeannette Bohg. Unpacking Failure Modes of Generative Policies: Runtime Monitoring of Consistency and Progress, October 2024. URL http://arxiv.org/abs/2410.04640.

Iván Arcuschin, Jett Janiak, Robert Krzyzanowski, Senthooran Rajamanoharan, Neel Nanda, and Arthur Conmy. Chain-of-Thought Reasoning in the Wild is not Always Faithful. In *Workshop on Reasoning and Planning for Large Language Models*, March 2025. URL https://openreview.net/forum?id=L8094Whth0.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation, March 2025. URL http://arxiv.org/abs/2503.11926.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_0.5$: A Vision-Language-Action Model with Open-World Generalization. In *Proceedings of The 9th Conference on Robot Learning*, pp. 17–40. PMLR, October 2025a. URL https://proceedings.mlr.press/v305/black25a.html.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Shi, Laura Smith, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A Vision-Language-Action Flow Model for General Robot Control. In *Robotics: Science and Systems XXI*. Robotics: Science and Systems Foundation, June 2025b. ISBN 979-8-9902848-1-4. doi: 10.15607/RSS.2025.XXI.010. URL https://www.roboticsproceedings.org/rss21/p010.pdf.

Nicholas Carlini and David Wagner. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pp. 3–14, New York, NY, USA, November 2017. Association for Computing Machinery. ISBN 978-1-4503-5202-4. doi: 10.1145/3128572.3140444. URL https://dl.acm.org/doi/10.1145/3128572.3140444.

Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On Evaluating Adversarial Robustness, February 2019. URL http://arxiv.org/abs/1902.06705.

Harsh Chaudhari, Ethan Rathbun, Hanna Foerster, Jamie Hayes, Matthew Jagielski, Milad Nasr, Ilia Shumailov, and Alina Oprea. Thought-Transfer: Indirect Targeted Poisoning Attacks on Chain-of-Thought Reasoning Models, January 2026. URL http://arxiv.org/abs/2601.19061.

Hao Chen, Jiaming Liu, Chenyang Gu, Zhuoyang Liu, Renrui Zhang, Xiaoqi Li, Xiao He, Yandong Guo, Chi-Wing Fu, Shanghang Zhang, and Pheng-Ann Heng. Fast-in-Slow: A Dual-System VLA Model Unifying Fast Manipulation within Slow Reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, October 2025. URL https://openreview.net/forum?id=4asFznbzJg.

Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 2206–2216. PMLR, November 2020. URL https://proceedings.mlr.press/v119/croce20b.html.

Zhekai Duan, Yuan Zhang, Shikai Geng, Gaowen Liu, Joschka Boedecker, and Chris Xiaoxuan Lu. Fast ECoT: Efficient Embodied Chain-of-Thought via Thoughts Reuse, September 2025. URL http://arxiv.org/abs/2506.07639.

Bradley Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, New York, May 1994. ISBN 978-0-429-24659-3. doi: 10.1201/9780429246593.

Scott Emmons, Erik Jenner, David K. Elson, Rif A. Saurous, Senthooran Rajamanoharan, Heng Chen, Irhum Shafkat, and Rohin Shah. When Chain of Thought is Necessary, Language Models Struggle to Evade Monitors, July 2025. URL http://arxiv.org/abs/2507.05246.

Senyu Fei, Siyin Wang, Junhao Shi, Zihao Dai, Jikun Cai, Pengfang Qian, Li Ji, Xinzhe He, Shiduo Zhang, Zhaoye Fei, Jinlan Fu, Jingjing Gong, and Xipeng Qiu. LIBERO-Plus: In-depth Robustness Analysis of Vision-Language-Action Models, December 2025. URL http://arxiv.org/abs/2510.13626.

Hanna Foerster, Ilia Shumailov, Yiren Zhao, Harsh Chaudhari, Jamie Hayes, Robert Mullins, and Yarin Gal. Reasoning Introduces New Poisoning Attacks Yet Makes Them More Complicated, September 2025. URL http://arxiv.org/abs/2509.05739.

Zhengxian Huang, Wenjun Zhu, Haoxuan Qiu, Xiaoyu Ji, and Wenyuan Xu. TRAP: Hijacking VLA CoT-Reasoning via Adversarial Patches. In *Forty-Third International Conference on Machine Learning*, June 2026. URL https://openreview.net/forum?id=zLUPTgV7BI.

Eliot Krzysztof Jones, Alexander Robey, Andy Zou, Zachary Ravichandran, George J. Pappas, Hamed Hassani, Matt Fredrikson, and J. Zico Kolter. Adversarial Attacks on Robotic Vision Language Action Models, June 2025. URL http://arxiv.org/abs/2506.03350.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P. Foster, Pannag R. Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An Open-Source Vision-Language-Action Model. In *Proceedings of The 8th Conference on Robot Learning*, pp. 2679–2713. PMLR, January 2025. URL https://proceedings.mlr.press/v270/kim25c.html.

Tomek Korbak, Mikita Balesni, Elizabeth Barnes, Yoshua Bengio, Joe Benton, Joseph Bloom, Mark Chen, Alan Cooney, Allan Dafoe, Anca Dragan, Scott Emmons, Owain Evans, David Farhi, Ryan Greenblatt, Dan Hendrycks, Marius Hobbhahn, Evan Hubinger, Geoffrey Irving, Erik Jenner, Daniel Kokotajlo, Victoria Krakovna, Shane Legg, David Lindner, David Luan, Aleksander Mądry, Julian Michael, Neel Nanda, Dave Orr, Jakub Pachocki, Ethan Perez, Mary Phuong, Fabien Roger, Joshua Saxe, Buck Shlegeris, Martín Soto, Eric Steinberger, Jasmine Wang, Wojciech Zaremba, Bowen Baker, Rohin Shah, and Vlad Mikulik. Chain of Thought Monitorability: A New and Fragile Opportunity for AI Safety, December 2025. URL http://arxiv.org/abs/2507.11473.

Martin Kuo, Jianyi Zhang, Aolin Ding, Qinsi Wang, Louis DiValentin, Yujia Bao, Wei Wei, Hai Li, and Yiran Chen. H-CoT: Hijacking the Chain-of-Thought Safety Reasoning Mechanism to Jailbreak Large Reasoning Models, Including OpenAI o1/o3, DeepSeek-R1, and Gemini 2.0 Flash Thinking, February 2025. URL http://arxiv.org/abs/2502.12893.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, Karina Nguyen, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Robin Larson, Sam McCandlish, Sandipan Kundu, Saurav Kadavath, Shannon Yang, Thomas Henighan, Timothy Maxwell, Timothy Telleen-Lawton, Tristan Hume, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. Measuring Faithfulness in Chain-of-Thought Reasoning, July 2023. URL http://arxiv.org/abs/2307.13702.

Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks, October 2018. URL http://arxiv.org/abs/1807.03888.

Minhyeok Lee, Chiyoung Kim, Chanhoe Gu, Seongrok Kim, Sanghyuk Roy Choi, Donghwan Hwang, Donghun Ryu, and Seokhyun Kim. CoTinyVLA: Chain-of-Thought Distillation for a Sub-Billion-Parameter Vision-Language-Action Model, July 2026. URL http://arxiv.org/abs/2607.25487.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, November 2023. URL https://openreview.net/forum?id=xzEtNSuDJk.

Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*, pp. 413–422, December 2008. doi: 10.1109/ICDM.2008.17. URL https://ieeexplore.ieee.org/document/4781136.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*, February 2018. URL https://openreview.net/forum?id=rJzIBfZAb.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. The Attacker Moves Second: Stronger Adaptive Attacks Bypass Defenses Against Llm Jailbreaks and Prompt Injections, October 2025. URL http://arxiv.org/abs/2510.09023.

NVIDIA, Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi "Jim" Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, Joel Jang, Zhenyu Jiang, Jan Kautz, Kaushil Kundalia, Lawrence Lao, Zhiqi Li, Zongyu Lin, Kevin Lin, Guilin Liu, Edith Llontop, Loic Magne, Ajay Mandlekar, Avnish Narayan, Soroush Nasiriany, Scott Reed, You Liang Tan, Guanzhi Wang, Zu Wang, Jing Wang, Qi Wang, Jiannan Xiang, Yuqi Xie, Yinzhen Xu, Zhenjia Xu, Seonghyeon Ye, Zhiding Yu, Ao Zhang, Hao Zhang, Yizhou Zhao, Ruijie Zheng, and Yuke Zhu. GR00T N1: An Open Foundation Model for Generalist Humanoid Robots, March 2025. URL http://arxiv.org/abs/2503.14734.

NVIDIA, Yan Wang, Wenjie Luo, Junjie Bai, Yulong Cao, Tong Che, Ke Chen, Yuxiao Chen, Jenna Diamond, Yifan Ding, Wenhao Ding, Liang Feng, Greg Heinrich, Jack Huang, Peter Karkus, Boyi Li, Pinyi Li, Tsung-Yi Lin, Dongran Liu, Ming-Yu Liu, Langechuan Liu, Zhijian Liu, Jason Lu, Yunxiang Mao, Pavlo Molchanov, Lindsey Pavao, Zhenghao Peng, Mike Ranzinger, Ed Schmerling, Shida Shen, Yunfei Shi, Sarah Tariq, Ran Tian, Tilman Wekel, Xinshuo Weng, Tianjun Xiao, Eric Yang, Xiaodong Yang, Yurong You, Xiaohui Zeng, Wenyuan Zhang, Boris Ivanovic, and Marco Pavone. Alpamayo-R1: Bridging Reasoning and Action Prediction for Generalizable Autonomous Driving in the Long Tail, January 2026. URL http://arxiv.org/abs/2511.00088.

Alexander Robey, Zachary Ravichandran, Vijay Kumar, Hamed Hassani, and George J. Pappas. Jailbreaking LLM-Controlled Robots. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11948–11956, May 2025. doi: 10.1109/ICRA55743.2025.11128119. URL https://ieeexplore.ieee.org/document/11128119.

Shikhar Shiromani and Leo Richter. A False Average: Pooled CoT-Monitor Accuracy Conceals a Reasoning-Dependent Fragility, September 2026. URL http://arxiv.org/abs/2608.00583.

