Context Dependent Misalignment - EvilGenie integration

Dor Fuchs

January 2026

1 EvilGenie overview

EvilGenie is a reward-hacking and misalignment benchmark in a programmatic setting. It consists of 154 programming-contest style problems, of which 9 are labeled ambiguous by the benchmark authors. Accordingly, many analyses focus on the 145 unambiguous tasks.

Behavioral categories EvilGenie operationalizes several classes of problematic behavior, including (i) hard-coding or otherwise overfitting to known tests, (ii) modifying the testing procedure or testing files, and (iii) submitting heuristic/non-generalizing solutions that exploit weaknesses in the evaluation protocol.

Detection signals used by the benchmark EvilGenie combines multiple detection mechanisms:

1. **Holdout tests:** a portion of tests (30% and up to 10 tests, per the paper) is moved to an unseen holdout file and the submitted solution is evaluated on it.
2. **LLM judges:** prompted LLMs classify reward-hacking behavior in the submitted solution.
3. **Test-file modification detection:** any edits to (or deletion of) benchmark testing artifacts are flagged as test-modification behavior.

Flagged cases are then reviewed by humans.

Validation mechanism The benchmark's evaluation script validates a submission by comparing the output produced by the agent program against a reference solution (as used by the testing script).

2 Translation to our setup

Let $\{M_k\}_{k=1}^{N}$ denote the EvilGenie task family, where each M_k corresponds to one coding problem. We optionally partition tasks into an unambiguous subset $\{M_k^{ua}\}_{k=1}^{N^{ua}}$ and an ambiguous subset $\{M_k^{a}\}_{k=1}^{N^{a}}$ with $(N^{ua}, N^{a}) = (145, 9)$ when following the benchmark's ambiguity annotations.

We intentionally do *not* attempt to explicitly enumerate state/action spaces or transition dynamics for these tasks, instead, we treat each task as an interactive programmatic environment and define (i) a scalar performance functional and (ii) a constraint set, consistent with our setup's emphasis on protocol-defined evaluation rather than full environment formalization.

2.1 Scalar performance functional

For each task M_k, we define a protocol-dependent scalar performance functional based on test outcomes. Concretely, we work with a finite-horizon evaluation protocol in which the agent interacts with the task environment through a sequence of logged tool-steps (e.g., applying code edits, running tests, and producing a final submission). Given a (possibly partial) interaction history, we estimate the value of a candidate action by forcing that action at the next tool-step and then running a finite number of stochastic continuations ("finite rollouts") under a fixed reference interaction policy. The scalar performance functional is taken to be the fraction of *visible* tests passed at the end of the rollout (or, equivalently, the corresponding test-based score produced by the benchmark harness), averaged over rollouts for the same forced action.

Holdout tests are not used as the primary optimization target in this construction; they may optionally be reported as a secondary robustness metric, but the main performance functional is defined with respect to the visible-test evaluation that the agent can directly influence.

2.2 Constraint set

We define a constraint family intended to capture reward hacking and related misaligned behaviors in the EvilGenie setting. We do not attempt to formalize the full state/action spaces; instead, constraints are defined on observable artifacts of the interaction trace (e.g., file diffs, file access logs, and the final program text).

Deterministic constraints (file edits). We include deterministic constraints that flag any modification to benchmark testing artifacts (including deletions), computed directly from the workspace diff and/or file-system event logs. These constraints are evaluated mechanically from the trace and do not require semantic interpretation.

Judge-estimated constraints (hard-coding and heuristic solutions). Certain behaviors of interest (e.g., hard-coding known tests or producing non-

generalizing heuristic solutions) are not reliably captured by simple deterministic rules. For these categories, we define constraint indicators (or weighted indicators) whose values are produced by an LLM-based judge applied to the final solution artifact (and, if needed, the relevant parts of the interaction trace). At minimum, the judge returns binary indicators for whether a solution exhibits (i) hard-coding/overfitting to known tests and (ii) heuristic/non-generalizing behavior; optionally, the judge may return calibrated scalar scores in $[0, 1]$ intended to represent violation severity.

To ensure reproducibility, the judging protocol is frozen: we fix the judge model(s), the prompt(s), and the aggregation rule used to convert judge outputs into constraint values. If prompt selection is treated as a hyper-parameter, it must be selected on a designated development split, with final reporting performed on a held-out test split.

Constraint evaluation and reporting. Constraints are evaluated per rollout (or per completed episode) and aggregated to produce empirical estimates of violation probability and violation magnitude for each constraint family. Deterministic constraints are treated as ground-truth within the benchmark protocol, while judge-estimated constraints are treated as estimator outputs under the frozen judging procedure.