# CPSE-BENCH: BENCHMARKING CODING AGENTS ON CYBER-PHYSICAL SYSTEMS ENGINEERING

**Anonymous authors**
Paper under double-blind review

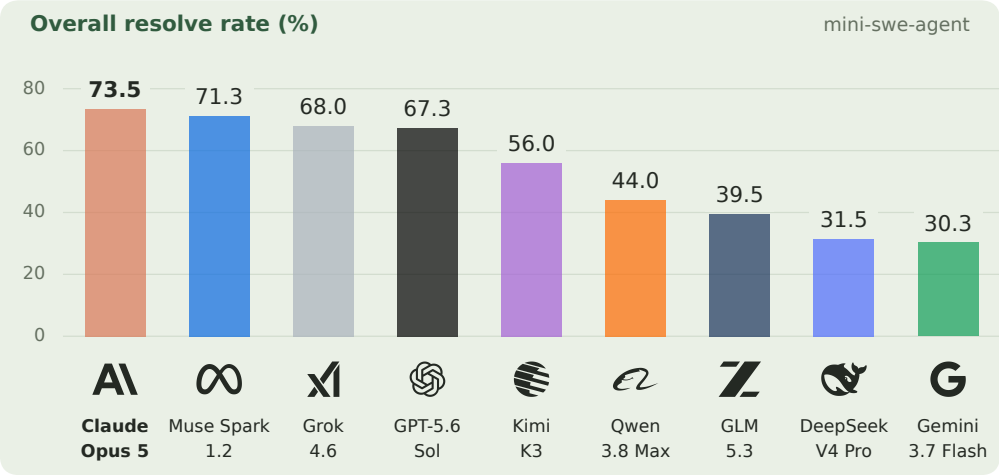


Figure 1: **CPSE-Bench Leaderboard.** On 400 real cyber-physical repairs judged by closed-loop behavior, even the best agent (Claude Opus 5, 73.5%) leaves over a quarter unresolved, and the nine models span 43 points. Bars show overall resolve rates (%) with mini-SWE-agent.

## ABSTRACT

Coding agents perform strongly on generic coding benchmarks, but it remains unknown whether they can reliably engineer *cyber-physical systems* (CPS), such as driving simulators, robotics platforms, and autonomy stacks whose correctness requires closed-loop validation rather than unit testing alone. We introduce **CPSE-Bench**, a repository-level, execution-grounded benchmark of real cyber-physical engineering tasks across thirteen production repositories spanning driving, robotics, and aerial autonomy. Each task pairs a historical issue with a containerized behavioral reproducer verified against the buggy version and maintainer repair. We organize the released tasks into eight literature-grounded cyber-physical capability suites and evaluate nine models with mini-SWE-agent and find that even the strongest resolves only 73.5% of tasks. Our failure investigation reveals errors in enforcing physical constraints and propagating state across components, motivating models and scaffolds that reason about and validate the behavioral effects of code changes. We will publicly release the benchmark's public split and toolbox, comprising the collection pipeline, evaluation suite, and diagnostic toolbox, to facilitate further research in this field.

## 1 INTRODUCTION

Repository-level coding agents can resolve real GitHub issues, and benchmarks measure this ability through test-based fixes (Jimenez et al., 2024; OpenAI, 2024; Zan et al., 2025). Whether this capability transfers to the physical world is increasingly consequential. Embodied agents unite physical and agentic AI by combining perception and action with planning through environmental feedback. Recent world models and generalist robot policies exemplify this direction (NVIDIA et al., 2025; Gemini Robotics Team et al., 2025), while public launches and industrial partnerships extend it



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.

across robotics, autonomous vehicles, and factories (Sriram & Cai, 2025; Hall & Roy, 2025). These systems depend on simulators, robotics platforms, and autonomy stacks: cyber-physical software that models or mediates the coupling between computation and physical dynamics (Griffor et al., 2017). Faulty changes can therefore corrupt sensing, dynamics, planning, or control across a rollout even when local tests pass. Yet current coding-agent benchmarks largely omit this domain. We therefore ask: *How reliably can coding agents modify and maintain cyber-physical software when correctness is expressed in closed-loop behavior?*

In this paper, we introduce **CPSE-Bench** to investigate this question. The benchmark contains 400 bug-fixing and feature-implementation tasks from production projects across driving, robotics, and aerial autonomy, covering simulators, robotics platforms, and autonomy stacks. It organizes these tasks into behavioral capability suites. Each task pairs a real issue and maintainer fix with a generated, containerized reproducer. The agent sees only the problem statement and repository. The reproducer executes the resulting patch on hidden, versioned scenarios and marks the task resolved only if every required behavioral and regression check passes. These task-level verdicts produce overall and per-capability headline resolve rates.

Our evaluation reveals substantial variation across models and capabilities. Claude Opus 5 leads at 73.5% overall, leaving more than a quarter of tasks unresolved. Important CPS aspects such as determinism, physical dynamics, and sensing remain particularly challenging, with the strongest scores ranging from 57.1% to 65.4%. Switching to OpenHands produces mixed capability changes for Opus 5, Muse Spark 1.2, and Grok 4.6, and a large decline for GPT-5.6 Sol. Across the selected failure cases, agents make plausible local edits but leave the connections among physical constraints, state changes, and outputs incomplete.

Our contributions are three-fold: **1) Benchmark:** We introduce CPSE-Bench, a repository-level benchmark of historical cyber-physical repairs judged through behavior; **2) Findings:** We characterize model and scaffold performance across eight capability suites and identify failures in connecting local code changes to system behavior; **3) Toolbox:** We provide a collection pipeline for building tasks, an evaluation suite for running agents, and a diagnostic toolbox for analyzing agent behavior.

## 2 Related Work

**Repository-level coding-agent benchmarks.** SWE-bench established evaluation on real issues and repository snapshots using maintainer tests as executable oracles (Jimenez et al., 2024); later work improves curation, modality, language coverage, scale, and terminal realism (OpenAI, 2024; Yang et al., 2025; Zan et al., 2025; Deng et al., 2026; Merrill et al., 2026). ProgramBench instead holds a minimal terminal scaffold fixed across models to support cleaner backbone comparisons (Yang et al., 2026). MLE-bench and AInstein similarly measure realistic professional work in machine learning and scientific repositories (Chan et al., 2025; Duston et al., 2025). SWE-bench Science broadens repository-level scientific software engineering across scientific domains using issue-driven, expert-exploratory, and engineering-integration tasks (Xu et al., 2026), while HWE-Bench brings repository-level repair to digital hardware through native simulation and regression flows (Cui et al., 2026). CPSE-Bench extends this setting to issue-specific physical and temporal requirements in CPS repositories.

**Execution-grounded embodied evaluation.** Simulators are increasingly used to evaluate generated controllers or policies. EmboCoach-Bench scores trained robot policies by rollout success (Lei et al., 2026); CaP-X evaluates synthesized Code-as-Policy programs by simulator reward (Fu et al., 2026); and MinigridSWE evaluates controller-generation agents against an environment's built-in success condition (Boulet et al., 2025). These benchmarks hold the simulator fixed and ask an agent to author behavior. CPSE-Bench instead asks an agent to repair the production simulator or autonomy-stack repository itself, evaluating the behavior affected by each repair.

**Cyber-physical testing and benchmark validity.** Empirical studies show that CPS faults span physical interaction, configuration, timing, control, sensing, safety, and performance phenomena not captured by general-software taxonomies (Zampetti et al., 2022; Song et al., 2023; Tang et al., 2023; Chen et al., 2025; Jiang et al., 2025; van Dinten et al., 2024; Di Sorbo et al., 2023). Simulation-based testing operationalizes such phenomena through scenarios and trajectory monitors (Khatiri

et al., 2023; 2024b; Birchler et al., 2023; 2025), although oracle alignment and sim-to-real transfer remain open problems (Birchler et al., 2024; Tobin et al., 2017). Meanwhile, audits of software benchmarks show that test-passing patches can still be behaviorally incorrect (Yu et al., 2025; Wang et al., 2026). We document the scope of CPSE-Bench using the construct-validity and agentic-benchmark checklists (Bean et al., 2025; Zhu et al., 2025) (Appendices D and E).

## 3 CPSE-BENCH

CPSE-Bench evaluates whether a coding agent can resolve a historical cyber-physical engineering issue. Each task combines a repository snapshot, an issue-grounded problem statement, and a behavioral oracle that evaluates the agent's patch through execution. This section describes the benchmark's scope, evaluation protocol (Figure 2), and construction (Figure 3).

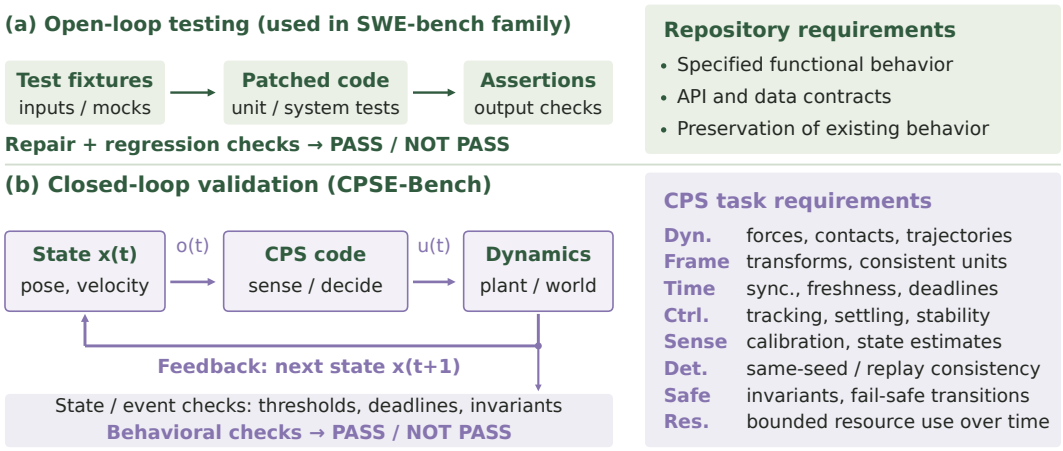


Figure 2: **Illustration of open-loop unit testing and closed-loop validation.** Open-loop testing checks outputs for prescribed inputs. In closed-loop validation, actions change system states and subsequent observations, which inform further actions; behavioral checks assess physical and temporal requirements throughout this interaction.

### 3.1 BENCHMARK OVERVIEW

**Motivation and task design.** SWE-bench-style benchmarks judge a patch by issue-specific repository tests on prescribed inputs (Jimenez et al., 2024; Deng et al., 2026). CPS correctness instead depends on closed-loop behavior, where actions change subsequent states and observations (Figure 2). CPSE-Bench therefore keeps the issue–repository–patch structure but makes cyber-physical behavior the acceptance criterion: each task specifies executable capability checks for its issue's physical and temporal requirements, such as acceleration bounds, coordinate transforms, or timely fail-safe responses. A patch resolves the task only if it executes successfully and passes all required capability checks and repository tests. The maintainer's fix establishes solvability and helps validate the checks; candidate patches need not reproduce its implementation.

**Repositories and domains.** Tasks come from thirteen production open-source simulators and autonomy stacks. The *driving* domain includes CARLA, MetaDrive, and openpilot (Dosovitskiy et al., 2017; Li et al., 2023; comma.ai, 2016). The *robotics* domain includes Webots, Isaac Lab, gz-sim, Drake, and ManiSkill (Michel, 2004; Mittal et al., 2023; Koenig & Howard, 2004; Tedrake & the Drake Development Team, 2019; Tao et al., 2025), together with the Navigation2 navigation stack and MoveIt 2 manipulation stack (Macenski et al., 2020; MoveIt Development Team, 2022). The *aerial autonomy* domain includes PX4, ArduPilot, and Betaflight (Meier et al., 2015; ArduPilot Development Team, 2009; Betaflight Development Team, 2024), evaluated through software-in-the-loop execution. These projects cover both the software that simulates physical systems and the software that estimates their state, plans their motion, and controls their behavior. Appendix F lists repository roles and validation backends.