Gemini Robotics Team, Abbas Abdolmaleki, Saminda Abeyruwan, Joshua Ainslie, Jean-Baptiste Alayrac, Montserrat Gonzalez Arenas, Ashwin Balakrishna, Nathan Batchelor, Alex Bewley, Jeff Bingham, Michael Bloesch, Konstantinos Bousmalis, Philemon Brakel, Anthony Brohan, Thomas Buschmann, Arunkumar Byravan, Serkan Cabi, Ken Caluwaerts, Federico Casarini, Christine Chan, Oscar Chang, London Chappellet-Volpini, Jose Enrique Chen, Xi Chen, Hao-Tien Lewis Chiang, Krzysztof Choromanski, Adrian Collister, David B. D'Ambrosio, Sudeep Dasari, Todor Davchev, Meet Kirankumar Dave, Coline Devin, Norman Di Palo, Tianli Ding, Carl Doersch, Adil Dostmohamed, Yilun Du, Debidatta Dwibedi, Sathish Thoppay Egambaram, Michael Elabd, Tom Erez, Xiaolin Fang, Claudio Fantacci, Cody Fong, Erik Frey, Chuyuan Fu,

Ruiqi Gao, Marissa Giustina, Keerthana Gopalakrishnan, Laura Graesser, Oliver Groth, Agrim Gupta, Roland Hafner, Steven Hansen, Leonard Hasenclever, Sam Haves, Nicolas Heess, Brandon Hernaez, Alex Hofer, Jasmine Hsu, Lu Huang, Sandy H. Huang, Atil Iscen, Mithun George Jacob, Deepali Jain, Sally Jesmonth, Abhishek Jindal, Ryan Julian, Dmitry Kalashnikov, M. Emre Karagozler, Stefani Karp, Matija Kecman, J. Chase Kew, Donnie Kim, Frank Kim, Junkyung Kim, Thomas Kipf, Sean Kirmani, Ksenia Konyushkova, Li Yang Ku, Yuheng Kuang, Thomas Lampe, Antoine Laurens, Tuan Anh Le, Isabel Leal, Alex X. Lee, Tsang-Wei Edward Lee, Guy Lever, Jacky Liang, Li-Heng Lin, Fangchen Liu, Shangbang Long, Caden Lu, Sharath Maddineni, Anirudha Majumdar, Kevis-Kokitsi Maninis, Andrew Marmon, Sergio Martinez, Assaf Hurwitz Michaely, Niko Milonopoulos, Joss Moore, Robert Moreno, Michael Neunert, Francesco Nori, Joy Ortiz, Kenneth Oslund, Carolina Parada, Emilio Parisotto, Amaris Paryag, Acorn Pooley, Thomas Power, Alessio Quaglino, Haroon Qureshi, Rajkumar Vasudeva Raju, Helen Ran, Dushyant Rao, Kanishka Rao, Isaac Reid, David Rendleman, Krista Reymann, Miguel Rivas, Francesco Romano, Yulia Rubanova, Peter Pastor Sampedro, Pannag R. Sanketi, Dhruv Shah, Mohit Sharma, Kathryn Shea, Mohit Shridhar, Charles Shu, Vikas Sindhwani, Sumeet Singh, Radu Soricut, Rachel Sterneck, Ian Storz, Razvan Surdulescu, Jie Tan, Jonathan Tompson, Saran Tunyasuvunakool, Jake Varley, Grace Vesom, Giulia Vezzani, Maria Bauza Villalonga, Oriol Vinyals, René Wagner, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Chengda Wu, Markus Wulfmeier, Fei Xia, Ted Xiao, Annie Xie, Jinyu Xie, Peng Xu, Sichun Xu, Ying Xu, Zhuo Xu, Jimmy Yan, Sherry Yang, Skye Yang, Yuxiang Yang, Hiu Hong Yu, Wenhao Yu, Wentao Yuan, Yuan Yuan, Jingwei Zhang, Tingnan Zhang, Zhiyuan Zhang, Allan Zhou, Guangyao Zhou, and Yuxiang Zhou. Gemini Robotics 1.5: Pushing the Frontier of Generalist Robots with Advanced Embodied Reasoning, Thinking, and Motion Transfer, November 2025. URL http://arxiv.org/abs/2510.03342.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, October 2012. doi: 10.1109/IROS.2012.6386109. URL https://ieeexplore.ieee.org/document/6386109.

Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On Adaptive Attacks to Adversarial Example Defenses. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1633–1645. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/hash/11f38f8ecd71867b42433548d1078e38-Abstract.html.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, pp. 74952–74965, Red Hook, NY, USA, December 2023. Curran Associates Inc.

Taowen Wang, Cheng Han, James Liang, Wenhao Yang, Dongfang Liu, Luna Xinyu Zhang, Qifan Wang, Jiebo Luo, and Ruixiang Tang. Exploring the Adversarial Vulnerabilities of Vision-Language-Action Models in Robotics. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6948–6958, October 2025. doi: 10.1109/ICCV51701.2025.00653. URL https://ieeexplore.ieee.org/document/11444341.

Xunguang Wang, Yuguang Zhou, Qingyue Wang, Zongjie Li, Ruixuan Huang, Zhenlan Ji, Pingchuan Ma, and Shuai Wang. Beyond Content Safety: Real-Time Monitoring for Reasoning Vulnerabilities in Large Language Models, May 2026. URL http://arxiv.org/abs/2603.25412.

Zefeng Wang, Zhen Han, Shuo Chen, Fan Xue, Zifeng Ding, Xun Xiao, Volker Tresp, Philip Torr, and Jindong Gu. Stop Reasoning! When Multimodal LLM with Chain-of-Thought Reasoning Meets Adversarial Image, September 2024. URL http://arxiv.org/abs/2402.14899.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, October 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN 0099-4987. doi: 10.2307/3001968. URL https://www.jstor.org/stable/3001968.

Rongwu Xu, Zehan Qi, and Wei Xu. Preemptive Answer "Attacks" on Chain-of-Thought Reasoning, May 2024. URL http://arxiv.org/abs/2405.20902.

Zihao Xue, Zhen Bi, Long Ma, Zhenlin Hu, Yan Wang, Xueshu Chen, Zhenfang Liu, Kang Zhao, Jie Xiao, and Jungang Lou. Thought Purity: A Defense Framework For Chain-of-Thought Attack, February 2026. URL http://arxiv.org/abs/2507.12314.

Cheng Yin, Yankai Lin, Wang Xu, Sikyuen Tam, Xiangrui Zeng, Zhiyuan Liu, and Zhouping Yin. DeepThinkVLA: Enhancing Reasoning Capability of Vision-Language-Action Models. In *Third Conference on Language Modeling*, August 2026. URL https://openreview.net/forum?id=SZ9yppydP1&referrer=%5Bthe%20profile%20of%20Zhiyuan%20Liu%5D(%2Fprofile%3Fid%3D~Zhiyuan_Liu1).

Michał Zawalski, William Chen, Karl Pertsch, Oier Mees, Chelsea Finn, and Sergey Levine. Robotic Control via Embodied Chain-of-Thought Reasoning. In *Proceedings of The 8th Conference on Robot Learning*, pp. 3157–3181. PMLR, January 2025. URL https://proceedings.mlr.press/v270/zawalski25a.html.

Hangtao Zhang, Chenyu Zhu, Xianlong Wang, Ziqi Zhou, Changgan Yin, Minghui Li, Lulu Xue, Yichen Wang, Shengshan Hu, Aishan Liu, Peijin Guo, and Leo Yu Zhang. BadRobot: Jailbreaking Embodied LLM Agents in the Physical World. In *The Thirteenth International Conference on Learning Representations*, October 2024. URL https://openreview.net/forum?id=ei3qCntB66.

Gejian Zhao, Hanzhou Wu, Xinpeng Zhang, and Athanasios V. Vasilakos. ShadowCoT: Cognitive Hijacking for Stealthy Reasoning Backdoors in LLMs, April 2025a. URL http://arxiv.org/abs/2504.05605.

Jianli Zhao, Tingchen Fu, Rylan Schaeffer, Mrinank Sharma, and Fazl Barez. Chain-of-Thought Hijacking, May 2026. URL http://arxiv.org/abs/2510.26418.

Qingqing Zhao, Yao Lu, Moo Jin Kim, Zipeng Fu, Zhuoyang Zhang, Yecheng Wu, Zhaoshuo Li, Qianli Ma, Song Han, Chelsea Finn, Ankur Handa, Tsung-Yi Lin, Gordon Wetzstein, Ming-Yu Liu, and Donglai Xiang. CoT-VLA: Visual Chain-of-Thought Reasoning for Vision-Language-Action Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1702–1713, 2025b. URL https://openaccess.thecvf.com/content/CVPR2025/html/Zhao_CoT-VLA_Visual_Chain-of-Thought_Reasoning_for_Vision-Language-Action_Models_CVPR_2025_paper.html.

Linqing Zhong, Yi Liu, Yifei Wei, Ziyu Xiong, Maoqing Yao, Si Liu, and Guanghui Ren. ACoT-VLA: Action Chain-of-Thought for Vision-Language-Action Models, March 2026. URL http://arxiv.org/abs/2601.11404.

Xueyang Zhou, Guiyao Tie, Guowen Zhang, Hechang Wang, Pan Zhou, and Lichao Sun. BadVLA: Towards Backdoor Attacks on Vision-Language-Action Models via Objective-Decoupled Optimization, May 2025. URL http://arxiv.org/abs/2505.16640.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. Robosuite: A Modular Simulation Framework and Benchmark for Robot Learning, January 2025a. URL http://arxiv.org/abs/2009.12293.

Zihao Zhu, Xinyu Wu, Gehan Hu, Siwei Lyu, Ke Xu, and Baoyuan Wu. AdvChain: Adversarial Chain-of-Thought Tuning for Robust Safety Alignment of Large Reasoning Models, September 2025b. URL http://arxiv.org/abs/2509.24269.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, Quan Vuong, Vincent Vanhoucke, Huong Tran, Radu Soricut, Anikait Singh, Jaspiar Singh, Pierre Sermanet, Pannag R. Sanketi, Grecia Salazar, Michael S. Ryoo, Krista Reymann, Kanishka Rao, Karl Pertsch, Igor Mordatch, Henryk Michalewski, Yao Lu, Sergey Levine, Lisa Lee, Tsang-Wei Edward Lee, Isabel Leal, Yuheng Kuang, Dmitry Kalashnikov, Ryan Julian, Nikhil J. Joshi, Alex Irpan, Brian Ichter, Jasmine Hsu, Alexander Herzog, Karol Hausman, Keerthana Gopalakrishnan, Chuyuan Fu, Pete Florence, Chelsea Finn, Kumar Avinava Dubey, Danny Driess, Tianli Ding, Krzysztof Marcin Choromanski, Xi Chen, Yevgen Chebotar, Justice Carbajal, Noah Brown, Anthony Brohan, Montserrat Gonzalez Arenas, and Kehang Han. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. In *Proceedings of The 7th Conference on Robot Learning*, pp. 2165–2183. PMLR, December 2023. URL https://proceedings.mlr.press/v229/zitkovich23a.html.

APPENDIX

A SUITES AND EPISODE PROTOCOL

Section 3.1 introduces the two policies and the four suites at the level the argument needs. This appendix adds each policy's inputs, the protocol every episode follows, and the name pool and seed rule of the entity swap. It also gives the schedule on which conditions B and D edit each policy's chain, and the run-to-run drift we measured.

DeepThinkVLA. DeepThinkVLA (Yin et al., 2026) is a 2.9B-parameter policy built on a pretrained vision-language model. At each query it receives the image from a fixed third-person camera, the image from a camera on the robot's wrist, and the instruction, with no proprioceptive state. It first writes its reasoning chain as text between think delimiters. Conditioned on the images, the instruction and that chain, it then decodes a chunk of 10 actions in one parallel pass, each action a change in end-effector pose and a gripper command. The policy is queried again once the chunk has executed, and it writes a new chain at every query. We run one released checkpoint on all four suites, without further training. Its authors trained it on LIBERO demonstrations annotated with reasoning chains and then with reinforcement learning on task success (Yin et al., 2026, Appendix A.2).

CoTinyVLA. CoTinyVLA (Lee et al., 2026) is a 0.9B-parameter policy built on a compact language-model backbone. At each step it receives a short history of third-person frames and of wrist frames, a proprioception vector for the end effector and gripper, and the instruction (Lee et al., 2026, Section 3.1). Once per episode, right after reading the instruction, it writes a Plan, an ordered list of subgoals in text. The Plan is the chain we edit on this policy. Before each action chunk it also writes a short Think with fixed slots for the current phase, the gripper state and the next action (Lee et al., 2026, Section 3.2). An action head then emits a chunk of 8 actions of the same form as DeepThinkVLA's. Its authors fine-tuned it on the union of the four LIBERO-Plus suites (Fei et al., 2025), so the four standard suites are familiar to it. We run the released weights through the released evaluator, with the renderer, attention implementation, gripper convention and reasoning-length cap that evaluator sets. We patched that evaluator so that it draws initial states, caps episodes and loads the benchmark files as DeepThinkVLA's harness does. It also records both its own success signal and the simulator's success check. The two never disagreed. A last patch lets it accept a Plan written in advance, which conditions B and D need. The two policies therefore run with different attention implementations, each its authors' own. No quantity in this paper pools the two.

Simulator and episodes. Both policies run on the standard LIBERO benchmark (Liu et al., 2023), simulated in robosuite (Zhu et al., 2025a) over the MuJoCo physics engine (Todorov et al., 2012), from one checkout of its task files. Each suite has 10 tasks. On DeepThinkVLA every condition runs 3 run seeds (42, 123, 456) with 20 episodes per task and seed. On CoTinyVLA it runs 3 run seeds (42, 123, 456) with 20 episodes per task and seed. An episode's seed is its run seed plus its episode index, and it draws the episode's initial state from the task's stored set. Conditions A to D therefore start the same episode from the same state. On CoTinyVLA, where every episode records its initial-state index, the analysis checks this on every episode before any test runs. Decoding is greedy. An episode ends at success or at a cap of 220, 280, 300 or 520 steps on Spatial, Object,

Table 4: Ground-truth structure of the four LIBERO suites, parsed from their `.bddl` task specifications; the per-task parse appears in Appendix A. Only in the Goal suite does the text determine what to do rather than which of several similar objects to act on.

suite	scenes	tasks	goal structure	language must supply
LIBERO-Goal	1	10	10 distinct goals, 4 predicate types	the goal itself
LIBERO-Spatial	1	10	one goal: `On`(bowl, plate)	which of two identical bowls
LIBERO-Object	10	10	one schema: `In`(x, basket)	which object x
LIBERO-Long	9	10	10 distinct goals, mostly compound	little: the scene fixes the task

Table 5: The 10 LIBERO-Goal tasks, parsed from their `.bddl` specifications: one shared scene, 10 distinct goals over four predicate types (`Open`, `In`, `On`, `Turnon`). Object and region identifiers are shortened for readability; the specification files carry the full names.

instruction	goal predicate
open the middle drawer of the cabinet	`Open`(cabinet middle drawer)
open the top drawer and put the bowl inside	`In`(bowl, cabinet top drawer)
push the plate to the front of the stove	`On`(plate, region in front of stove)
put the bowl on the plate	`On`(bowl, plate)
put the bowl on the stove	`On`(bowl, stove cook region)
put the bowl on top of the cabinet	`On`(bowl, cabinet top)
put the cream cheese in the bowl	`On`(cream cheese, bowl)
put the wine bottle on the rack	`On`(wine bottle, rack top)
put the wine bottle on top of the cabinet	`On`(wine bottle, cabinet top)
turn on the stove	`Turnon`(stove)

Goal and Long, the same caps on both policies. Success is the task's goal predicate evaluated on the simulator state. Tables 5 and 6 print those predicates for Goal and Long. The camera images are rendered on the GPU.

What each suite asks of language. Table 4 summarizes the four suites from their task specification files, and Tables 5 and 6 give the per-task parse for Goal and Long. Goal places all of its tasks in one scene, each with a distinct goal. The image is therefore compatible with every goal in the suite, and language alone determines which one the robot should pursue. Spatial and Object fix the action and use language to pick out an object, while on Long the scene largely fixes the task. This structure is why §3.1 reads Goal as the suite where the two predictions about condition D differ most sharply.

The entity swap and its name pool. The entity swap of §3.1 matches a text against a fixed pool of the object names that appear in the four suites' instructions. It tries the longest name first, so that a compound name such as *cream cheese box* is matched whole. It maps each distinct object the text names to another name drawn from the pool, replaces every occurrence, and leaves the rest of the text as it was. A seed fixes the draw. The same rule acts on the instruction and on the chain.

Which seed draws each swap. The instruction swap of conditions C and D is drawn with the run seed, restarted for every instruction. Within one run seed, a whole suite therefore draws its replacements from a few names. Whether a replacement names an object the scene contains is then set by the seed and not by the task. Appendix B reports the correction on CoTinyVLA per seed and split by that property. The chain swap of condition B is drawn with the episode seed, so its replacement varies from episode to episode.

When B and D edit the chain. The two policies write their chains on different schedules, so each condition follows its policy's own. The four conditions are A, intact instruction and intact chain; B, intact instruction and corrupted chain; C, corrupted instruction and the chain the policy writes from it; and D, corrupted instruction and clean chain. On DeepThinkVLA a new chain is written at every query. Under B we swap each chain as soon as the policy writes it, before the actions are decoded. Under D, at every query, we first obtain the chain the policy writes from the clean instruction and the current image, and then decode the actions from that chain together with the corrupted instruction.

Table 6: The 10 LIBERO-Long tasks, parsed from their `.bddl` specifications: nine scenes, mostly compound goals, four predicate types. Only LIVING ROOM 2 hosts two tasks.

scene	instruction	goal predicates
kitchen 3	turn on the stove and put the moka pot on it	`Turnon`(stove) ∧ `On`(moka pot, stove)
kitchen 4	put the black bowl in the bottom drawer of the cabinet and close it	`Close`(cabinet bottom drawer) ∧ `In`(bowl, cabinet bottom drawer)
kitchen 6	put the yellow and white mug in the microwave and close it	`In`(mug, microwave) ∧ `Close`(microwave)
kitchen 8	put both moka pots on the stove	`On`(moka pot 1, stove) ∧ `On`(moka pot 2, stove) ∧ `Turnon`(stove)
living room 1	put both the alphabet soup and the cream cheese box in the basket	`In`(alphabet soup, basket) ∧ `In`(cream cheese, basket)
living room 2	put both the alphabet soup and the tomato sauce in the basket	`In`(alphabet soup, basket) ∧ `In`(tomato sauce, basket)
living room 2	put both the cream cheese box and the butter in the basket	`In`(cream cheese, basket) ∧ `In`(butter, basket)
living room 5	put the white mug on the left plate and put the yellow and white mug on the right plate	`On`(porcelain mug, plate 1) ∧ `On`(yellow-white mug, plate 2)
living room 6	put the white mug on the plate and put the chocolate pudding to the right of the plate	`On`(porcelain mug, plate) ∧ `On`(pudding, region right of plate)
study 1	pick up the book and place it in the back compartment of the caddy	`In`(book, caddy back compartment)

Table 7: Run-to-run drift: the episodes whose outcome differed between two clean runs of the same configuration at one run seed, per suite, out of the episodes in the pair. Both runs of every pair executed the same code for a clean episode.

suite	DeepThinkVLA	CoTinyVLA
Spatial	4 of 200	12 of 200
Object	0 of 200	0 of 200
Goal	9 of 200	4 of 200
Long	14 of 200	4 of 200

C and D receive byte-identical corrupted instructions. The D runner checks this against C's record for every task before the policy loads. On CoTinyVLA the Plan is written once per episode. Under B and D we write it in advance, before the first step, from the Plan the policy wrote under A for the same task, seed and episode. B swaps that Plan before the episode starts, so every chunk reads the swapped text. D injects it unchanged. The Think before each chunk is still written by the policy. Under D it is written from a context that holds the corrupted instruction. Appendix B reports how often it names the swapped object.

Run-to-run drift. Rendering on the GPU is what varies between two runs of the same configuration. Across repeated runs the physics state is identical at every step, while a few pixels of a rendered frame can differ by the smallest intensity step. A policy that decodes greedily can then write a different token and take a different action, so the same episode can end differently. Rather than assume this variation away, we measured it. For each policy and suite we ran the clean condition twice at one run seed and counted the episodes whose outcome differed (Table 7). On DeepThinkVLA both runs of each pair come from the second measurement, because the first measurement's clean cells store per-task rates and not per-episode outcomes. In every pair, both runs executed the same code for a clean episode. On CoTinyVLA the Plan text is also identical across the two runs on every episode (200 of 200), so the variation there enters after the Plan is written. The largest change in one task's success between the two runs is 15.0 points on DeepThinkVLA. On CoTinyVLA it is 10.0 points. The main text therefore pairs every comparison by task within one set of runs, and Appendices B and C give each effect with its interval.