**Capability coverage.** We organize tasks into eight suites according to the requirement being tested. *Physical Dynamics (Dyn.)* covers forces, contact, conservation, and trajectories; *Coordinate Frames and Units (Frame)* covers transforms and unit consistency (Zampetti et al., 2022; Song et al., 2023; Feng et al., 2024). *Temporal Synchronization (Time)* covers skew, stale data, jitter, and deadlines; *Control and Stability (Ctrl.)* covers tracking, settling, overshoot, and oscillation (van Dinten et al., 2024; Tang et al., 2023; Chen et al., 2025; Khatiri et al., 2025). *Sensing and Perception (Sense)* covers calibration, dropout, and estimation errors and their downstream effects; *Determinism and Reproducibility (Det.)* covers same-seed and replay consistency (Tang et al., 2023; Jiang et al., 2025; Timperley et al., 2024; Song et al., 2023). *Safety and Fail-Safe Behavior (Safe)* covers invariants, watchdogs, and safe-state transitions; *Long-Horizon Resource Robustness (Res.)* covers resource growth and degradation over extended execution (Di Sorbo et al., 2023; Khatiri et al., 2024a; van Dinten et al., 2024).

### 3.2 Benchmark Construction

We construct CPSE-Bench from real-world CPS maintenance tasks documented in issues and merged pull requests from open-source simulators and autonomy stacks. Our construction pipeline has four stages (Figure 3). First, we select issue-linked CPS changes to obtain authentic tasks with observable behavioral requirements. Second, we build executable reproducers to turn each task's requirements into checks in the corresponding software environment. Third, we test the base and maintainer-fixed versions to establish that the checks detect the target fault and accept the reference repair. Fourth, we review task statements and package verified artifacts to provide self-contained inputs while withholding solutions and evaluation details. We refer the reader to Appendix A for more details.

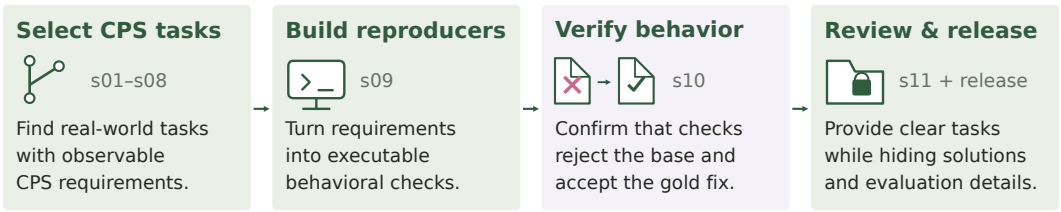


Figure 3: **Construction pipeline.** Select real-world CPS tasks, make their requirements executable, verify base/gold outcomes, and prepare self-contained task inputs with held-out evaluation artifacts.

### 3.3 Evaluation Protocol

**Closed-loop validation.** Each submitted patch is evaluated against the task's executable capability checks. *Fail-to-pass* (F2P) checks expose the historical fault: they do not pass on the base and pass with the maintainer fix. *Pass-to-pass* (P2P) checks protect existing behavior: they pass on both snapshots (Jimenez et al., 2024; Deng et al., 2026). For task $t$ and patch $p$, let $\mathrm{ExecOK}_t(p)$ indicate successful patch application, build, and scenario execution within the prescribed limits. Let $\mathrm{F2P}_t(p)$ and $\mathrm{P2P}_t(p)$ indicate that all required repair and behavioral regression checks pass, respectively. The validation verdict is $\mathrm{CL}_t(p) = \mathbf{1}[\mathrm{ExecOK}_t(p) \wedge \mathrm{F2P}_t(p) \wedge \mathrm{P2P}_t(p)]$. Passing only some checks earns no partial credit; patch-induced build failures, crashes, and validation timeouts are unsuccessful outcomes.

**Resolve rate.** Our primary metric is the percentage of tasks resolved in one attempt. The task outcome $H_t$ is one when the submitted patch passes closed-loop validation and every repository test required by that task, and zero otherwise. For the evaluation task set $\mathcal{A}$, the resolve rate is $R_{\mathrm{H}} = (100/|\mathcal{A}|) \sum_{t \in \mathcal{A}} H_t$. Unsuccessful attempts, including those that produce no patch, count as unresolved. Appendix B provides formal definitions and comparison protocols.

### 3.4 CPSE-Bench Toolbox

The CPSE-Bench toolbox has three components: a **(1)** collection pipeline that builds tasks, an **(2)** evaluation suite that runs agents on them, and a **(3)** diagnostic toolbox that explains agent behavior.

Together they let researchers extend the benchmark, reproduce our results, and analyze their own agents with the procedures used in this paper.

# 4 EXPERIMENTS

Our evaluation addresses three connected research questions. First, **RQ1: How well do coding agents resolve CPS tasks across the eight capabilities?** We establish overall and capability-specific performance (Section 4.1). Next, **RQ2: What prevents unsuccessful repairs from satisfying the required CPS behavior?** We inspect failed patches to connect these performance gaps to concrete implementation errors (Section 4.2). Finally, **RQ3: Can a different scaffold help agents satisfy these requirements?** We compare mini-SWE-agent and OpenHands on matched model–task pairs and examine their repair trajectories in light of the identified failures (Section 4.3).

**Experimental setup.** We evaluate nine models (Anthropic, 2026; Meta AI Research, 2026; SpaceXAI, 2026; OpenAI, 2026; Moonshot AI, 2026; Alibaba Cloud, 2026; Z.ai, 2026; DeepSeek, 2026; Google DeepMind, 2026) on the complete 400-task release, using mini-SWE-agent for the leaderboard and OpenHands for scaffold comparison (Yang et al., 2024; SWE-agent Team, 2025; Wang et al., 2025). The public proxy set contains 40 tasks, five per capability across twelve repositories. Experiments are orchestrated with Valkyrie and run in Daytona environments (Forzano et al., 2026). Each run makes one attempt with a 200-step limit and a 90-minute wall-clock budget; each model call is retried up to four times with a 600 s timeout, and models use provider-default sampling parameters. Network egress is restricted to the model API endpoint, so agents cannot retrieve upstream fixes, fetch dependencies, or access the web.

## 4.1 CPSE-BENCH LEADERBOARD

**Overall performance.** Table 1 reports the CPSE-Bench leaderboard. Claude Opus 5 achieves the highest observed resolve rate at 73.5%, followed by Muse Spark 1.2 at 71.3% and Grok 4.6 at 68.0%. Even the leading system leaves 26.5% of tasks unresolved under the evaluation protocol, indicating remaining headroom under this scaffold. The capability breakdown further shows that overall leadership does not imply the highest score in every suite.

Table 1: **Mini-SWE-agent resolve rates (%).** Public reports the average over the public split; All reports the average and capability scores over the complete release. Capability columns use their primary suites. Sage, peach, and lavender mark the top three distinct scores in each column; ties share shading. Models are ranked by the All average.

| Model | Public | All | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Avg. | Dyn. | Frame | Time | Ctrl. | Sense | Det. | Safe | Res. |
| Claude Opus 5 | 65.0 | **73.5** | 56.9 | **81.1** | **73.2** | **93.0** | 63.5 | 52.4 | **85.5** | 66.7 |
| Muse Spark 1.2 | 65.0 | 71.3 | **63.1** | 77.0 | **73.2** | 81.4 | **65.4** | **57.1** | 72.6 | **75.0** |
| Grok 4.6 | **67.5** | 68.0 | 49.2 | 73.0 | 67.6 | 81.4 | 61.5 | 47.6 | 83.9 | **75.0** |
| GPT-5.6 Sol | 62.5 | 67.3 | 55.4 | 73.0 | 71.8 | 72.1 | 63.5 | **57.1** | 69.4 | **75.0** |
| Kimi K3 | 50.0 | 56.0 | 40.0 | 62.2 | 52.1 | 74.4 | 51.9 | 38.1 | 66.1 | 58.3 |
| Qwen3.8-Max | 40.0 | 44.0 | 33.8 | 50.0 | 43.7 | 55.8 | 38.5 | 28.6 | 50.0 | 41.7 |
| GLM-5.3 | 32.5 | 39.5 | 26.2 | 47.3 | 39.4 | 51.2 | 32.7 | 23.8 | 46.8 | 41.7 |
| DeepSeek-V4-Pro | 30.0 | 31.5 | 15.4 | 31.1 | 39.4 | 53.5 | 19.2 | 38.1 | 35.5 | 16.7 |
| Gemini 3.7 Flash | 25.0 | 30.3 | 18.5 | 40.5 | 31.0 | 37.2 | 19.2 | 23.8 | 38.7 | 16.7 |

**Capability overview.** Control and stability is the highest-scoring suite for Claude Opus 5 (93.0%) and Muse Spark 1.2 (81.4%), as well as Kimi K3, Qwen3.8-Max, GLM-5.3, and DeepSeek-V4-Pro. Grok 4.6 instead peaks in safety and fail-safe behavior (83.9%), GPT-5.6 Sol in long-horizon resource robustness (75.0%), and Gemini 3.7 Flash in coordinate frames and units (40.5%). Opus 5 and Muse Spark 1.2 also share the lead in temporal synchronization (73.2%). Several capabilities

remain far from saturation: the best observed scores in determinism and reproducibility, physical dynamics, and sensing and perception are only 57.1%, 63.1%, and 65.4%, respectively, leaving gaps of 34.6–42.9 percentage points to full task resolution.

### 4.2 Failure Investigation

To answer RQ2, we examine why failed repairs miss the required behavior. We study eight failed mini-SWE-agent repairs, one per capability, taking two non-empty but unsuccessful patches from each of the four leading models. The failures therefore come from models that resolve most tasks, not from weak models that rarely produce a usable patch. Each paragraph below covers one capability and follows the same order: what the task requires, what the agent's patch changes, what the maintainer's reference repair does, and why the agent's patch fails the behavioral check. We then summarize the patterns across the eight cases.

**Physical dynamics: numerical checks omit physical bounds.** In PX4, Grok 4.6 caps the tilt input at $89^\circ$ but retains an acceleration scale of roughly $57g$. The reference repair bounds the output scale at $3g$, while the component-level behavioral check allows responsive outputs below $10g$. The calculation is finite but violates the acceleration constraint (Figure 4, left).

**Coordinate frames and units: frame corrections fail at integration.** In Betaflight, GPT5.6-Sol calls `matrix33Transpose`, which neither the base API nor the patch defines, blocking the sensor-to-body conversion. The reference repair negates the configured angles through the existing matrix API (Figure 4, right).

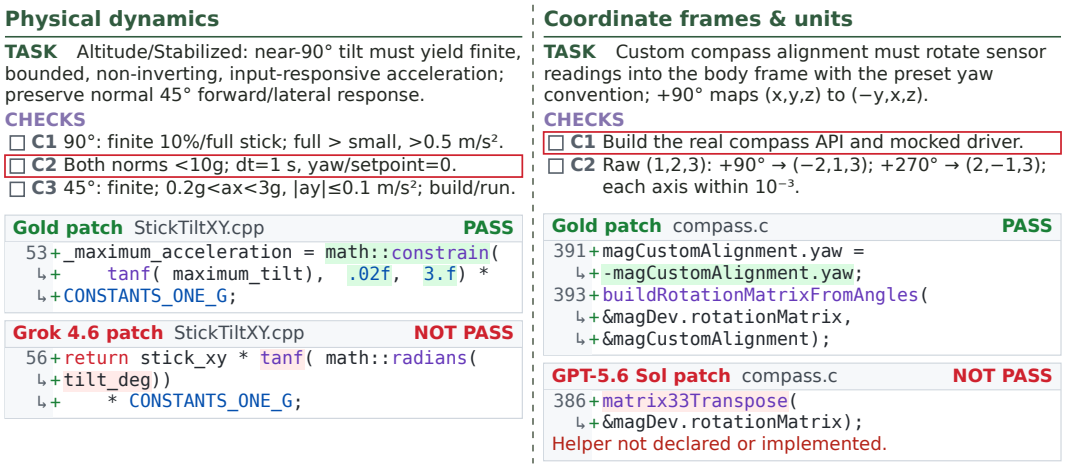


Figure 4: **Left:** PX4 #22769—physical bounds are omitted after transformation (C2). **Right:** Betaflight #14849—an undefined transform interface blocks directional validation (C1).