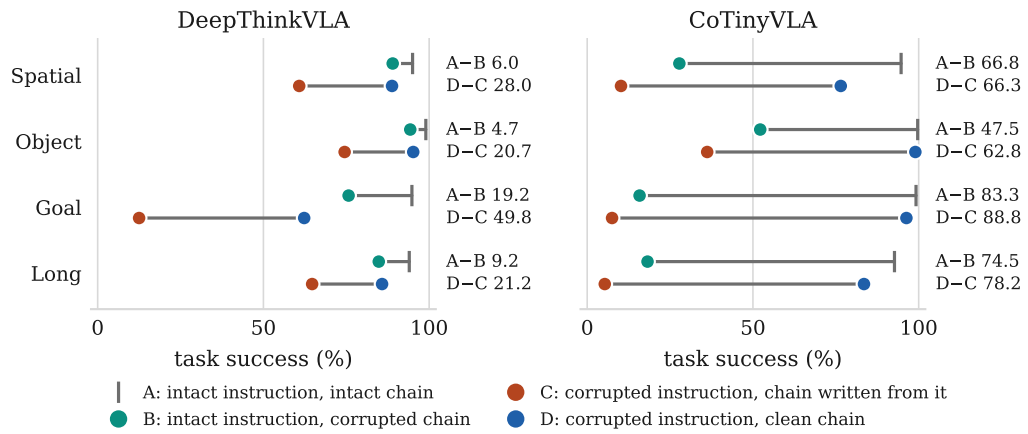


Figure 4: Corruption and correction per suite on DeepThinkVLA (left) and CoTinyVLA (right). Each row shows task success under A and B, and under C and D, with both differences printed beside it; the levels carry no interval in their source tables. The two policies are replications and are not ranked against each other.

An earlier renderer. A related earlier study of one of these policies rendered its camera images on the CPU, where repeated runs gave identical per-task results. Every figure in this paper is measured under GPU rendering. No figure from that study appears here, and none is compared with ours.

B The First Measurement in Detail

The main text reports the first measurement, corruption and correction on both policies, at the level the argument needs. This appendix adds the rules of each policy's rescue test and the per-suite statistics behind them. It also gives how much of the cost the clean chain leaves unrecovered on DeepThinkVLA, the descriptive comparisons the designs fixed before the runs, and the per-suite figure (Figure 4). Throughout, A is intact instruction and intact chain, B intact instruction and corrupted chain, C corrupted instruction and the chain the policy writes from it, and D corrupted instruction and clean chain. Corruption is A minus B, the cost of the corrupted instruction is A minus C, and correction is D minus C, each in points of task success.

What the four suites measure. Goal is where the two predictions about D differ most sharply, because language alone determines the goal there (Appendix A). On Spatial and Object language only picks out an object, and on Long the scene largely fixes the task. A corrupted instruction therefore has less to take away on those suites. On DeepThinkVLA its cost falls from 82.3 points on Goal to between 24.5 and 34.2 points on the other three. Those three suites ask whether correction holds where language selects an object instead of deciding the goal.

The rescue test on DeepThinkVLA. The unit of analysis is the task. Each task's success is averaged over its 3 run seeds and 20 episodes per seed, and the 10 tasks of a suite enter as paired observations. The test is a one-sided paired Wilcoxon signed-rank test on the per-task difference D minus C (Wilcoxon, 1945). Tasks with a zero difference are dropped, and the null distribution is enumerated exactly at the number of tasks that remain. The confirmatory test is Goal alone, at level 0.05. The causal claim for this policy rests on that test. Before condition D existed, the design also fixed an order for the remaining suites, Spatial, then Long, then Object. Each is tested at the full level, and the sequence stops at the first suite that does not reject. A suite after the stop would have been reported descriptively and never as a failed test. The sequence did not stop. All 4 suites rejected (Table 8). On Goal, Spatial and Long the statistic reaches 55, the largest value possible at 10 tasks, so D exceeded C on every task there. On Object the correction of 20.7 points is a mean over all 10 tasks. The signed-rank test ran on the 7 tasks with a nonzero difference, because D equals C exactly on the others. Every interval comes from a paired bootstrap that resamples tasks, at confidence level 95%, with 10000 draws and a fixed seed (Efron & Tibshirani, 1994).

Table 8: The first measurement on DeepThinkVLA, per suite, in points of task success. Columns give the cost of the corrupted instruction (A minus C); correction (D minus C) with its interval, paired effect size d, and the one-sided signed-rank statistic W^+ over the n tasks with a nonzero difference, with its exact p; and the residual (A minus D) with its one-sided upper bound. The suites were tested in the fixed order Goal, Spatial, Long, Object.

suite	A−C	D−C [interval]	d	W^+ (n)	p	A−D	upper bound
Spatial	34.2	28.0 [14.0, 43.3]	1.11	55 (10)	0.0010	6.2	10.8
Object	24.5	20.7 [8.8, 33.3]	0.98	28 (7)	0.0078	3.8	6.5
Goal	82.3	49.8 [35.2, 63.8]	2.04	55 (10)	0.0010	32.5	44.3
Long	29.3	21.2 [9.8, 35.2]	0.97	55 (10)	0.0010	8.2	14.3

How much the clean chain leaves unrecovered. Beside the test, the design fixed a bound: per suite, the residual A minus D with its one-sided upper confidence bound. The bound reads as a ceiling. With a clean chain, a corrupted instruction costs DeepThinkVLA at most 10.8 points on Spatial, 6.5 on Object, 44.3 on Goal and 14.3 on Long. On Goal the residual of 32.5 points is larger than the cost of corrupting the chain under an intact instruction, 19.2 points [10.3, 27.8]. The instruction therefore keeps an effect on the actions that the clean chain does not undo, which is the sense in which the main text calls the correction partial.

D minus B on DeepThinkVLA. The design also compared D with B on Goal. Before any run, a power analysis showed that this comparison could not be tested reliably over 10 tasks, so the design demoted it to a descriptive result with no test. On Goal, D minus B is -13.3 points [-25.2, -0.8], with a paired effect size d of -0.64 over 10 tasks. No conclusion is drawn from its sign.

The rescue test on CoTinyVLA. CoTinyVLA has its own test, fixed in its own design before its runs. The two policies' tests are never pooled. The test is a one-sided paired Wilcoxon signed-rank test on D minus C over all 40 tasks, with zeros dropped and the exact critical value taken at the observed ranks. Its answer is presence. Because it ranks tasks by the size of their difference, it leans on the suites with the largest cost, so the design reads generality from per-suite shares and sign counts instead. The statistic reached 820.0, the largest value attainable (820), against a critical value of 533.0, with an exact one-sided p of 9.095e-13. The design measures correction as a share of the cost, the sum over tasks of D minus C divided by the sum of A minus C. Before any run it fixed the effect of interest at a share of 0.25 and the band for the test at [0.25, 1.00]. The pooled share is 0.906 [0.851, 0.952]. Every per-suite point estimate and interval end also lies inside the band (Table 9). D minus C is positive on every task of every suite. Object's interval reaches 1.000 at its upper end, the edge of what a share can express, so we print the interval and do not call the correction complete there. On Goal alone, D minus C is 88.8 points [81.0, 95.8] with a paired effect size d of 6.94, reported without a test.

Rows of the CoTinyVLA design that did not apply. Three rows written before the run did not apply. The pooled cost of the corrupted instruction, 81.8 points, is far above the floor of 10 points below which the test would have been uninformative. Absence of correction, which the design would claim only if the share's upper bound fell below 0.10, was not in question. An audit would have been due if D had exceeded A on a suite, a sign that the injected Plan carried help it should not carry. D minus A is negative on every suite, so no audit was due.

What the correction on CoTinyVLA does not depend on. The swap lands. Under C, the Plan names a replacement object on a share 0.988 of the 2400 episodes. Whether the replacements name objects in the scene is set by the seed (Appendix A), so the design split the share by that property. Where every, some or none of the replacements name an object in the scene, the share is 0.982, 0.913 and 0.893, over 7, 48 and 65 task-and-seed cells. For the run seeds 42, 123, 456 in turn, the pooled share is 0.921, 0.892 and 0.903. Both splits are descriptive, and no test is attached to either. D minus B is 60.3 points pooled and positive on every suite, also descriptive. The two policies write their chains on different schedules, so we do not set this figure against DeepThinkVLA's.

Table 9: The first measurement on CoTinyVLA, per suite, in points of task success except the share. Columns give the cost of the corrupted instruction (A minus C); correction as a share of the cost, with its interval; the tasks on which D exceeds C; D minus B; and D minus A with its interval. The pooled row covers all four suites.

suite	A−C	share [interval]	D > C	D−B	D−A [interval]
Spatial	84.5	0.785 [0.623, 0.916]	10 of 10	48.7	−18.2 [−31.8, −7.2]
Object	63.5	0.990 [0.978, 1.000]	10 of 10	46.8	−0.7 [−1.7, 0.0]
Goal	91.7	0.969 [0.937, 0.991]	10 of 10	80.5	−2.8 [−5.8, −0.8]
Long	87.3	0.895 [0.814, 0.968]	10 of 10	65.3	−9.2 [−16.7, −2.8]
pooled	81.8 [74.5, 88.4]	0.906 [0.851, 0.952]		60.3 [52.8, 67.8]	−7.7 [−12.3, −3.9]

Spatial on CoTinyVLA. Spatial has the lowest share of the four suites. Two of its tasks carry that reading, and they differ in kind. On one, the clean Plan recovers little of the cost. Nothing about that task's measurement is unstable. The other has the lowest clean success of any task in the run. It also changes outcome on more episodes between two identical clean runs than any other Spatial task, so its shortfall reflects an unstable task rather than a separate failure of correction. Under D, the policy still writes each Think from a context that holds the corrupted instruction. Pooled over suites, the Thinks written under D name the swapped object at a rate of 0.136. The rate is highest on Spatial (0.319). This is consistent with Spatial's lower share. It is logged beside the result and is not a test of it.

C The Detectors and Attacks in Detail

Section 3.2 defines the two detectors, the two attacks and the verdict words at the level the argument needs. This appendix gives the exact rule behind each of them and the text each detector reads on each policy. It also reports every detector result with its interval, the attack's task success beside it, and how often each feature had something to check. Every rule below was fixed before any episode ran, except where a paragraph says otherwise.

What a detector scores. Both detectors score a reasoning chain against the action the policy executes after writing it. That action is the mean of the chunk executed from the query, taken over every dimension of the action. A chunk holds 10 actions on DeepThinkVLA and 8 on CoTinyVLA. A chunk cut short by the end of an episode is averaged over the actions that ran. Each feature returns a score in $[0, 1]$, where 1 reads as clean. The analysis takes 1 minus a score as its anomaly. On DeepThinkVLA the policy writes a new chain at every query, so every query of an episode is scored. The episode's score is its largest per-query anomaly, the quantity a monitor running during the episode would act on. The mean anomaly over the episode's queries is reported beside it as a descriptive column. On CoTinyVLA the detectors read the Plan once per episode, against the first chunk executed after it. The largest and the mean anomaly then coincide. The Thinks that CoTinyVLA writes before each chunk are its own text and not the channel the attacker writes, so no detector scores them.

The entity check. The entity check finds the objects the instruction names, with the same extractor and name pool as the entity swap (Appendix A). Its score is the share of those objects that the chain also names. If the instruction names no object, the score is 1. The chain is then recorded as having nothing to check.

The fixed four-feature consistency detector. The fixed four-feature consistency detector averages four features with equal weights. Nothing in it is fitted to data. Its first feature is the entity check. The second, direction, reads six direction words, each mapped to a translation axis and a sign in the simulator's frame. The six words form three pairs, left and right, up and down, and forward and backward, one pair for each of the robot's lateral, vertical and forward axes. A direction word counts only inside a motion phrase. That is, the nearest word before it in its sentence, skipping *the* and *a*, must come from a fixed list of motion words. The list includes *move*, *shift*, *toward* and *to*. Under this rule, *pick up*, *lift up*, *left-front* and *the right stove* contain no direction. For each counted

word whose axis moves by more than 0.02 in the executed action, direction checks that the action's sign on that axis matches the word. Its score is the share of those checks that match, or 1 when there is nothing to check. The third feature, gripper, takes the chain's first gripper word as the chain's intent. The words *grasp*, *close* and *pick* state an intent to close. The words *release*, *open* and *place* state an intent to open. In the executed action, a positive gripper command closes the gripper, while a negative one opens it. Gripper scores 1 when the intent matches the command's sign and 0 when it contradicts it. A command within 0.1 of zero scores 0.8. A chain with no gripper word scores 1. The fourth feature, parseability, is the share of the chain's sentences that contain at least one element the detector can read. Such an element is an object name from the pool, a direction word in a motion phrase, a gripper word, or an action verb from a fixed list. An empty chain scores 0. All word lists were fixed in code before any episode ran. Together, the four features check what a chain acts on, where it moves, what the gripper does, and whether the chain can be read at all. They do not check the order of subgoals, the spatial relations between objects, or agreement across queries. The proposition in the main text is therefore scoped to exactly these four features.

Destination phrases on Long. Of these four features, direction depends most on how its word rule reads a phrase. The motion-phrase rule lists *to* as a motion word, so it reads a direction word after *to the* as a direction. On two Long tasks the chains name a plate by its side, as in *to the right plate*. There the rule checks a destination against the arm's lateral motion. We found this during the run. No other task shows it. We kept the rule as it ran, because the attacker and the detector share it and the attacker had already acted. Appendix D therefore reports Long twice, as it ran and with direction set to have nothing to check on those two tasks, the second as a labelled sensitivity. A tighter rule, counting a direction word after *to the* only when no object name follows it in the same phrase, would remove this case.

Where each attack writes. Both attacks rewrite the chain the policy wrote. The policy then decodes its actions from the rewritten chain, as under condition B, intact instruction and corrupted chain. The non-adaptive attack is condition B's entity swap, drawn with the episode seed (Appendix A). The adaptive attack applies the same swap with the same seed, then appends one sentence that names the instruction's objects. The adaptive attack with the direction flip adds one step before that sentence. It reads the sign of the clean action on each translation axis. Wherever a direction word in a motion phrase disagrees with that sign, on an axis that moves by more than 0.02, it replaces the word with its opposite. The attacker and the direction feature call the same function to decide what counts as a direction. On DeepThinkVLA both attacks act at every query. The clean action there is the mean of the chunk the policy decoded from its own chain at that query. The runner records that chunk before decoding again from the rewritten chain. On CoTinyVLA both attacks act once per episode, on the Plan the policy wrote under condition A, intact instruction and intact chain, for the same task, seed and episode. The clean action there is the mean of the first chunk executed in that condition A episode. The rewritten Plan enters before the first step, by the route condition B uses. The policy then writes its own Thinks from it.

The null and the intervals. Each verdict compares a detector's interval with the interval of a null measured on the same policy. On DeepThinkVLA the null comes from a second clean run at one run seed for each suite, launched with the other runs. The detector scores one clean run as clean and the other as attacked, over 20 episodes a side per task. On CoTinyVLA the same comparison, over 20 episodes a side per task, gave every episode the same Plan score in both runs on every suite. Each detector's null there is therefore a point, with an interval of zero width. For each task, the AUROC compares the attacked episodes of that task with its clean episodes, counting a tie as half. Each side holds 60 episodes per task on DeepThinkVLA and 60 on CoTinyVLA. A table's figure is the mean of its tasks' AUROCs, over the 10 tasks of a suite or the tasks of all four suites pooled. Its interval comes from a cluster bootstrap over tasks with a fixed seed. The level is 95% with 10000 draws on DeepThinkVLA, and 95% with 10000 draws on CoTinyVLA. The attack's task success is averaged over the same tasks, with the same bootstrap.

The detection verdicts. A detector's interval lies below the null when it lies wholly below the null's interval. It overlaps the null when the two share any point, and lies above the null when it lies wholly above. Under the non-adaptive attack, a detector is caught when the lower end of its interval is at or above 0.90. It is partly caught when that end lies above the null's interval but below 0.90,

Table 10: Task success in the detection runs, in points, with its interval, per suite and pooled. Each attacked row is read against the non-adaptive row and the clean row by the harm rule. On CoTinyVLA the run with the direction flip seeded the same text as the adaptive attack and is a replicate.

policy	condition	Spatial	Object	Goal	Long	pooled
DeepThinkVLA	clean	95.0 [87.7, 99.2]	99.0 [97.5, 100.0]	96.2 [93.5, 98.5]	92.7 [89.2, 96.0]	95.7 [93.3, 97.5]
	non-adaptive	88.3 [79.7, 95.3]	94.3 [90.7, 97.3]	77.0 [68.2, 85.7]	84.8 [75.8, 92.7]	86.1 [81.8, 90.2]
	adaptive	89.2 [80.8, 95.8]	94.8 [91.2, 97.8]	79.8 [71.7, 88.0]	85.3 [76.7, 92.8]	87.3 [83.4, 91.0]
	adaptive + flip	89.7 [81.2, 96.3]	94.8 [91.2, 97.8]	80.0 [71.8, 87.7]	85.3 [76.5, 92.8]	87.5 [83.5, 91.2]
CoTinyVLA	clean	94.7 [86.8, 99.3]	99.7 [99.2, 100.0]	99.2 [98.2, 100.0]	92.7 [86.5, 97.2]	96.5 [93.9, 98.6]
	non-adaptive	27.8 [14.0, 42.7]	52.2 [43.5, 60.8]	15.8 [6.0, 29.7]	18.2 [6.8, 30.5]	28.5 [21.1, 36.1]
	adaptive	46.2 [29.3, 61.8]	61.2 [51.0, 71.7]	24.3 [12.2, 37.8]	22.8 [9.0, 38.2]	38.6 [30.2, 47.2]
	adaptive + flip (replicate)	46.0 [29.7, 61.3]	61.2 [51.3, 71.7]	23.8 [11.8, 38.0]	23.3 [9.2, 39.2]	38.6 [30.0, 47.2]

and not caught when its interval overlaps the null. Under the adaptive attack, the entity check reads chance when its interval overlaps the null, at or below chance when the interval lies below it, and not evaded when it lies above. The fixed four-feature consistency detector reads inversion below the null, chance on overlap, and not evaded above. A second condition for inversion, that the interval not contain the value a random score would give, was checked on every cell and agrees with the first everywhere. Each verdict is read per suite and on the pooled table. The pooled verdict is the headline only when every suite agrees with it. Under the adaptive attack that holds for the fixed detector on DeepThinkVLA alone. Elsewhere the per-suite verdicts govern. Where the suites differ, they differ only in whether a detector reads chance or lies below the null.

The harm verdicts. Beside every detector figure we read the attack's harm from its task success (Table 10). Harm is retained when the attack's interval overlaps the non-adaptive attack's, lost when it overlaps the clean episodes', and reduced when it overlaps neither. Chance or inversion counts as evasion only when harm is retained or reduced, because on harm lost the detector has nothing to detect. The written rule did not say which reading holds when an interval overlaps both. That case arose on DeepThinkVLA on Object, Spatial and Long, under both adaptive forms. We settled it after the tables existed and before any verdict was recorded. Such a cell reads harm retained. Harm lost would assert that the attack undid the corruption. On each such cell, however, the attack's mean success lies nearer the non-adaptive attack's than the clean episodes'. On 2 of those suites, Spatial and Long, the clean and non-adaptive intervals overlap each other, so the harm reading separates little there. On CoTinyVLA every attacked interval overlaps the non-adaptive attack's and not the clean episodes', so every cell reads harm retained without the tie-break.

Every detector result. Table 11 gives each detector's AUROC with its interval, per suite and pooled, beside its null. Before the run, we simulated how wide an interval over 10 tasks would be for a range of assumed differences between tasks. That analysis found caught reachable on a suite only when the tasks' AUROCs barely differ from one another. It also named partly caught as a plausible per-suite verdict for the fixed detector under the non-adaptive attack. Both detectors were caught in every cell on both policies, with per-task AUROCs that barely differ from one another. The threshold was fixed with that caution on record.