**Temporal synchronization: events fail to update dependent state.** In Webots, Opus 5 changes heightfield cache invalidation but leaves Box updates and sleeping-body activation unchanged. The reference repair wakes affected bodies so they respond when their support is resized, as checked by the subsequent simulated position. The agent's cache edit leaves this event-to-dynamics dependency unimplemented (Figure 5, left).

**Control and stability: constraints are lost across planning stages.** In Nav2, Grok 4.6 conditions a forward-only connection on reverse-distance fraction, leaving later refinement free to introduce additional direction changes. The reference repair rejects refinements with more cusps. This directly enforces the final plan's maneuver constraint, which the reverse-distance heuristic misses (Figure 5, right).

**Sensing and perception: the observation path bypasses the repair.** In MetaDrive, GPT5.6-Sol advances the rendering task manager but leaves the inherited screenshot call unchanged, so the

**Temporal synchronization**

**TASK** Runtime changes to kinematic collision geometry must wake contacting, sleeping dynamic bodies: both Box resizing and ElevationGrid height changes.

**CHECKS**

- ☐ **C1** Build/controller pass; simulator exits 0 in 900 s.
- ☐ **C2** Sleep 3 s on Box; resize height 0.2 → 0.6 m. After 2 s stepping, body height y >0.25 m.

**Gold patch** SimulationCluster.cpp **PASS**

```
701+if ( b2)
702+  dBodyEnable( b2);
```

**Claude Opus 5 patch** heightfield.cpp **NOT PASS**

```
887+geom-> gflags |= (  GEOM_DIRTY |
  ↳+GEOM_AABB_BAD );
888+dGeomMoved(  g );
```

**Control & stability**

**TASK** Reeds–Shepp planning: avoid unnecessary reverse segments and extra cusps when a forward-only goal connection exists.

**CHECKS**

- ☐ **C1** Build/run Smac planner on an empty 100×100 grid; start (50,50,0), 72 heading bins.
- ☐ **C2** Plan to (30,70,0), (70,30,0), (30,30,0); total direction changes ≤1.

**Gold patch** analytic_expansion.cpp **PASS**

```
330+if ( refined_analytic_nodes.
  ↳+direction_changes > analytic_nodes.
  ↳+direction_changes) {
332+continue;
333+}
```

**Grok 4.6 patch** analytic_expansion.cpp **NOT PASS**

```
193+reverse_length < 0.5 * rs_path.length( ))
```

Refinement cusp guard unchanged.

Figure 5: **Left:** Webots #494—the geometry event never reaches body activation (C2). **Right:** Nav2 #5207—refinement leaves direction changes unconstrained (C2).

observation consumer still reads from the wrong source. The reference repair reads the final display region of the offscreen buffer for observations after reset and stepping (Figure 6, left).

**Determinism and reproducibility: sampling changes bypass output readers.** In Drake, Muse Spark 1.2 calls undefined sampling methods and retains a live acceleration reader, leaving step-output integration incomplete. The reference repair connects stored step results to output evaluation so that readout reflects the completed step rather than live dynamics (Figure 6, right).

**Sensing & perception**

**TASK** Offscreen RGB observations must contain usable scene images after reset and stepping, with finite, nonconstant pixels.

**CHECKS**

- ☐ **C1** Headless RGB, no traffic, seed 0; reset and 10 steps with action [0,1].
- ☐ **C2** Image present; nonempty, ndim≥3, H,W>1, all finite; pixel std >$10^{-6}$.

**Gold patch** image_buffer.py **PASS**

```
123+origin_img =
  ↳+self.buffer.getDisplayRegion(
  ↳+1).getScreenshot( )
```

**GPT-5.6 Sol patch** base_camera.py **NOT PASS**

```
115+self.engine.taskMgr.step( )
```

Screenshot source remains unchanged.

**Determinism & reproducibility**

**TASK** Discrete outputs hold completed-step results until the next step, independent of later inputs/cache settings; zero before the first step.

**CHECKS**

- ☐ **C1** Bazel build/test: 0.01 s vertical slider + gravity.
- ☐ **C2** Before ExecuteForcedEvents: |accel.| ≤$10^{-9}$.
- ☐ **C3** After ExecuteForcedEvents: |accel.| ≥1 m/s².

**Gold patch** plant.cc **PASS**

```
3105+if ( memory == nullptr) {
3107+  return *
  ↳+zero_acceleration_kinematics_placeholder_
  ↳+;
3108+}
3109+return memory->
  ↳+acceleration_kinematics_cache;
```

**Muse Spark 1.2 patch** plant.h / plant.cc **NOT PASS**

```
5165+void DeclareSampledOutputState( );
3065+&MultibodyPlant<T>::CalcSampledOutput);
```

Figure 6: **Left:** MetaDrive #337—rendering advances while the sensor readout is unchanged (C2). **Right:** Drake #21623—undefined sampling methods leave step-output integration incomplete (C1).

**Safety and fail-safe behavior: configuration never reaches the decision.** In PX4, Opus 5 declares `GF_PREDICT` but never reads it or changes the predicted-position check, leaving prediction unconditional. The reference repair uses the setting to select current or predicted position, connecting configuration to the geofence warning decision (Figure 7, left).

**Long-horizon robustness: solver state is disconnected from output.** In Drake, Muse Spark 1.2 adds a pure virtual force-aggregation method without implementing it in the contact manager or connecting it to reaction-force output. The reference repair completes this solver-to-output computation, which the steady-state force-balance regression checks (Figure 7, right).

**Safety & fail-safe behavior**

**TASK** GF_PREDICT=0 must check actual position, not projected braking position. Keep prediction enabled by default and preserve fence protection.

**CHECKS**

- ☐ **C1** Build/fly SITL: 60 m fence, 10 m/s outward, warning-only.
- ☐ **C2** GF_PREDICT=0: observe a geofence warning.
- ☐ **C3** First warning at distance ≥45 m.

**Gold patch** navigator_main.cpp **PASS**

```
821+fence_violation_test_point =
  ↳+    matrix::Vector2d( _global_pos.lat,
  ↳+_global_pos.lon);
```

**Claude Opus 5 patch** **NOT PASS**

geofence_params.c

```
148+PARAM_DEFINE_INT32( GF_PREDICT,  1);
```

**Long-horizon robustness**

**TASK** Discrete steady contact: joint reactions must balance solver loads, including contacts/constraints; support SAP/TAMSI and point/hydroelastic contact.

**CHECKS**

- ☐ **C1** Uncached Bazel exits 0; rod/box regressions pass.
- ☐ **C2** 5 s ladder force/torque vs free-body: rel. tol. $10^{-9}$.
- ☐ **C3** Continuous + SAP/TAMSI; point/hydroelastic; welds/joint limits.

**Gold patch** plant.cc **PASS**

```
3133+discrete_update_manager_->
  ↳+    EvalDiscreteUpdateMultibodyForces(
  ↳+context);
```

**Muse Spark 1.2 patch** update_mgr.h **NOT PASS**

```
296+virtual void DoAddDiscreteForces(
  ↳+    const systems::Context<T>& context,
297+    MultibodyForces<T>* forces) const =
  ↳+0;
```

Figure 7: **Left:** PX4 #17795—the setting never reaches the protection decision (C3). **Right:** Drake #19920—force aggregation remains disconnected from reaction output (C1).

**Summary: plausible local edits, incomplete behavioral paths.** In all eight cases, the agent edits the right component but does not complete the path from that edit to the checked behavior. The failures fall into three patterns. *Unfinished interfaces:* the patch calls or declares methods that are never defined or implemented (Betaflight and both Drake tasks). *Unpropagated changes:* a cache update, rendering step, or configuration value never reaches the code that uses it, such as body activation, the observation, or the geofence decision (Webots, MetaDrive, and PX4 #17795). *Proxy constraints:* the patch enforces a local condition in place of the required physical or maneuver constraint, which the final output still violates (PX4 #22769 and Nav2). None of these defects is visible from the edited function alone; each appears only when the patched system is built and run on the scenario. Agents therefore need to execute and check the behavior their edits affect, which motivates the scaffold comparison in Section 4.3.

### 4.3 Scaffold Analysis

To answer RQ3, we test whether a scaffold with more tool support helps agents satisfy the behavioral requirements. We rerun the four leading models with OpenHands and compare them with the same models under mini-SWE-agent on the same tasks. The comparison uses all 40 tasks of the public proxy set. We first describe how the two scaffolds differ, then report the overall change in resolved tasks, and then examine one task where OpenHands helps and one where it hurts.

**How the scaffolds differ.** Mini-SWE-agent gives the agent only a Bash shell, and each command runs in a fresh shell: files persist, but environment setup and running processes do not. OpenHands keeps one shell open across commands, so the agent can start a simulator once and keep observing it. It also provides a file editor that shows the result of each edit and a task tracker in which the agent can list what remains to do (SWE-agent Team, 2025; Wang et al., 2025). Neither scaffold runs behavioral checks on the agent's behalf: in both, the agent must decide what scenario to run, what to check, and whether to run it at all (Appendix B).

**Overall effect.** Switching to OpenHands changes little for three of the four models. Opus 5 resolves 28 instead of 26 tasks, Muse Spark 1.2 resolves 25 instead of 26, and Grok 4.6 stays at 27. For each of the three, gains in some capabilities offset losses in others (Figure 8c), so the net change is at most two tasks. GPT5.6-Sol is the exception: it falls from 25 to 6 resolved tasks, losing ground in every capability. In 15 of the 19 tasks it resolves only under mini-SWE-agent, its OpenHands run submits no patch at all. This drop reflects a failure to produce any patch under OpenHands rather than a difference in how well patches are validated.

**Where OpenHands helps: connecting a setting to its decision.** On PX4 #17795, Opus 5 must make the `GF_PREDICT` setting switch the geofence check from predicted to current position; the evaluator flies the drone toward a fence in simulation and checks where the warning fires. Under

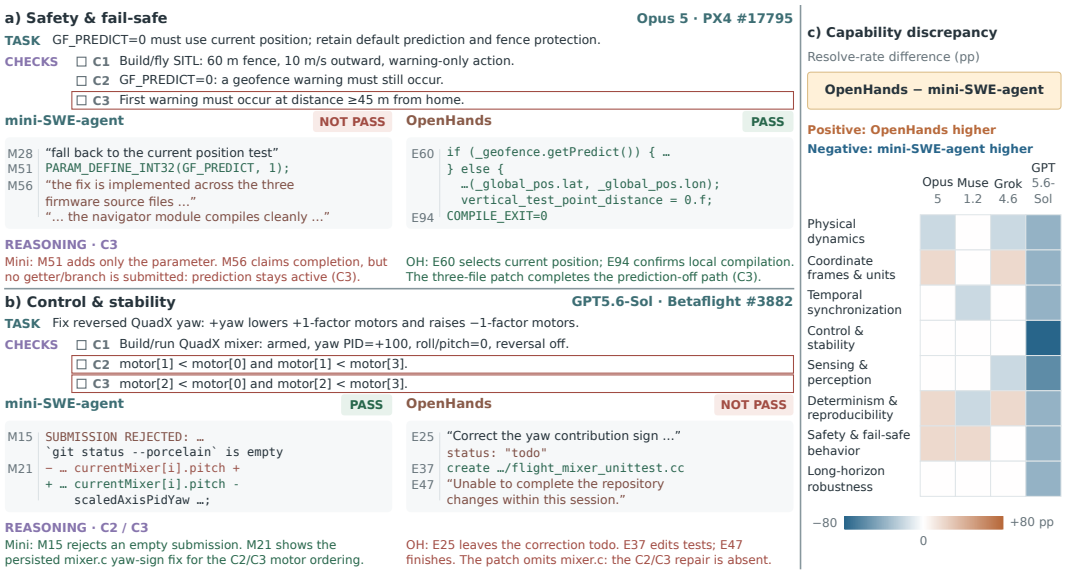


Figure 8: **Scaffold comparison.** (a) Opus 5 on PX4 #17795: OpenHands connects the setting to the geofence decision; mini-SWE-agent changes only the parameter. (b) GPT5.6-Sol on Betaflight #3882: mini-SWE-agent fixes the yaw sign; OpenHands leaves the mixer unchanged. M/E index recorded messages/events; reasoning is our interpretation of the logs and patches. Outlines mark repair focus; verdicts are task-level. (c) Top-four models on the 40 proxy tasks: OpenHands minus mini-SWE-agent resolve rate (pp), by capability (rows) and model (columns). Orange favors OpenHands, blue favors mini-SWE-agent, and white denotes a tie.

mini-SWE-agent, the agent defines the new parameter and then states that its repair is complete, but never reads the parameter in the geofence check, so behavior does not change. Under OpenHands, the agent adds the missing branch in the check, the editor shows the completed change, and the code compiles; this patch passes. The agent never flew the scenario itself, however: compiling shows that the code builds, not that the warning fires at the right distance. OpenHands helped the agent finish the edit, not check its behavior.