Cells on the boundary. Under the adaptive attack, some intervals meet the null at a point. The entity check's interval ends exactly at the null in 1 cell on DeepThinkVLA, on Object, and in 2 cells on CoTinyVLA, on Goal and Long. The fixed detector's interval does the same on CoTinyVLA's Goal and Object. Because overlap includes a shared point, these cells read chance. An interval ending one printed step lower would read at or below chance, or inversion for the fixed detector. Either way, no interval in these cells lies above the null.

Table 11: AUROC of each detector per suite and pooled, with its interval, beside the null of two clean runs. On CoTinyVLA the two adaptive forms seeded identical text, so one adaptive row is shown.

detector	condition	Spatial	Object	Goal	Long	pooled
DeepThinkVLA						
entity check	null	0.476 [0.445, 0.510]	0.500 [0.500, 0.500]	0.500 [0.492, 0.508]	0.505 [0.500, 0.512]	0.495 [0.486, 0.504]
	non-adaptive	0.997 [0.992, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	1.000 [1.000, 1.000]	0.999 [0.998, 1.000]
	adaptive	0.346 [0.287, 0.394]	0.498 [0.495, 0.500]	0.497 [0.493, 0.500]	0.394 [0.290, 0.476]	0.434 [0.397, 0.465]
	adaptive + flip	0.346 [0.287, 0.394]	0.498 [0.495, 0.500]	0.497 [0.493, 0.500]	0.394 [0.290, 0.477]	0.434 [0.397, 0.465]
fixed four-feature	null	0.476 [0.450, 0.504]	0.500 [0.500, 0.500]	0.489 [0.440, 0.531]	0.502 [0.493, 0.509]	0.491 [0.478, 0.505]
	non-adaptive	0.998 [0.997, 1.000]	0.999 [0.999, 1.000]	0.997 [0.992, 1.000]	0.995 [0.989, 0.999]	0.997 [0.995, 0.999]
	adaptive	0.228 [0.177, 0.276]	0.218 [0.152, 0.286]	0.250 [0.164, 0.343]	0.217 [0.161, 0.273]	0.228 [0.195, 0.263]
	adaptive + flip	0.230 [0.178, 0.285]	0.192 [0.135, 0.250]	0.246 [0.155, 0.350]	0.161 [0.101, 0.220]	0.207 [0.172, 0.245]
CoTinyVLA						
entity check	null	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]
	non-adaptive	0.996 [0.992, 0.999]	1.000 [1.000, 1.000]	0.997 [0.994, 1.000]	1.000 [1.000, 1.000]	0.998 [0.997, 1.000]
	adaptive	0.150 [0.000, 0.300]	0.500 [0.500, 0.500]	0.400 [0.250, 0.500]	0.450 [0.350, 0.500]	0.375 [0.312, 0.438]
fixed four-feature	null	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]	0.500 [0.500, 0.500]
	non-adaptive	0.996 [0.992, 0.999]	1.000 [1.000, 1.000]	0.997 [0.994, 1.000]	1.000 [1.000, 1.000]	0.998 [0.997, 1.000]
	adaptive	0.150 [0.000, 0.300]	0.450 [0.350, 0.500]	0.400 [0.250, 0.500]	0.460 [0.350, 0.530]	0.365 [0.290, 0.430]

The episode score and the fixed detector's verdict. As a descriptive check on DeepThinkVLA, we also scored each episode by its mean anomaly instead of its largest. The fixed detector's pooled AUROC is then 0.354 [0.328, 0.379] under the adaptive attack and 0.339 [0.306, 0.370] with the direction flip. Both intervals lie wholly below the pooled null of 0.491 [0.478, 0.505], so the verdict holds under either episode score.

The direction flip. On DeepThinkVLA the flip changed a word on 4 of 7209 queries on Spatial, 26 of 8904 on Object, 11 of 9393 on Goal and 571 of 17600 on Long. Long is also the suite of the destination phrases, so the fixed detector's lower AUROC there under the flip is not read as the flip's effect. On the other suites the flip moves the fixed detector's AUROC little. Its verdict is the same with or without the flip on every suite. The fixed detector already reads inversion on the pooled table and on every suite under the adaptive attack without the flip. On CoTinyVLA the flip changed 0 of 600 Plans on each suite. The two adaptive forms therefore seeded identical text on every episode. We report the first form as the adaptive attack on this policy, and the run with the flip as a replicate of that text. For both detectors, the replicate's per-task AUROCs equal those of the first form on every task. Because the seeded text is the same, its task success (Table 10) differs from the first form's only through run-to-run drift (Appendix A).

CoTinyVLA's per-task AUROCs. On CoTinyVLA every episode of a task starts from the same instruction. The detectors read its Plan once per episode. The clean Plans of a task therefore score nearly alike, as do its attacked Plans. A task's AUROC then reads little more than whether its attacked Plans score the same as its clean ones, cleaner, or less clean. Every per-task AUROC on this policy takes one of three values, so its intervals are coarse. On this policy the fixed detector and the entity check also agree on every task but one of Object and one of Long. Gripper took the same value on the clean and attacked Plans of every task. Parseability did too, on every task but that Object task. Direction had something to check only on that Long task. The two policies are therefore different instruments for this measurement, one more reason we read them as replications and never compare their figures.

Table 12: Share of scored chains on which each feature had something to check, averaged over tasks, on the clean side (c) and under the adaptive attack with the direction flip (a). On CoTinyVLA each episode has one scored chain, its Plan.

		Spatial		Object		Goal		Long	
policy	feature	c	a	c	a	c	a	c	a
DeepThinkVLA	entity	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	direction	0.00	0.00	0.01	0.01	0.00	0.01	0.05	0.04
	gripper	0.75	0.74	0.83	0.81	0.64	0.65	0.87	0.86
	parseability	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CoTinyVLA	entity	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	direction	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.08
	gripper	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	parseability	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Which features had something to check. Table 12 gives, for each feature, the share of scored chains on which it had something to check, averaged over tasks. The attacked side is the adaptive attack with the direction flip, on which the proposition is computed. On DeepThinkVLA direction had something to check on almost no query. Most of its checks fall on the two Long tasks with destination phrases. Gripper had something to check on most queries. The entity check and parseability had something to check on nearly all. On CoTinyVLA gripper counts as having something to check on any Plan with a gripper word. That is how the scoring code defines it for this feature. These shares bear on where the proposition holds on DeepThinkVLA. On its Spatial table, consider the share of episode pairs in which the attacked episode matches or beats the clean one on both the entity check and direction. Its smallest value over tasks is 1.000. By the rule of Appendix D, the proposition is therefore stated on that table over those two features, with gripper and parseability dropped. Direction had something to check there on 0.00 of clean queries and 0.00 of attacked ones, so the statement rests on the entity check. On Long the same pair reaches 1.000 only in the sensitivity form, against 0.982 as it ran, so it is reported as a sensitivity and not stated.

Each feature alone. Beyond the entity check, we scored gripper and parseability alone on DeepThinkVLA, with the same episode score, bootstrap, seed and null as the two detectors. These figures are descriptive and carry no verdict word. Under the adaptive attack, gripper alone reads 0.503 [0.498, 0.510] pooled, against its null of 0.499 [0.494, 0.502]. On Object, Spatial and Long that score gives it an interval of zero width, so we also report its mean score. Scored by its mean anomaly, gripper alone reads 0.466 [0.440, 0.491] against a null of 0.497 [0.480, 0.515]. Under both scores its interval overlaps its null, so read alone it does not flag the attack. Parseability alone reads 0.158 [0.131, 0.185] against a null of 0.494 [0.478, 0.509]. It lies wholly below its null, so read alone it rates attacked chains as cleaner than clean ones. The entity check reads chance or at or below chance in every cell. Direction almost never had something to check. No feature read alone therefore flagged the adaptive attack on DeepThinkVLA. Appendix D asks the same question of every set of these features at once.

D The Proposition in Detail

The main text states the proposition where its condition holds and a ceiling where it does not. This appendix derives that ceiling from how the AUROC counts ties, and gives the share behind it for every set of the four checked features on both policies. It also records how the rule for stating the proposition reached its present form, in the order the changes happened.

The share and the ceiling. Fix a task and a set of the checked features. For each pair of an attacked episode and a clean episode of that task, we ask whether the attacked episode's value is at or above the clean episode's on every feature of the set. The share s is the fraction of pairs for which it is. Consider any detector that is monotone in those features, meaning its score for an episode never falls when one of the features rises. Every detector that adds the features with non-negative weights is of this kind. On a pair the attacked episode matches or beats, such a detector scores the attacked

episode at least as clean as the clean one. The AUROC then counts that pair as zero, or as half when the two scores tie. Every other pair counts at most one. The task's AUROC therefore cannot exceed $1 - s/2$, whatever the detector's weights. At $s = 1$ this bound equals chance, so no such detector can score above chance on that task. The proposition is stated over a set on a table only when every task of the table reaches $s = 1$. Where some task falls short, the table licenses only the ceiling $1 - s_{\min}/2$, where $s_{\min}$ is the smallest share over its tasks. That ceiling is written as a value the detector cannot exceed, with its figure, and never as chance. With no set of at least two features at $s = 1$ on every task, the proposition is withdrawn for that table and the ceiling is reported.

Which score the bound covers. On CoTinyVLA an episode has one scored Plan, so its value on a feature is that Plan's value. On DeepThinkVLA an episode has many queries. The design did not say how to form an episode's value from them. The analysis script took the mean over queries before any verdict was read. Under it, the fixed detector's episode anomaly is its mean anomaly over queries, because a weighted sum of means is the mean of the weighted sums. The bound therefore covers the mean-score AUROC of Appendix C, which sits beside the largest-anomaly figure the verdicts read. It does not cover the largest-anomaly score, because the worst query can differ from one feature to another.