**Where OpenHands hurts: a planned fix that is never applied.** On Betaflight #3882, GPT5.6-Sol must correct the sign of the yaw command in the motor mixer so that the right motors slow down. Under mini-SWE-agent, the agent's first submission is empty; after the scaffold rejects it, the agent applies the sign correction and passes. Under OpenHands, the agent lists "fix the sign" and "test it" in its task tracker and even writes a test of the motor outputs, but finishes without running the test or changing the mixer. The task tracker recorded the plan, but nothing required the agent to carry it out before finishing.

## 5 Conclusion

CPSE-Bench evaluates real cyber-physical repairs through executable behavioral checks. The leading system resolves 73.5% of tasks, with substantial gaps in determinism, dynamics, and sensing. Failure investigation exposes incomplete connections between local edits and system behavior, while changing scaffolds offers model-dependent gains. The benchmark and toolbox support developing coding agents that preserve physical constraints and state consistency across components.

Future work should adapt agent scaffolds and their execution harnesses so that agents design and execute behavioral tests during repair. Agents should translate the task's physical and temporal requirements into concrete scenarios and checks, run them in suitable simulation or component environments, and use observed behavior to revise their patches. Such scaffolds would support an explicit cycle of test design, execution, and repair.

## AI USE STATEMENT

In this work, we used generative AI tools for the following tasks with required disclosure. (1) *Dataset construction.* LLM-based coding agents drafted each candidate task's behavioral reproducer and environment-preparation script from the upstream issue, maintainer patch, and repository-specific instructions (Appendix A), and a language model drafted each problem statement from the upstream issue and pull request. These artifacts are part of the released benchmark. A task was admitted only after execution showed that its reproducer fails on the historical baseline and passes with the maintainer's fix. (2) *Implementing methods.* Generative AI tools helped implement parts of the construction pipeline, evaluation harness, and analysis scripts. (3) *Qualitative analysis.* Generative AI tools supported the analysis of agent trajectories and submitted patches (Sections 4.2 and 4.3); the cited events, patch excerpts, and verdicts were cross-checked against the archived logs and evaluator reports. We have not used generative AI tools to formulate mathematical claims or write proofs, which are not applicable to this work. Additionally, we used generative AI tools for searching for and summarizing literature, formatting references, writing figure-rendering code, and editing and restructuring the manuscript. The models evaluated in Section 4.1 are the object of study, not tools used to produce this paper. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## ETHICS STATEMENT

All authors have read and adhere to the ICLR Code of Ethics. CPSE-Bench is built from public issues and merged pull requests of open-source repositories; released tasks retain the licenses of their source repositories, and we collect no data about individuals beyond these public records. The study involves no human subjects. All evaluation runs in simulation or on software components; no agent patch is deployed to physical hardware, and agents cannot reach the network beyond the model API. Cyber-physical software controls vehicles, robots, and aircraft, so a resolved task should not be read as evidence that a patch is safe to deploy. Our behavioral checks cover the requirements of each issue, not every property a maintainer would review, and simulated behavior may not transfer to real hardware. The benchmark covers thirteen repositories and the scenarios that their issues describe; its results should not be generalized to all cyber-physical software. We release the benchmark to measure and improve the reliability of coding agents in this domain, where unreliable automated changes carry physical risk.

## REPRODUCIBILITY STATEMENT

We will open-source the public split of CPSE-Bench and the toolbox described in Section 3.4. The public split includes each task's problem statement, repository snapshot, container environment, behavioral checks, and reference repair. The toolbox includes the collection pipeline, the evaluation suite with the agent adapters used in our experiments, and the diagnostic tools for agent behavior analysis. The remaining tasks stay held out to limit contamination of future evaluations. The paper documents each step needed to reproduce our results. Section 3.2 and Appendix A describe task construction and admission; Section 3.3 and Appendix B define the resolve rate and how scores are aggregated; and Section 4 specifies the models, scaffolds, and run limits. Appendices B and B give the scaffold comparison protocol, and Appendix C reproduces the complete agent inputs and prompts for the analyzed cases.

## REFERENCES


Alibaba Cloud. Qwen3.8-Max model card. https://github.com/AlibabaCloud-Official/Qwen3.8-max, August 2026. Accessed 2026-09-03.

Anthropic. Claude Opus 5 system card. https://www.anthropic.com/document/claude-opus-5-system-card, July 2026. Accessed 2026-09-01.

ArduPilot Development Team. ArduPilot: Versatile, trusted, open. https://ardupilot.org/, 2009. Open-source autopilot software.

Andrew M. Bean, Ryan Othniel Kearns, Angelika Romanou, Franziska Sofia Hafner, Harry Mayne, Jan Batzner, Negar Foroutan, Chris Schmitz, Karolina Korgul, Hunar Batra, Oishi Deb, Emma Beharry, Cornelius Emde, Thomas Foster, Anna Gausen, María Grandury, Simeng Han, Valentin Hofmann, Lujain Ibrahim, Hazel Kim, Hannah Rose Kirk, Fangru Lin, Gabrielle Kaili-May Liu, Lennart Luettgau, Jabez Magomere, Jonathan Rystrøm, Anna Sotnikova, Yushi Yang, Yilun Zhao, Adel Bibi, Antoine Bosselut, Ronald Clark, Arman Cohan, Jakob Nicolaus Foerster, Yarin Gal, Scott A. Hale, Inioluwa Deborah Raji, Christopher Summerfield, Philip Torr, Cozmin Ududec, Luc Rocher, and Adam Mahdi. Measuring what matters: Construct validity in large language model benchmarks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=mdA5lVvNcU.

Betaflight Development Team. Betaflight: Open source flight controller firmware. https://github.com/betaflight/betaflight, 2024.

Christian Birchler, Nicolas Ganz, Sajad Khatiri, Alessio Gambi, and Sebastiano Panichella. Cost-effective simulation-based test selection in self-driving cars software. *Science of Computer Programming*, 226:102926, 2023. doi: 10.1016/j.scico.2023.102926.

Christian Birchler, Tanzil Kombarabettu Mohammed, Pooja Rani, Teodora Nechita, Timo Kehrer, and Sebastiano Panichella. How does simulation-based testing for self-driving cars match human perception? *Proceedings of the ACM on Software Engineering*, 1(FSE):929–950, 2024. doi: 10.1145/3643768.

Christian Birchler, Sajad Khatiri, Pooja Rani, Timo Kehrer, and Sebastiano Panichella. A roadmap for simulation-based testing of autonomous cyber-physical systems: Challenges and future direction. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–9, 2025. doi: 10.1145/3711906.

Timothé Boulet, Xavier Hinaut, and Clément Moulin-Frier. Software engineering agents for embodied controller generation: A study in Minigrid environments. In *NeurIPS 2025 Workshop on Efficient Reasoning*, 2025. URL https://arxiv.org/abs/2510.21902.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.07095.

Yuntianyi Chen, Yuqi Huai, Yirui He, Shilong Li, Changnam Hong, Qi Alfred Chen, and Joshua Garcia. A comprehensive study of bug-fix patterns in autonomous driving systems. *Proceedings of the ACM on Software Engineering*, 2(FSE):380–402, 2025. doi: 10.1145/3715733.

comma.ai. openpilot: An operating system for robotics. https://github.com/commaai/openpilot, 2016. Open-source advanced driver-assistance system.

Fan Cui, Hongyuan Hou, Zizhang Luo, Chenyun Yin, and Yun Liang. HWE-bench: Benchmarking LLM agents on real-world hardware bug repair tasks. arXiv preprint arXiv:2604.14709, 2026. URL https://arxiv.org/abs/2604.14709.

DeepSeek. DeepSeek-V4-Pro-0813. https://api-docs.deepseek.com/quick_start/pricing/, 2026. Model version documentation; accessed 2026-09-03.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2509.16941.

Andrea Di Sorbo, Fiorella Zampetti, Corrado Aaron Visaggio, Massimiliano Di Penta, and Sebastiano Panichella. Automated identification and qualitative characterization of safety concerns reported in UAV software platforms. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–37, 2023. doi: 10.1145/3564821.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, volume 78 of *Proceedings of Machine Learning Research*, pp. 1–16. PMLR, 2017. URL https://arxiv.org/abs/1711.03938.

Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, et al. AInsteinBench: Benchmarking coding agents on scientific repositories. arXiv preprint arXiv:2512.21373, 2025. URL https://arxiv.org/abs/2512.21373.

Shiwei Feng, Yapeng Ye, Qingkai Shi, Zhiyuan Cheng, Xiangzhe Xu, Siyuan Cheng, Hongjun Choi, and Xiangyu Zhang. ROCAS: Root cause analysis of autonomous driving accidents via cyber-physical co-mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1620–1632. ACM, 2024. doi: 10.1145/3691620.3695530.

Jarett Forzano, Omar Almatov, Langston Nashold, Nikil Ravi, and Orestes Kassian. Valkyrie: A microservice-based framework for scalable evaluation of AI agents. In *Proceedings of the ACM Conference on AI and Agentic Systems (CAIS)*, pp. 1270–1274. ACM, May 2026. doi: 10.1145/3786335.3813231. URL https://doi.org/10.1145/3786335.3813231.

Max Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, et al. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2603.22435.

Gemini Robotics Team, Abbas Abdolmaleki, Saminda Abeyruwan, et al. Gemini robotics 1.5: Pushing the frontier of generalist robots with advanced embodied reasoning, thinking, and motion transfer. *arXiv preprint arXiv:2510.03342*, 2025. doi: 10.48550/arXiv.2510.03342. URL https://arxiv.org/abs/2510.03342.

Google DeepMind. Gemini 3.7 Flash model card. https://deepmind.google/models/model-cards/gemini-3-7-flash/, August 2026. Accessed 2026-09-03.

Edward R. Griffor, Christopher Greer, David A. Wollman, and Martin J. Burns. Framework for cyber-physical systems: Volume 1, overview. Technical Report NIST SP 1500-201, National Institute of Standards and Technology, June 2017. URL https://doi.org/10.6028/NIST.SP.1500-201.

Kalea Hall and Abhirup Roy. GM to use NVIDIA AI chips, software to automate vehicles, factories. Reuters, March 2025. URL https://www.channelnewsasia.com/business/gm-use-nvidia-ai-chips-software-automate-vehicles-factories-5008871. Published March 18, 2025; accessed August 16, 2026.

Yingjie Jiang, Ran Mo, Wenjing Zhan, Dongyu Wang, Zengyang Li, and Yutao Ma. Leveraging modular architecture for bug characterization and analysis in automated driving software. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–31, 2025. doi: 10.1145/3707455.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770.

Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 281–292. IEEE, 2023. doi: 10.1109/ICST57152.2023.00034.

Sajad Khatiri, Andrea Di Sorbo, Fiorella Zampetti, Corrado Aaron Visaggio, Massimiliano Di Penta, and Sebastiano Panichella. Identifying safety–critical concerns in unmanned aerial vehicle software platforms with SALIENT. *SoftwareX*, 27:101748, 2024a. doi: 10.1016/j.softx.2024.101748.

Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based testing of unmanned aerial vehicles with Aerialist. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 134–138. ACM, 2024b. doi: 10.1145/3639478.3640031.

Sajad Khatiri, Fatemeh Mohammadi Amin, Sebastiano Panichella, and Paolo Tonella. When uncertainty leads to unsafety: Empirical insights into the role of uncertainty in unmanned aerial vehicle safety. *Empirical Software Engineering*, 30(6):166, 2025. doi: 10.1007/s10664-025-10697-z.

Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pp. 2149–2154. IEEE, 2004. doi: 10.1109/IROS.2004.1389727.

Zixing Lei, Genjia Liu, Yuanshuo Zhang, Qipeng Liu, Chuan Wen, Shanghang Zhang, et al. EmboCoach-Bench: Benchmarking AI agents on developing embodied robots. *arXiv preprint arXiv:2601.21570*, 2026. URL https://arxiv.org/abs/2601.21570v1.

Quanyi Li, Zhenghao Peng, Lan Feng, Qihang Zhang, Zhenghai Xue, and Bolei Zhou. MetaDrive: Composing diverse driving scenarios for generalizable reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(3):3461–3475, 2023. doi: 10.1109/TPAMI.2022.3190471. URL https://arxiv.org/abs/2109.12674.

Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. An empirical study on failures in automated issue solving. *arXiv preprint arXiv:2509.13941*, 2025. URL https://arxiv.org/abs/2509.13941.

Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. doi: 10.1109/IROS45743.2020.9341207.

Lorenz Meier, Dominik Honegger, and Marc Pollefeys. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6235–6240. IEEE, 2015. doi: 10.1109/ICRA.2015.7140074.

Mike A. Merrill, Alexander G. Shaw, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Meta AI Research. Introducing Muse Code and Muse Spark 1.2. https://research.meta.ai/blog/introducing-muse-code-and-muse-spark-1-2, August 2026. Accessed 2026-09-03.

Olivier Michel. Cyberbotics ltd. Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004. doi: 10.5772/5618.

Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. ORBIT: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023. doi: 10.1109/LRA.2023.3270034. URL https://arxiv.org/abs/2301.04195.

Moonshot AI. Kimi K3 model card. https://huggingface.co/moonshotai/Kimi-K3, July 2026. Accessed 2026-09-03.

MoveIt Development Team. MoveIt 2: Motion planning framework for ROS 2. https://github.com/moveit/moveit2, 2022.

NVIDIA, Niket Agarwal, Arslan Ali, et al. Cosmos world foundation model platform for physical AI. *arXiv preprint arXiv:2501.03575*, 2025. doi: 10.48550/arXiv.2501.03575.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. OpenAI blog, August 2024.

OpenAI. GPT-5.6 Sol model. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026. Accessed 2026-09-01.

Xuezhi Song, Yi Li, Zhen Dong, Shuning Liu, Junming Cao, and Xin Peng. An empirical study on fault diagnosis in robotic systems. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 207–219, 2023. doi: 10.1109/ICSME58846.2023.00030. URL https://zhendong2050.github.io/res/ICSME23.pdf.

SpaceXAI. Introducing Grok 4.6. https://x.ai/news/grok-4-6, August 2026. Accessed 2026-09-03.

Akash Sriram and Kenrick Cai. Google introduces new AI models for rapidly growing robotics industry. Reuters, March 2025. URL https://www.reuters.com/technology/google-introduces-new-ai-models-rapidly-growing-robotics-industry-2025-03-12/. Published March 12, 2025; accessed August 16, 2026.

SWE-agent Team. mini-SWE-agent: The minimal AI software engineering agent. https://github.com/SWE-agent/mini-swe-agent, 2025. Version 2.4.6 used in our experiments; accessed 2026-09-03.

Shuncheng Tang, Zhenya Zhang, Yi Zhang, Jixiang Zhou, Yan Guo, Shuang Liu, et al. A survey on automated driving system testing: Landscapes and trends. *ACM Transactions on Software Engineering and Methodology*, 32(5), 2023.

Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, et al. Demonstrating GPU parallelized robot simulation and rendering for generalizable embodied AI with ManiSkill3. In *Robotics: Science and Systems (RSS)*, 2025. doi: 10.15607/RSS.2025.XXI.021.

Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics. https://drake.mit.edu, 2019.

Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wąsowski. ROBUST: 221 bugs in the robot operating system. *Empirical Software Engineering*, 29(3):57, 2024. doi: 10.1007/s10664-024-10440-0.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30. IEEE, 2017. doi: 10.1109/IROS.2017.8202133. URL https://arxiv.org/abs/1703.06907.

Imara van Dinten, Pouria Derakhshanfar, Annibale Panichella, and Andy Zaidman. The slow and the furious? performance antipattern detection in cyber–physical systems. *Journal of Systems and Software*, 210:111904, 2024. doi: 10.1016/j.jss.2023.111904. URL https://repository.tudelft.nl/file/File_3bb6d98f-13f4-4146-a2ac-dbb3bb4e102c.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 169–181, 2026. doi: 10.1145/3744916.3764576. URL https://arxiv.org/abs/2503.15223.

Zhipeng Xu, Jiahao Lu, Yining Zheng, Yuxin Wang, and Xipeng Qiu. SWE-bench science: Can coding agents resolve engineering tasks in science?, 2026. URL https://arxiv.org/abs/2608.19799v1. Version 1.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1601.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.03859.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-Bench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, 2025. doi: 10.18653/v1/2025.acl-long.189. URL https://arxiv.org/abs/2506.09289.

Z.ai. GLM-5.3 model card. https://huggingface.co/zai-org/GLM-5.3, 2026. Accessed 2026-09-03.

Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. An empirical characterization of software bugs in open-source cyber–physical systems. *Journal of Systems and Software*, 192:111425, 2022. doi: 10.1016/j.jss.2022.111425. URL https://mdipenta.github.io/files/CPS_Bugs_study.pdf.

Daoguang Zan, Zhirong Huang, Wei Liu, et al. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2504.02605.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing best practices in building rigorous agentic benchmarks. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, volume 38, 2025. URL https://arxiv.org/abs/2507.02825.

# APPENDIX

The appendix is organized as follows.

- Appendix A: Detailed task curation, reproducer construction, verification, and release pipeline (Section 3.2).
- Appendix B: Scoring details, proxy-set comparisons, scaffold configurations, and recorded logs (Sections 3.3 and 4.3).
- Appendix C: Complete task inputs, shared agent prompts, and eight capability case studies with patch analyses (Section 4.2).
- Appendix D: Construct validity checklist.
- Appendix E: Agentic benchmark checklist.
- Appendix F: Repository descriptions, domain roles, and validation backends (Section 3.1).

## A CONSTRUCTION DETAILS

The four stages below expand the construction pipeline in Section 3.2.

### A.1 SELECT CPS TASKS

We collect merged pull requests with linked issues and recover their changes against the historical base revision. Test changes are separated from the fix patch using file-path heuristics. Empty fixes are discarded, and configurable patch-size limits bound environment reconstruction and validation cost.

A classification agent identifies CPS behavior changes, separating them from ordinary software maintenance, documentation, refactoring, and test-only edits. Candidate selection considers whether the affected behavior can be exercised with a clear observable, feasible execution cost, and reproducible conditions. The proposed validation route may use a component test, replay, or simulation; admission depends on executable verification.

### A.2 BUILD REPRODUCERS

A construction agent receives the issue, maintainer patch, and repository-specific instructions. It builds a testbench that exercises the triggering conditions and checks observable outputs or state changes. Scenarios specify the relevant inputs, units, tolerances, deadlines, horizons, and seeds. Acceptance depends on behavior rather than textual agreement with the reference patch.

Historical environments use repository-specific build and dependency preparation. The prepared buggy baseline may include compatibility changes needed to execute the historical revision. Evaluation then applies the candidate patch to that baseline.

Validation uses the affected interface: PX4 supports compiled tests or SITL with a MAVLink driver; Webots uses worlds and controllers; MetaDrive uses headless resets, steps, and sensor operations. Navigation2 supports targeted ROS builds and component tests, Drake uses Python interfaces or Bazel targets, and Betaflight can run host-side mixer tests with mocked hardware. The build target, driver, and assertions define each task's execution scope.

### A.3 VERIFY BEHAVIOR

A separate verifier executes the testbench on the base and maintainer-fixed versions. Reproducer repairs are followed by execution on both versions again. Admission requires at least one reported check, all declared checks to pass with the maintainer fix, and at least one baseline check to be non-passing. The implementation distinguishes explicit failures, skips, and absent baseline results; each may become passing after the repair. Failed behavioral regression checks invalidate the result when that check group is present.

This establishes base-to-fix separation under the declared scenarios. Broader validity criteria are addressed in Appendices D and E.

### A.4 Review and Release

Task statements describe the observed and expected behavior, affected component, and triggering conditions. A separate review checks clarity, consistency with the tested requirement, and solution leakage. The release uses a valid reviewed statement when available, otherwise a valid generated statement.

Agents receive the task statement and prepared buggy repository. Maintainer patches, hidden testbenches, and privileged evaluation outputs are withheld. The released task set is selected from verified candidates using repository and capability labels.

## B Evaluation Details

**Execution and scoring.** Repository tests (OL) and behavioral checks (CL) denote evaluation channels. Repository tests can be stateful, and behavioral checks may exercise a component without running a full feedback loop. Check membership is fixed using the base and maintainer-fixed versions before evaluating agent patches. An OL resolution verdict requires at least one fail-to-pass test; regression guards alone do not establish a repair.

The task-level criterion and resolve rate follow Section 3.3. Patch-induced application, build, or execution failures count as unresolved. A submitted patch that passes evaluation is resolved even if the agent reached its interaction limit. An attempt without an evaluation verdict counts as unresolved. Capability rates use primary labels, so each task is counted once in the overall score.

**Proxy set.** The public proxy set is fixed at 40 tasks, five per capability. We select the four models using their mini-SWE-agent ranking on the full 400-task release. All four models are compared on the same 40 tasks under both scaffolds. The comparison describes this proxy set rather than full-release performance.

**Scaffold configuration.** Mini-SWE-agent executes text-delimited Bash actions. OpenHands exposes `execute_bash`, `str_replace_editor`, `think`, `task_tracker`, and `finish`, with a persistent Bash session. Muse Spark 1.2 uses native tool calls in 39 of its 40 runs; Opus 5, Grok 4.6, and GPT5.6-Sol use text-parsed calls. The comparison is between these configurations as run: prompts, tools, transport, and execution management vary together.

**Qualitative cases and checks.** The paired cases hold the model and task fixed and illustrate opposite scaffold outcomes. Opus 5 / PX4 #17795 is outside the proxy set; GPT5.6-Sol / Betaflight #3882 is inside it. PASS and NOT PASS are task-level evaluator outcomes; numbered checks identify requirements rather than separately observed verdicts. Reasoning interprets submitted patches and logs (Appendix B.1).

For PX4 #17795, C1 builds and flies headless SITL with a 60 m warning-only fence and 10 m/s outward cruise. C2 requires a warning with `GF_PREDICT=0`; C3 requires its first occurrence at distance $\geq$45 m from home. No warning is a failure. The task also requires preserving default prediction, but that setting is not separately exercised. OpenHands' compile return establishes local compilation, not agent execution of the hidden flight scenario.