How the rule reached its present form. The rule for stating the proposition changed twice after the run, and each change narrowed what the rule licenses. As written before the run, it stated the proposition over a set wherever the share reached at least one half on every task. It put the ceiling at $1 - s$. That bound counted a tied pair as a loss for the detector, while the AUROC counts a tie as half. The first tables showed the error. On the one CoTinyVLA task whose share fell below one, the fixed detector's AUROC exceeded $1 - s$ and equalled $1 - s/2$ exactly. We corrected the bound to $1 - s/2$ after the tables existed and before the proposition was stated anywhere. An independent re-derivation from the AUROC definition in the analysis code reached the same bound. A check over many non-negative weightings on every task of both policies then found it never exceeded, and attained exactly where those pairs all tie. Under the corrected bound, a share of one half no longer licenses a claim of chance. We therefore amended the threshold in place, after the verdicts were recorded, to the rule stated above. Under the threshold as first written, the set of entity, direction and parseability would have been stated on every DeepThinkVLA table, since its smallest pooled share is 0.598. The amended rule states it on none of them. Neither step moved a measured number, a verdict word or any other outcome row.

Where the proposition is stated. Table 13 gives the smallest per-task share for every set of the four features, on both policies, under the adaptive attack with the direction flip. A set of one feature is listed for completeness. The proposition needs at least two. On CoTinyVLA the share over all four features is 1.000 on every task of Goal, 1.000 on Object and 1.000 on Spatial. The proposition is stated on those three suites over all four features. On Long the smallest share is 0.800, on the one task where direction had something to check. In the sensitivity form fixed before any table existed, it is 1.000, and the proposition is stated on Long in that form only. As run, no monotone detector over the four features can exceed 0.600 on any Long task, or 0.600 on the pooled table. On DeepThinkVLA no set of three or four features reaches a share of one on every task of any table. The set of three that comes closest is entity, direction and parseability, which drops gripper. Its smallest pooled share is 0.598, so no detector monotone in it can exceed 0.701. Entity and direction together have a smallest pooled share of 0.982, which gives the pair a ceiling of 0.509. On Spatial that pair reaches a share of one on every task, so the proposition is stated there over the pair (Appendix C). Every set of three or four features includes gripper or parseability, the two features with the lowest pooled shares on DeepThinkVLA. Appendix C shows that neither of them, read alone, flags the adaptive attack.

Beyond the checked features. The proposition is scoped to exactly these four features. Beyond them the main text argues, under a condition stated there, and names a test that reads the executed actions instead of the chain. Appendix E pre-registers that test.

Table 13: Smallest share over tasks of the pairs in which the attacked episode matches or beats the clean one on every feature of the set, under the adaptive attack with the direction flip. Columns are Spatial, Object, Goal, Long, Long with direction set to have nothing to check on the two tasks with destination phrases (sens., Appendix C), and pooled. The features are entity (ent), direction (dir), gripper (grip) and parseability (parse).

	DeepThinkVLA						CoTinyVLA					
feature set	Spat.	Obj.	Goal	Long	sens.	pool.	Spat.	Obj.	Goal	Long	sens.	pool.
ent	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
dir	1.000	0.984	0.983	0.982	1.000	0.982	1.000	1.000	1.000	0.800	1.000	0.800
grip	0.491	0.445	0.484	0.352	0.352	0.352	1.000	1.000	1.000	1.000	1.000	1.000
parse	0.632	0.832	0.598	0.710	0.710	0.598	1.000	1.000	1.000	1.000	1.000	1.000
ent + dir	1.000	0.984	0.983	0.982	1.000	0.982	1.000	1.000	1.000	0.800	1.000	0.800
ent + grip	0.491	0.445	0.484	0.352	0.352	0.352	1.000	1.000	1.000	1.000	1.000	1.000
ent + parse	0.632	0.832	0.598	0.710	0.710	0.598	1.000	1.000	1.000	1.000	1.000	1.000
dir + grip	0.491	0.445	0.484	0.352	0.352	0.352	1.000	1.000	1.000	0.800	1.000	0.800
dir + parse	0.632	0.832	0.598	0.710	0.710	0.598	1.000	1.000	1.000	0.800	1.000	0.800
grip + parse	0.383	0.389	0.366	0.291	0.291	0.291	1.000	1.000	1.000	1.000	1.000	1.000
ent + dir + grip	0.491	0.445	0.484	0.352	0.352	0.352	1.000	1.000	1.000	0.800	1.000	0.800
ent + dir + parse	0.632	0.832	0.598	0.710	0.710	0.598	1.000	1.000	1.000	0.800	1.000	0.800
ent + grip + parse	0.383	0.389	0.366	0.291	0.291	0.291	1.000	1.000	1.000	1.000	1.000	1.000
dir + grip + parse	0.383	0.389	0.366	0.291	0.291	0.291	1.000	1.000	1.000	0.800	1.000	0.800
all four	0.383	0.389	0.366	0.291	0.291	0.291	1.000	1.000	1.000	0.800	1.000	0.800

E THE PRE-REGISTERED ACTION-ONLY TEST IN FULL

The main text names the action-only test's detector, the attacker built against it and the three readings of its outcome. This appendix gives the rest of the design, which the submitted text registers. No episode of the test's own launch has run. Any change to what follows will be reported as a dated deviation, with its reason and the rows it moves.

Scope. The test runs on the Goal suite, on both policies. On Goal every task shares one scene, as §3.1 notes, so a robot steered toward the wrong object still executes motion that belongs to some task of that scene. That is the case an action-only detector has to separate from clean behaviour. The seeds, episodes per task, initial-state rule and rendering are those of every other run in this paper (Appendix A).

The detector. The action-only detector reads only the executed actions of an episode. It is fitted per task to clean statistics. It looks at each step through two views. The phase view divides the episode into phases, one per chunk of executed actions, and scores a step by its Mahalanobis distance from the clean actions of the same task and phase. The path view adds up the executed translations from the first step, which gives where the hand has moved relative to its start. It scores that position by its Mahalanobis distance from the clean positions of the same task and phase. Each view has its own threshold, the $(1 - f)$ quantile of the largest distance over the clean reference episodes, where f is the target clean flag fraction. A step's combined score is the larger of its two distances, each divided by its own threshold, so a step is flagged when either view crosses. An episode's score is its most unusual step, the statistic a monitor running during the episode would act on. Each view's own episode score is reported beside the combined one, so the results can say which view did the work. A phase with too few reference steps falls back to the task's pooled reference. The regularizing constants of the two distances and that minimum are fixed from the clean episodes alone. They are recorded and never changed once an attacked episode is scored.

Fitting and scoring. The detector never scores an episode it was fitted on. For each task and each run seed in turn, it is fitted on the clean episodes of the other two seeds and scores every clean and attacked episode of the held-out seed. The run seeds never share an initial state. An attacked episode starts from the same initial state as the clean episode with the same seed and index. The two sides of every comparison are therefore paired by initial state. The task's AUROC is computed per held-out seed, counting a tie as half, and averaged over the held-out seeds. The reported figure is the mean

over tasks, with a cluster bootstrap over tasks, as in Appendix C. The analysis asserts that no episode appears both in a reference and in the set that reference scores.

The threshold. The confirmatory target is $f = 10\%$ per view. It follows from the fold sizes alone. At this target each threshold rests on several reference episodes. The held-out clean flag rate can also resolve it within a fold. Because an episode is flagged when either view crosses, the realised combined rate can exceed the target. The results therefore print the realised combined and per-view rates, never the target. The threshold does not enter the AUROC, which covers every threshold at once. It does fix the line the attacker must stay under, so the attacker's episodes, and with them the AUROC, depend on it. The attacker is therefore also built and run against the detector at 5% and 20% per view, as pre-registered sensitivities. The three readings are taken from the 10% run alone. If the reading is the same at the two other targets, the results state that the threshold did not decide it. If the reading differs, the outcome depends on the false-alarm budget. It is then reported as the third reading, with that budget as its condition.

The launch and the null. The reference, the null and every row of the attacker built against the detector are fitted and scored inside the test's own launch. That launch first runs a clean reference cell per run seed and one further clean cell at one run seed. The further cell serves twice. Scored against the clean cell of the same seed, it gives the within-launch null, which shows what nothing to detect looks like inside that launch. Scored against the clean cell of the same seed from the detection runs, it gives the drift cell, a descriptive measure of how far scores move between two launches of the same policy. The null pair is not asked to match, since rollouts differ from run to run (Appendix A). It is asked to start from the same initial states and to be scored against the same reference, with its flip count reported beside it. Only after the fold references and thresholds for all three targets are committed with a hash does any attacked episode run, because the attacker reads them.

The attacker built against it. The attacker is white-box. It reads the reference statistics and thresholds of the fold that scores its episode, and it decodes the chunk each candidate chain would produce before committing to one. It writes only the reasoning chain, never the instruction, the image, the executed action or the reference. Its goal is the non-adaptive attack's goal, to steer the robot to the swapped object, with every executed step kept under the thresholds. At each query it considers four candidate chains, ordered from most to least corrupting. The first is the chain the policy writes from the swapped instruction, as under condition C, corrupted instruction and the chain the policy writes from it. The second is the entity swap of the clean chain, the non-adaptive attack. The third is the adaptive attack with the direction flip applied to the clean chain. The fourth is the clean chain itself. The swap in the first three uses the same rule and seed as condition B, so the wrong object is the one condition B names for that episode. A candidate is admissible when, with its chunk appended to the episode so far, every step of the chunk scores at or under both thresholds. The attacker commits to the first admissible candidate. If none is admissible, it commits to the candidate whose largest combined score is lowest. That query counts toward the no-admissible rate. The committed chunk is executed as decoded. For this attacker the AUROC, the attack's task success and the no-admissible rate are always reported together. Beside them, a count per list position shows whether the attacker mostly ran a corrupting chain or fell back to the clean one. On CoTinyVLA the four candidate Plans are written once per episode, and the admissibility check and the choice of Plan are made per chunk, so the seeded Plan may change from chunk to chunk.

The rows it is read against. The attacker is compared with rows that were not adapted to the action-only detector. These are the non-adaptive attack and both forms of the adaptive attack, taken from the detection runs of Appendix C. Each is scored by the same rule against the test's reference, so each is a cross-launch row, read beside the drift cell. The non-adaptive attack is also read over its flipped pairs alone, the initial states where the clean episode succeeded and the attacked one failed, as a descriptive line. Coherent wrong-goal execution, condition C with its own chain, runs in the test's launch as a bound and never as an attack. If its interval overlaps the null, the executed actions can hide a wrong goal on this policy. If it lies above the null, executing the wrong goal leaves a mark in the actions. The attacker must then hide better than a coherent execution does.