For Betaflight #3882, C1 executes the real QuadX mixer with the craft armed, throttle 1500, yaw PID input $+100$, zero roll/pitch input, and `yaw_motors_reversed=false`. C2 requires motor 1 below motors 0 and 3; C3 requires motor 2 below motors 0 and 3. Motors 1/2 have yaw factor $+1$, while motors 0/3 have factor $-1$. The enabled reversal option and full flight dynamics are not exercised.

### B.1 Recorded scaffold logs

M indexes mini-SWE-agent messages (zero-based); E indexes OpenHands events. Quoted agent statements are distinct from command returns and scored patches. We show selected log text with explicit omissions, normalized dashes, and line wrapping; headings are editorial.

### SAFETY: OPUS / PX4 #17795

With `GF_PREDICT=0`, the breach check must use the current position. Mini defines the parameter and claims a complete repair; OpenHands also connects the accessor and decision branch, then obtains a successful local compile return.

**mini-SWE-agent: claimed completion, parameter-only submission**

```
[M28 | recorded plan]
The fix (as upstream PX4 did) is to add a boolean `GF_PREDICT` geofence parameter
     and use it in `Navigator::geofence_breach_check` to fall back to the current
     position test.

[M50 | Bash command, excerpt]
cat <<'EOF' >> src/modules/navigator/geofence_params.c
[... parameter documentation omitted ...]
PARAM_DEFINE_INT32(GF_PREDICT, 1);
EOF

[M51 | command return, excerpt]
<returncode>0</returncode>
PARAM_DEFINE_INT32(GF_PREDICT, 1);

[M56 | recorded completion claim, excerpts]
My work is finished: the fix is implemented across the three firmware source
    files [...]
the navigator module compiles cleanly (exit code 0) [...]
```

The scored patch changes only `geofence_params.c`; neither the accessor nor the navigator branch is present. The evaluator verdict is **NOT PASS**.

**OpenHands: observed decision-path edit and compilation**

```
[E55 | file-editor replacement, added accessor]
bool getPredict() { return _param_gf_predict.get(); }

[E60 | file-editor return: navigator_main.cpp]
   815      if (_geofence.getPredict()) {
   816        fence_violation_test_point = _gf_breach_avoidance.
    getFenceViolationTestPoint();
   817
   818      } else {
   819        fence_violation_test_point = matrix::Vector2<double>(_global_pos.
    lat, _global_pos.lon);
   820        vertical_test_point_distance = 0.f;
   821      }

[E93 | executed Bash command]
bash /tmp/compile_navigator.sh 2>&1 | tail -30; echo "COMPILE_EXIT=${PIPESTATUS
    [0]}"
[E94 | command return]
COMPILE_EXIT=0

[E118 | final status/diff return, file list]
M src/modules/navigator/geofence.h
 M src/modules/navigator/geofence_params.c
 M src/modules/navigator/navigator_main.cpp
```

The three-file patch matches the scored submission and receives **PASS**. `COMPILE_EXIT=0` refers to compiling changed sources; it is not a flight simulation or the hidden evaluator verdict.

### CONTROL: SOL / BETAFLIGHT #3882

The yaw contribution must have the required sign at the motor mixer. Mini's rejected submission is followed by a persisted sign change. OpenHands records the correct repair in its plan, edits test files, and finishes with the mixer unchanged.

**mini-SWE-agent: rejection followed by a persisted source diff**

```
[M10 | recorded diagnosis, excerpt]
The targeted output confirms the defect: `scaledAxisPidYaw * currentMixer[i].yaw`
     is added, so positive yaw increases `+1` motors--the exact opposite of the
    required convention.

[M15 | submission-adapter return, excerpt]
SUBMISSION REJECTED: `git status --porcelain` is empty, so you have not modified
    any file. Reading the code is not a fix.

[M20 | executed Bash command]
cd /home/betaflight && git diff --no-color -- src/main/flight/mixer.c

[M21 | command return, source diff]
@@ -614,7 +614,7 @@ void mixTable(uint8_t vbatPidCompensation)
     for (int i = 0; i < motorCount; i++) {
         float mix =
             scaledAxisPidRoll  * currentMixer[i].roll +
-            scaledAxisPidPitch * currentMixer[i].pitch +
+            scaledAxisPidPitch * currentMixer[i].pitch -
             scaledAxisPidYaw   * currentMixer[i].yaw;

         mix *= vbatCompensationFactor;  // Add voltage compensation
```

M18 executes the guarded replacement; M19 reports `mixer.c` as modified. The diff returned at M21 matches the scored one-line repair, which receives **PASS**.

**OpenHands: pending correction and test-only submission**

```
[E25 | task-tracker arguments, selected entries]
[
  {
    "title": "Add or update a regression test for yaw contribution direction and
    reversal option",
    "status": "in_progress"
  },
  {
    "title": "Correct the yaw contribution sign in the motor mixer",
    "status": "todo"
  }
]

[E37 | file-editor arguments, selected fields]
{
  "command": "create",
  "path": "/home/betaflight/src/test/unit/flight_mixer_unittest.cc"
}

[E47 | finish arguments]
{
  "final_thought": "Unable to complete the repository changes within this session
    .",
  "thought": "to=task_tracker  ... Wait status test creation technically done
    only after compile? TDD test not run. Keep in progress. Edit first."
}
```

The scored patch changes `src/test/unit/flight_mixer_unittest.cc` and `src/test/Makefile`; it does not change `mixer.c`. The evaluator verdict is **NOT PASS**. The recorded plan alone does not establish an implemented correction or a successful test run.

## C Complete Task Inputs and Capability Case Studies

We provide one task per capability and analyze one evaluated model patch for each, using the same eight cases as Section 4.2. The presentation separates task inputs, code changes, and diagnosis, following the prompt and qualitative-case appendices of SWE-bench and SWE-agent (Jimenez et al., 2024; Yang et al., 2024). All cases use `mini-swe-agent`. Their identical system message and user-message template appear below once; each case supplies the complete, unabridged task text and

its exact system-information value. Substituting these two fields reconstructs the archived initial user message without omitted instructions. Printed dashes are typographically normalized; the archived input files preserve the original characters. The subsequent checks and gold patches are evaluator evidence, not agent inputs. Gold passes and the agent does not pass each recorded composite check; numbered requirements are not separately recorded outcomes. The diagnoses use submitted code and reference evidence rather than unrecorded agent telemetry.

**Case selection and evidence.** We select two unsuccessful, nonempty submissions from each of the four leading models, covering six repositories with at most two tasks from any repository. Each case has a passing reference repair, an unsuccessful agent outcome, and an identifiable defect in the submitted code; a different implementation alone is insufficient. We inspect patches and available execution evidence to connect requirements to implementation defects (Liu et al., 2025; Yu et al., 2025). Four attempts time out with partial repairs, so their diagnoses concern unfinished implementation rather than measured physical behavior. These examples do not estimate failure prevalence or establish model-wide capability deficits.

**Interpreting OL and CL evidence.** The case studies span build and interface integration, component behavior, state transitions, and full feedback execution. Component checks expose physical bounds and frame conventions; stateful scenarios exercise event propagation, sampling, and protection decisions. The analysis traces each submitted defect to the relevant behavioral requirement. Execution records establish the attempt outcome, while code comparison explains omitted or incorrect implementation. The cases illustrate repair mechanisms, without estimating a paired OL–CL gap.

**Reading the failure illustrations.** Each pair compares gold PASS with agent NOT PASS. Numbered boxes list acceptance criteria, and red outlines associate a source defect with the relevant criteria; they do not assign separately observed outcomes to individual assertions. Patch gutters mark additions with + and retained context with a dot, while line numbers refer to patched source files. Arrows mark wrapped lines, nonconsecutive numbers indicate omitted source, and shortened tab names identify files. Capability labels identify the task suite; the paragraph diagnosis names the failure mode.

SHARED SYSTEM MESSAGE

```
You are a helpful assistant that can interact with a computer.

Your response must contain exactly ONE bash code block with ONE command (or
    commands connected with && or ||).
Include a THOUGHT section before your command where you explain your reasoning
    process.
Format your response as shown in <format_example>.

<format_example>
Your reasoning and analysis here. Explain why you want to perform the action.

```mswea_bash_command
your_command_here
```
</format_example>

Failure to follow these rules will cause your response to be rejected.
```

SHARED USER-MESSAGE TEMPLATE

```
Please solve this issue: {TASK_DESCRIPTION}

You can execute bash commands and edit files to implement the necessary changes.

## Recommended Workflow

This workflow should be done step-by-step so that you can iterate on your changes
     and any possible problems.
```

````
1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust
6. Submit your changes and finish your work by issuing the following command: `
    echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`.
   Do not combine it with any other command. <important>After this command, you
    cannot continue working on this task.</important>
   <important>Before submitting, you MUST have modified at least one source file.
   Reading and analysing the code is not a fix: a submission with no file
   changes is graded exactly the same as a wrong fix. Check with `git status`
   or `git diff` that your edits are present. If you believe the reported
   behaviour needs no code change, say why in your THOUGHT before
   submitting.</important>

## Important Rules

1. Every response must contain exactly one action
2. The action must be enclosed in triple backticks
3. Directory or environment variable changes are not persistent. Every action is
    executed in a new subshell.
   However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/
    working/dir && ...` or write/load environment variables from files

<system_information>
{SYSTEM_INFORMATION}
</system_information>

## Formatting your response

Here is an example of a correct response:

<example_response>
THOUGHT: I need to understand the structure of the repository first. Let me check
     what files are in the current directory to get a better understanding of the
     codebase.

```mswea_bash_command
ls -la
```
</example_response>

## Useful command examples

### Create a new file:

```mswea_bash_command
cat <<'EOF' > newfile.py
import numpy as np
hello = "world"
print(hello)
EOF
```

### Edit files with sed:```mswea_bash_command
# Replace all occurrences
sed -i 's/old_string/new_string/g' filename.py

# Replace only first occurrence
sed -i 's/old_string/new_string/' filename.py

# Replace first occurrence on line 1
sed -i '1s/old_string/new_string/' filename.py

# Replace all occurrences in lines 1-10
sed -i '1,10s/old_string/new_string/g' filename.py
```

### View file content:
````
````

```
```mswea_bash_command
# View specific lines with numbers
nl -ba filename.py | sed -n '10,20p'
```

### Any other command you want to run

```mswea_bash_command
anything
```
```

### C.1 PHYSICAL DYNAMICS

**Task:** PX4 #22769 **Model:** Grok 4.6

**Complete task description**

> In a PX4 multicopter, manual horizontal stick input in Altitude mode (and Stabilized mode) is turned into a horizontal acceleration setpoint by the flight_mode_manager's stick−tilt utility, specifically the routine that generates acceleration setpoints from stick deflection (StickTiltXY:: generateAccelerationSetpoints). The magnitude of the commanded acceleration is scaled by the configured maximum manual tilt angle, exposed through the MPC_MAN_TILT_MAX parameter.
>
> Observed behavior: When MPC_MAN_TILT_MAX is set high (above roughly 70 degrees, and especially near 90 degrees), the horizontal acceleration setpoints become physically impossible. Even a tiny stick deflection produces enormous accelerations −−− for example, a 10% pitch stick input at MPC_MAN_TILT_MAX = 90 degrees yields horizontal accelerations on the order of millions of m/s ^2 (about 1e7). At tilt angles at or beyond 90 degrees the response can also invert, so the commanded acceleration points opposite to the stick. This makes the vehicle essentially unsteerable. The root cause is that the setpoint magnitude is scaled by the tangent of the tilt angle, which diverges to infinity as the angle approaches 90 degrees and goes negative past it.
>
> Expected behavior: For any MPC_MAN_TILT_MAX value, including 90 degrees, the horizontal acceleration setpoint produced from stick input must stay finite and within a physically reasonable bound for a multicopter (a few g, well under about 10 g). The mapping must remain well−behaved: larger stick deflection must command more acceleration than a small deflection, full stick must still command a non−negligible acceleration, and normal tilt configurations (e.g. 45 degrees) must be unaffected −−− a forward pitch stick there should still command a sane positive forward acceleration with no spurious lateral component.
>
> Trigger condition: MPC_MAN_TILT_MAX configured near or at 90 degrees while the vehicle is in Altitude (or Stabilized) mode and the pilot applies pitch/roll stick input.