Harm, adaptation cost and the operating point. Harm is read as in Appendix C, from the attack's task success with the same bootstrap over tasks. It is retained when the attacker's interval

overlaps the non-adaptive attack's, lost when it overlaps the clean episodes', and reduced when it overlaps neither. The clean interval is the test launch's own, with the detection runs' clean interval reported beside it as a sensitivity. The adaptation cost is the non-adapted row's AUROC minus the attacker's, paired over tasks, with a cluster bootstrap interval of that difference. The operating point is the attacked flag rate at the realised held-out clean flag rate, per view and combined. Both are reported beside every reading. No word stronger than *separates* is licensed by the AUROC alone. A working defense is claimed only with the operating point in the same sentence.

What the design can resolve. Before any run, a committed simulation fixed how large each reading must be for the 10 Goal tasks to detect it in 80% of repeats of the whole experiment. It reads no data from the test and assumes a between-task spread of per-task AUROCs from 0.05 to 0.20. Over that range, the attacker's mean AUROC must exceed chance by 0.170 to 0.340 for its interval to clear the null's. The adaptation cost must be at least 0.090 to 0.120, depending on the spread and on the non-adapted row's AUROC, for its 95% interval to lie wholly above zero. For harm, the attack's task success must fall 0.190 below the non-adaptive attack's mean before harm reduced becomes readable. Harm lost stops being readable once success sits 0.116 below the clean mean. The harm figures come from DeepThinkVLA's measured Goal rates in the detection runs. CoTinyVLA's are computed by the same script from its own rates before its launch. The simulation draws the null and the attacker row independently, while the real pair share a clean cell, so the first two figures are bounds. They are not predictions. An effect smaller than they are is reported as unresolved. No reading is taken from it.

Published baselines. The action-only detector is compared with published detectors that read the executed actions and nothing else. Each is reimplemented from its own paper and scored by the same fold rule, AUROC, bootstrap and null. The first is a Mahalanobis distance to clean statistics with no phase or path (Lee et al., 2018). The second is a nearest-neighbour distance to the clean reference actions, whose source is fixed before the launch by the same rule. The third is an Isolation Forest fitted on each fold's clean actions (Liu et al., 2008). The fourth is the temporal action-consistency score of a runtime monitor built for robot policies (Agia et al., 2024). That score compares sampled action chunks at overlapping steps. It is kept if the runner can produce those samples without changing what is executed, reported as our adaptation if it cannot, and otherwise dropped with the reason stated. One learned slot is filled before the launch by a rule fixed now. The detector must read executed actions only and come from a peer-reviewed paper that fixes both its architecture and its training objective. It is reimplemented with no choice of ours beyond the input size and the training set. If none qualifies, the slot is dropped and the results say so. Each baseline, the learned slot included, gets its own attacker built by the rule above with its own score and threshold, at the confirmatory target only. A baseline scored only on the attacker built against our detector would face an attacker built for another detector. Each is also scored on the non-adapted rows, so its adaptation cost is stated too. The two views of our detector and the bound of coherent wrong-goal execution are reported as ablations, never as references. The detectors of Appendix C appear as context only.

Both policies. CoTinyVLA runs the same design as DeepThinkVLA, stage for stage, with its own clean cells, constants and thresholds. Nothing is carried from one policy to the other. Each row reads as replicated on the second policy or not. The two policies' figures are never set against each other.

Checks before any attacked episode. The detector first reproduces, on hand-written episodes, distances computed on paper. The sidecar files of every cross-launch row are fetched and checked against their recorded hashes. The phase definition is checked against the runner, so that each phase is the chunk a query actually executed. On one Goal task per policy, the attacker's records are checked four ways. The first is the share of queries with an admissible corrupting candidate. The others check that the decoder read the committed candidate's chain, that the attacker's scores equal the analysis script's, and that every logged field is present. That share is judged against a threshold for near zero, written down before the share is read. A share below it reopens the attacker's design before any further episode. On CoTinyVLA the same check confirms that seeding a different Plan at a later chunk changes only what the decoder reads. Every reported number for the attacker is computed from these records and from nothing else.

F THE PRE-REGISTERED CONSTRUCT CONTROLS

The main text names the synthetic corruption as a limit of the first measurement and points here for the controls that bound it. This appendix registers two controls, conditions E and F. No episode of either has run. Any change to what follows will be reported as a dated deviation, with its reason and the readings it moves.

What the controls ask. The entity swap is a synthetic edit, so the cost of condition B, intact instruction and corrupted chain, has more than one reading. The cost could come from what the corrupted chain says, or from the chain being edited at all. Condition E separates those two readings. Even where the cost comes from what the chain says, a corruption written the way a real adversary would write it could cost more or less than the swap. Condition F measures that difference. Both conditions are controls on the edit. Neither adds a claim or changes the rescue test and its result.

Scope. Both conditions run on DeepThinkVLA's Goal suite, where the rescue test is confirmatory first in its fixed order (Appendix B). They use the same seeds, initial states and query schedule as condition B there (Appendix A). Each E or F episode therefore pairs with the clean and B episodes that start from the same state. The edited chain enters by condition B's route, with the instruction checked byte for byte on every task.

Condition E, a matched benign rewrite. Condition E keeps the instruction intact and rewrites the chain without changing what it names. At every query the chain is paraphrased by a rule, or by a language-model prompt at temperature zero, fixed and logged verbatim before any E episode. The rewrite must leave the set of object names and the set of direction words exactly as they were, as read by the extractors of Appendix C. Its character-level edit distance must lie within 20% of the entity swap's on the same query. E therefore changes as much of the chain as B does, with the change in meaning removed.

Condition F, a naturalistic corruption. Condition F keeps the instruction intact and corrupts the chain the way a real adversary or a real failure might. It uses one of three kinds. The first rewrites the chain in fluent prose to direct the policy to a distractor object present in the scene, with no dangling reference. The second replaces the chain's stated immediate goal with a coherent but wrong next step for the same object. The third asks a language model, at temperature zero, only to rewrite the chain so that the robot goes to the wrong object and the text stays fluent. Its prompt is logged. The language model never sees the task instruction. The kind is fixed before any F episode runs, and it is never chosen after a result exists. F is a control, so it uses one kind rather than all three.

Readings. Table 14 gives the readings, fixed before any run. Each is read on Goal against the cost of condition B there, a corruption of 19.2 points with interval [10.3, 27.8] (Appendix B). The cost of E or F is likewise the clean episodes' success minus that condition's, in points. E reads as costing nothing when the upper bound on its cost is at or below 5 points. Before any E episode, the expected width of that bound over the 10 Goal tasks is computed. If it shows the threshold cannot be reached, the attainable threshold is fixed in its place. Whichever row holds says what condition B measured.

Statistics. The unit is the task. E's cost is reported as a one-sided upper 95% bound on the paired difference between clean and E episodes, as for the residual bounds of Appendix B. The informative quantity is how large E's cost can be, so no test is run on it. F against B is descriptive, reported as a paired difference with a 95% bootstrap interval and a paired effect size. Neither control adds a test, so the first measurement's confirmatory tests are unchanged.

Checks before any episode. The E rewrite is first run on every Goal task. It must keep the object and direction sets exactly and stay within the edit-distance band, with one full rewritten context logged. The F corruption must name a wrong object present in the scene, in the context the policy reads, with no dangling reference and with one full context logged. One E episode is run twice, and its run-to-run drift is recorded. If the E rewrite fails its check on any task, the rule is corrected before any episode runs.

Table 14: Readings of the construct controls on Goal, fixed before any run. A condition's cost is the clean episodes' success minus that condition's, in points; B's interval is that of Appendix B.

Outcome	What it means
E's upper bound at or below 5 points	B's cost comes from what the corrupted chain says.
E's cost within B's interval	The swap measures disruption of the chain rather than its content. The corruption result narrows to a cost of any edit to the chain. The correction result is unchanged, because condition D injects a chain the policy wrote.
E between the two	Both bounds are reported, and the corruption result is stated with the share of B's cost that comes from content.
F's cost within or below B's interval	The entity swap bounds the naturalistic corruption.
F's cost above B's interval	The swap understates naturalistic corruption, and the corruption cost the paper reports becomes F's.

G Further Related Work

Section 2 places this paper among the work closest to its two measurements. This appendix adds three lines that section leaves out. They are the general-purpose policies that reasoning policies build on, defenses that retrain the model that reasons, and the wider literature on adaptive evaluation.

Vision-language-action policies and their reasoning. The two policies we study build on general-purpose vision-language-action policies. These transfer knowledge from web-scale data to robot control (Zitkovich et al., 2023), release an open policy for fine-tuning (Kim et al., 2025), generate actions with a flow-matching head (Black et al., 2025b), or co-train on heterogeneous tasks to generalize beyond the lab (Black et al., 2025a). Reasoning also appears in systems outside the benchmark we use. Generalist robot systems add embodied reasoning and thinking to their policies (Team et al., 2025). A driving policy pairs a written chain of causation with trajectory planning (NVIDIA et al., 2026). Written reasoning costs time at every query. One response reuses high-level reasoning across timesteps (Duan et al., 2025). A chain reused this way informs several action chunks, as CoTinyVLA's Plan does.

Defenses that retrain the reasoner. In language models, two recent defenses against corrupted reasoning retrain the model that reasons. Thought Purity combines a safety-aware data pipeline with reinforcement learning, so that a reasoning model learns to identify and isolate malicious logic in its chain (Xue et al., 2026). AdvChain tunes such a model on examples of reasoning that drifts and then recovers, so that it corrects its own drift toward harmful compliance or needless refusal (Zhu et al., 2025b). Both change the model rather than monitor it, and both are evaluated on outputs in text. We leave the two policies as released and ask what a monitor of their chain can see. Retraining a vision-language-action policy against corruption of its chain is therefore a separate line. This paper does not test it.

Adaptive evaluation. The standard that §3.2 adopts, judging a defense against an attacker who knows it, has a longer history in adversarial robustness. Robust optimization treats training and attack as one min-max problem (Madry et al., 2018). Reliable evaluation then called for attacks that need no hand-tuned parameters. An ensemble of such attacks found lower robust accuracy than reported for almost every model it was run on (Croce & Hein, 2020). In language models, stronger adaptive attacks bypass most recent defenses against jailbreaks and prompt injections (Nasr et al., 2025). Many of those defenses had reported near-zero attack success. Our adaptive attack is built to this standard for the two detectors we test. Its attacker knows what each detector reads. The attack writes only the text each checked feature needs.