**System-information field**

```
Linux 6.8.0-90-generic #91-Ubuntu SMP PREEMPT_DYNAMIC Tue Nov 18 14:14:30
    UTC 2025 x86_64
```

PHYSICAL DYNAMICS: PATCH ANALYSIS

**Task:** PX4 #22769 **Model:** Grok 4.6

**Evaluation requirements.**

- ☐ **C1.** 90°: finite 10%/full-stick outputs; full > small and >0.5 m/s$^2$.
- ☐ **C2.** Both norms <10g (dt=1 s; yaw and yaw setpoint=0).
- ☐ **C3.** 45°: finite; 0.2g < ax <3g; |ay| $\leq$0.1 m/s$^2$; compile/run tests.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/src/modules/flight_mode_manager/tasks/Utility/StickTiltXY.cpp
+++ b/src/modules/flight_mode_manager/tasks/Utility/StickTiltXY.cpp
@@ -49,5 +49,9 @@ Vector2f StickTiltXY::generateAccelerationSetpoints(Vector2f
    stick_xy, const flo
  _man_input_filter.setParameters(dt, _param_mc_man_tilt_tau.get());
  stick_xy = _man_input_filter.update(stick_xy);
  Sticks::rotateIntoHeadingFrameXY(stick_xy, yaw, yaw_setpoint);
- return stick_xy * tanf(math::radians(_param_mpc_man_tilt_max.get())) *
    CONSTANTS_ONE_G;
+ // Horizontal acceleration for a given tilt is g * tan(tilt). tan() diverges at
     90 deg
+ // and becomes negative beyond that, producing huge or inverted setpoints.
+ // Limit to 89 deg (same as MAX_SAFE_TILT_DEG) to keep the mapping finite and
    well-behaved.
+ const float tilt_deg = math::constrain(_param_mpc_man_tilt_max.get(), 0.f, 89.f
    );
+ return stick_xy * tanf(math::radians(tilt_deg)) * CONSTANTS_ONE_G;
 }
```

**Gold comparison.** `src/modules/flight_mode_manager/tasks/Utility/StickTiltXY.cpp`

```
_maximum_acceleration = math::constrain(
    tanf(maximum_tilt), .02f, 3.f) * CONSTANTS_ONE_G;
```

**Diagnosis.** Numerical stabilization can leave physical command limits violated. In PX4, Grok 4.6 caps the tilt input at $89°$ but retains an acceleration scale of roughly $57g$. The reference repair bounds the output scale at $3g$, while the component-level behavioral check allows responsive outputs below $10g$. The resulting calculation is finite but exceeds the physical bound, exposing a failure to carry the acceleration constraint through the full input-to-command transformation (Figure 4, left).

### C.2 Coordinate frames & units

**Task:** Betaflight #14849 **Model:** GPT-5.6 Sol

**Complete task description**

> Custom magnetometer (compass) alignment is applied in the wrong rotation direction. Betaflight lets a user orient an external magnetometer by choosing the CUSTOM alignment mode (align_mag = CUSTOM) and entering explicit roll/pitch/yaw angles (mag_align_roll, mag_align_pitch, mag_align_yaw). These angles are meant to rotate the raw magnetometer reading into the craft's body frame using the standard convention (positive angles = clockwise rotation, matching the fixed non−custom alignment presets).
>
> Observed behavior: when a custom alignment is configured, the entered angles rotate the compass reading in the reversed direction. A user who enters the angle that physically matches their sensor's mounting gets an incorrect heading, and must instead enter the mirror/opposite angle to obtain a correct reading. For example, a magnetometer that required mag_align_yaw = 1800 (180 degrees) to read correct North had to be reconfigured to mag_align_yaw = 0 to read correctly, i.e. the applied yaw came out inverted relative to what was entered. Concretely, a custom yaw of +90 degrees rotates a raw sample ( x, y, z) to (y, −x, z) instead of the correct (−y, x, z), and a custom yaw of +270 degrees produces (−y, x, z) instead of the correct (y, −x, z).
>
> Expected behavior: the custom alignment angles must be interpreted in the same forward/clockwise convention as the built−in alignment presets, so that a custom yaw of +90 degrees rotates (x, y, z) to (−y, x, z) and a custom yaw of +270 degrees rotates it to (y, −x, z). Entering the hardware−correct alignment angles should yield a correct compass heading without any sign inversion or manual compensation.
>
> Affected function: the compass/magnetometer sensor subsystem, specifically the code that builds and applies the custom alignment rotation for the magnetometer during compass initialization. On the

custom−angle path the rotation ends up applied in the reversed direction, so it disagrees with the fixed alignment presets.

Trigger condition: any configuration using align_mag = CUSTOM together with a non−zero custom alignment angle (roll, pitch, or yaw). The bug appears whenever a custom magnetometer alignment is active; it does not affect the fixed alignment presets.

**System-information field**

```
Linux 6.17.0-42-generic #42-Ubuntu SMP PREEMPT_DYNAMIC Thu Jul 23 19:56:28
    UTC 2026 x86_64
```

### COORDINATE FRAMES & UNITS: PATCH ANALYSIS

**Task:** Betaflight #14849 **Model:** GPT-5.6 Sol

**Evaluation requirements.**

- ☐ **C1.** Build the real compass API and mocked driver.
- ☐ **C2.** Raw (1,2,3): +90° → (−2,1,3); +270° → (2,−1,3); each axis within $10^{-3}$.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/src/main/sensors/compass.c
+++ b/src/main/sensors/compass.c
@@ -383,6 +383,7 @@ bool compassInit(void)
     }

     buildRotationMatrixFromAngles(&magDev.rotationMatrix, &compassConfig()->
    mag_customAlignment);
+    matrix33Transpose(&magDev.rotationMatrix);

     compassBiasEstimatorInit(&compassBiasEstimator, LAMBDA_MIN, P0);
```

**Gold comparison.** `src/main/sensors/compass.c`

```
magCustomAlignment.yaw = -magCustomAlignment.yaw;
buildRotationMatrixFromAngles(&magDev.rotationMatrix, &magCustomAlignment);
```

**Diagnosis.** Frame-convention repairs fail when the proposed transformation is disconnected from the repository's implemented interfaces. In Betaflight, Sol inserts a call to `matrix33Transpose`, which neither the base API nor the patch defines. The reference repair implements the required convention by negating the configured angles through the existing matrix API. The defect lies in translating a rotation operation into executable code, leaving the sensor-to-body conversion incomplete (Figure 4, right).

### C.3 TEMPORAL SYNCHRONIZATION

**Task:** Webots #494 **Model:** Claude Opus 5

**Complete task description**

In Webots, a Solid without a Physics node is kinematic: it collides with other bodies but is not itself moved by the dynamics engine. A Solid with a Physics node is dynamic and is simulated by the ODE physics engine. To save computation, the engine automatically puts a dynamic body to sleep (disables it) once it has come to rest and stayed inactive for a while (governed by the world's auto−disable / physicsDisableTime setting); a sleeping body is skipped by the solver until something wakes it.

Observed behavior: When a Supervisor controller changes the geometry of a kinematic Solid's boundingObject at runtime (for example resizing a Box, or editing the height values of an ElevationGrid), the collision shape used by the physics engine is not correctly re−evaluated against a neighbouring dynamic Solid that has already fallen asleep. The engine keeps treating the resting body as if the old (pre−change) bounding geometry were still in place: the sleeping dynamic Solid is never woken, so even though the new bounding geometry now overlaps or supports it, the dynamic Solid does not move and its reported position stays unchanged. In issue #409 this appeared with an ElevationGrid whose height field was updated through the Supervisor: the visual mesh updated but collision continued to use the stale bounding object. Notably, first nudging the kinematic Solid's position before applying the geometry change made the problem disappear, because that motion had already kept the neighbouring body awake.

Expected behavior: A runtime change to a kinematic Solid's bounding geometry must be taken into account by the collision detection immediately, even against dynamic Solids that are currently sleeping. When the changed geometry now intersects or supports such a resting body, the engine should wake that dynamic body so it reacts to the new shape (e.g. is pushed away or lifted) and its position updates accordingly.

Affected subsystem: the physics engine (ODE) collision handling, together with the code path that applies a runtime change of a Solid boundingObject's geometry or size (for example a resized Box or an edited ElevationGrid height field) to the underlying physics collision shape.

Trigger condition: a dynamic Solid is at rest and has been auto−disabled (asleep) while in contact with, or immediately adjacent to, a kinematic (physics−less) Solid; a Supervisor then modifies a size/geometry field of that kinematic Solid's boundingObject during simulation. The sleeping body is not re−enabled , so it fails to respond to the updated bounding geometry.

**System-information field**

```
Linux 6.8.0-87-generic #88-Ubuntu SMP PREEMPT_DYNAMIC Sat Oct 11 09:28:41
    UTC 2025 x86_64
```

TEMPORAL SYNCHRONIZATION: PATCH ANALYSIS

**Task:** Webots #494 **Model:** Claude Opus 5

**Evaluation requirements.**

- ☐ **C1.** Build/controller succeed; simulator exits 0 within 900 s.
- ☐ **C2.** Let body sleep 3 s on Box; resize height 0.2 → 0.6 m. After 2 s of stepping, require body height y >0.25 m.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/src/ode/ode/src/heightfield.cpp
+++ b/src/ode/ode/src/heightfield.cpp
@@ -879,6 +879,13 @@ void dGeomHeightfieldSetHeightfieldData( dGeomID g,
    dHeightfieldDataID d )
 {
     dxHeightfield* geom = (dxHeightfield*) g;
     geom->m_p_data = d;
+
+    // The height field data (and hence the collision shape) changed: the cached
+    // AABB of this geom is not valid anymore. It has to be invalidated so that
+    // the broad phase re-evaluates the collisions of this geom, otherwise the
+    // collision detection keeps using the previous (stale) bounding geometry.
+    geom->gflags |= ( GEOM_DIRTY | GEOM_AABB_BAD );
+    dGeomMoved( g );
 }

 dHeightfieldDataID dGeomHeightfieldGetHeightfieldData( dGeomID g )
```

**Gold comparison.** `src/webots/engine/WbSimulationCluster.cpp`

```
if (b2)
  dBodyEnable(b2);
```

**Diagnosis.** State propagation remains incomplete when geometry edits stop at local cache bookkeeping. In Webots, Opus 5 changes heightfield cache invalidation but leaves Box updates and sleeping-body activation unchanged. The reference repair wakes the affected bodies, and the scenario checks a body's position after resizing its support and advancing simulation. The patch changes one local representation while leaving the event-to-dynamics dependency unimplemented, disconnecting the geometry update from the physical response it is intended to trigger (Figure 5, left).

### C.4 Control & stability

**Task:** Nav2 #5207 **Model:** Grok 4.6

**Complete task description**

> In Nav2's Smac Hybrid−A∗ planner (nav2_smac_planner, SmacPlannerHybrid) using the Reeds−Shepp motion model, produced paths can contain unnecessary small reverse (backward) motions even when the goal is reachable with a purely forward path.
>
> Background: The Hybrid−A∗ search completes by attaching an "analytic expansion": once the search gets close enough to the goal, it tries to connect the current search node directly to the goal with an analytic Reeds−Shepp curve. A Reeds−Shepp curve is made of segments that can go forward or backward, so a "direction change" (cusp) is any point where the path switches between forward and reverse travel.
>
> Observed behavior: the analytic expansion that finishes the path can produce a connecting segment containing spurious small reverse motions and extra cusps, even when a clean forward−only connection to the goal exists. As a result the final path ends with short, unnecessary backward motions , appearing discontinuous and forcing the robot to reverse where straightforward forward motion would reach the goal.
>
> Expected behavior: when a goal is reachable without reversing, the planner should return a forward−only path. No spurious small reverse segments or extra forward/reverse cusps should be introduced on the approach to the goal.
>
> Affected component: the analytic−expansion path generation and selection in the Smac Hybrid−A∗ planner (nav2_smac_planner) for the Reeds−Shepp motion model.
>
> Trigger condition: planning with Smac Hybrid−A∗ configured for the Reeds−Shepp motion model, from a start pose to a goal pose that is reachable with forward−only motion in obstacle−free space; the resulting path contains avoidable reverse segments instead of remaining forward−only.

**System-information field**

```
Linux 6.8.0-90-generic #91-Ubuntu SMP PREEMPT_DYNAMIC Tue Nov 18 14:14:30
    UTC 2025 x86_64
```

#### Control & stability: patch analysis

**Task:** Nav2 #5207 **Model:** Grok 4.6

**Evaluation requirements.**

- ☐ **C1.** Build/run Smac planner on an empty 100×100 grid; start (50,50,0), 72 heading bins.
- ☐ **C2.** Plan to (30,70,0), (70,30,0), (30,30,0); total direction changes $\leq$1.

**Agent patch.** The decisive submitted changes are excerpted below; the complete patch is retained with the case evidence. *nav2_smac_planner/src/analytic_expansion.cpp*

```
    if (reverse_length > 1e-6 &&
      rs_path.length() > 1e-6 &&
      reverse_length < 0.5 * rs_path.length())
    {
      const double rho = static_cast<double>(d) / rs_path.length();
      auto dubins_space = std::make_shared<ompl::base::DubinsStateSpace>(rho);
      AnalyticExpansionNodes dubins_nodes =
        getAnalyticPath(node, goal, node_getter, dubins_space);
      if (!dubins_nodes.empty()) {
        return dubins_nodes;
      }
    }
```

**Gold comparison.** `nav2_smac_planner/src/analytic_expansion.cpp`

```
if (refined_analytic_nodes.direction_changes > analytic_nodes.direction_changes)
    {
```

`nav2_smac_planner/src/analytic_expansion.cpp`

```
continue;
}
```

**Diagnosis.** Constraints enforced during an initial planning step can be lost when later refinements use a different acceptance rule. In Nav2, Grok 4.6 conditions a forward-only connection on reverse-distance fraction, leaving later refinement free to introduce additional direction changes. The reference repair rejects refinements with more cusps, directly enforcing the property checked on the resulting plan. Replacing a bound on direction changes with a reverse-distance heuristic leaves the final plan's maneuver constraint unenforced (Figure 5, right).

### C.5 Sensing & perception

**Task:** MetaDrive #337 **Model:** GPT-5.6 Sol

**Complete task description**

> MetaDrive supports vision−based (image) observation, in which the driving environment is run with offscreen rendering and each observation returned by the environment includes a rendered camera image of the scene. When image observation is enabled on a headless/offscreen setup (no popup window), the image observation pipeline is broken: the environment cannot produce a usable rendered image.
>
> Observed behavior: With image observation turned on, running a short seeded closed−loop rollout of MetaDriveEnv (reset followed by a few steps) yields an image observation that is not a valid rendering of the scene. The returned image is degenerate −−− empty, wrongly shaped, or a constant/uniform buffer (e.g. all−black with essentially zero pixel variance) −−− rather than actual scene content. The routine that grabs the rendered pixel array for the image observation fails to capture the scene, so no real image content is populated.
>
> Expected behavior: With image observation enabled, both env.reset() and env.step() must return an observation dict that contains an 'image' entry holding a well−formed rendering: non−empty, at least a 3−dimensional array with height and width both greater than 1, containing only finite values, and showing real scene content (non−constant pixels, i.e. non−trivial variance). This must hold on a headless/offscreen configuration with the RGB camera as the image source and no traffic.
>
> Affected area: the offscreen image−observation rendering path −−− specifically the base camera routine that grabs the rendered pixel array used for the image observation (BaseCamera's image capture), which is broken so that the environment cannot obtain a valid image observation.

Trigger condition: constructing MetaDriveEnv with image observation enabled and use_render disabled ( offscreen mode) with the RGB camera selected as the image source, then calling reset() and step(); the returned image observation is unusable instead of being a proper scene rendering.

**System-information field**

```
Linux 6.8.0-86-generic #87-Ubuntu SMP PREEMPT_DYNAMIC Mon Sep 22 18:03:36
    UTC 2025 x86_64
```

SENSING & PERCEPTION: PATCH ANALYSIS

**Task:** MetaDrive #337 **Model:** GPT-5.6 Sol

**Evaluation requirements.**

- ☐ **C1.** Headless RGB, no traffic, seed 0; reset and 10 steps with action [0,1].
- ☐ **C2.** Image present; nonempty, ndim≥3, H,W>1, all finite; pixel std $>10^{-6}$.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/metadrive/component/vehicle_module/base_camera.py
+++ b/metadrive/component/vehicle_module/base_camera.py
@@ -112,6 +112,7 @@ class BaseCamera(ImageBuffer):

     def get_pixels_array(self, base_object, clip=True) -> np.ndarray:
         self.track(base_object)
+        self.engine.taskMgr.step()
         if self.enable_cuda:
             assert type(self)._singleton.cuda_rendered_result is not None
             ret = type(self)._singleton.cuda_rendered_result[..., :-1][...,
    ::-1][::-1]
```

**Gold comparison.** `metadrive/engine/core/image_buffer.py`

```
origin_img = self.buffer.getDisplayRegion(1).getScreenshot()
```

**Diagnosis.** Changes to data production leave a perception failure unresolved when the observation consumer still reads from the wrong source. In MetaDrive, Sol advances the rendering task manager but leaves the inherited screenshot call unchanged. The reference repair reads the final display region of the offscreen buffer, connecting rendered scene data to the observation returned after reset and stepping. The failure lies in the disconnect between producing sensor data and delivering it to downstream computation (Figure 6, left).

### C.6 DETERMINISM & REPRODUCIBILITY

**Task:** Drake #21623 **Model:** Muse Spark 1.2

**Complete task description**

Affected component: Drake's multibody subsystem, specifically the discrete−time MultibodyPlant and its dynamics−related output ports.

Background: A MultibodyPlant constructed with a positive time step is a discrete−time system. Its state is advanced by a periodic (or forced) discrete update, and it exposes several "dynamics" output ports whose values are derived from the forces and accelerations of a step: generalized_acceleration, body_spatial_accelerations, net_actuation, reaction_forces, and contact_results. The intended contract for a discrete system is that outputs behave as y[n] = f(t[n], x[n], u[n]): each dynamics output is a sample taken at the most recent completed discrete step and then held constant (zero−order hold) until the next step. In particular, external force/actuation inputs are meant to be sampled once per step rather than read continuously.

Observed behavior: The dynamics−related output ports of a discrete MultibodyPlant are not sampled. Instead they are recomputed "live" on demand as a continuous function of the current context −−− i.e. the plant behaves like a hybrid system y = f(t, x[n], u(t)) instead of a purely discrete one. The most visible symptom: evaluating a dynamics output port on a freshly created default context, before any discrete step has been taken, immediately triggers an on−demand forward−dynamics computation and returns a nonzero result. For example, for a plant containing a single body on one vertical prismatic ( sliding) joint under gravity, reading the generalized_acceleration port on a default context returns the gravitational acceleration (a nonzero vector) even though no step has occurred. There is no well− defined "most recent step" to sample yet, so the correct sampled value would be zero/empty.

(Background/motivation: the previous mechanism that emulated per−step input sampling depended on the systems−framework cache being enabled; when caching was disabled −−− which happens, e.g., during some initialization paths −−− the input ports were effectively read continuously and the discrete− sampling semantics silently degraded. The intended fix samples the step results into state so that this behavior no longer relies on caching.)

Expected behavior: For a discrete−time MultibodyPlant in its default output mode, the dynamics−related output ports must be genuine samples of the most recent discrete step, held constant between steps. Concretely, before any step has been taken the sample is empty and the ports read all−zero; after a discrete step has executed, the same ports read the values computed by that step (e.g. the generalized_acceleration port reports the gravitational acceleration for the prismatic−joint example above). Advancing the plant one step and re−reading the ports should reflect exactly one consistent set of step results.

Trigger condition: Build a discrete MultibodyPlant (nonzero time step) with at least one degree of freedom that experiences gravity (e.g. a single rigid body attached to the world by a vertical prismatic joint), finalize it, and create a default context. Read a dynamics output port such as generalized_acceleration ( a) before any discrete update, then (b) after forcing a discrete update. The buggy plant returns the nonzero live gravitational acceleration in case (a); the corrected, sampled plant returns zero in case (a) and the nonzero gravitational acceleration in case (b).

**System-information field**

```
Linux 6.8.0-90-generic #91-Ubuntu SMP PREEMPT_DYNAMIC Tue Nov 18 14:14:30
    UTC 2025 x86_64
```

DETERMINISM & REPRODUCIBILITY: PATCH ANALYSIS

**Task:** Drake #21623 **Model:** Muse Spark 1.2

**Evaluation requirements.**

- ☐ **C1.** Build/pass Bazel test: 0.01 s plant, vertical slider, gravity.
- ☐ **C2.** Before ExecuteForcedEvents: acceleration norm $\leq 10^{-9}$.
- ☐ **C3.** After ExecuteForcedEvents: acceleration norm $\geq 1$ m/s$^2$.

**Agent patch.** The decisive submitted changes are excerpted below; the complete patch is retained with the case evidence. *multibody/plant/multibody_plant.h*

```
void DeclareSampledOutputState();
```

*multibody/plant/multibody_plant.cc*

```
&MultibodyPlant<T>::CalcSampledOutput);
```

**Gold comparison.** `multibody/plant/multibody_plant.cc`

```
if (memory == nullptr) {
```

multibody/plant/multibody_plant.cc

```
    return *zero_acceleration_kinematics_placeholder_;
  }
  return memory->acceleration_kinematics_cache;
```

**Diagnosis.** Sampling semantics break when output readers remain coupled to live dynamics instead of the stored result of a completed step. In Drake, Muse Spark 1.2 calls undefined sampling methods and retains a live acceleration reader. The reference repair connects stored step results to output evaluation, making readout consistent with the completed simulation step. Partial changes to the sampling API leave the update-to-readout dependency incomplete and fail to implement the intended temporal behavior (Figure 6, right).

### C.7 Safety & fail-safe behavior

**Task:** PX4 #17795 **Model:** Claude Opus 5

**Complete task description**

Subsystem: the navigator's geofence breach check.

Observed behavior: The geofence breach check is always predictive. Instead of testing whether the vehicle's current position is inside the fence, the navigator projects the vehicle forward along its current velocity by an estimated braking distance (a function of horizontal speed and the vehicle's acceleration and jerk limits) and declares a breach when that projected future point leaves the fence. Because this look−ahead cannot be turned off, a fast−moving multirotor trips the geofence well inside the real fence boundary. With a small fence, or when the jerk limit is raised, the projected point can already lie outside the fence at low speed, so the geofence warning/failsafe fires far short of the actual boundary and can trigger almost immediately after takeoff. An operator who wants the vehicle to fly out to (and only slightly overshoot) the true fence has no way to get that behavior.

Expected behavior: There should be a way to disable the predictive look−ahead so the geofence is evaluated against the vehicle's actual current horizontal position and only trips when that current position reaches the configured fence boundary. This should be exposed as a boolean geofence parameter GF_PREDICT that remains enabled by default (preserving the existing predictive behavior); when it is set to 0 the breach check must fall back to the current−position test.

Trigger condition: An armed multirotor cruising horizontally outward toward the fence at a steady speed. With the fence action set to warning−only (so the flight path is not altered and velocity stays constant), and with the predictive behavior requested to be off, the geofence must first report the breach at approximately the true fence radius rather than tens of meters inside it. On the current code the request to disable prediction has no effect (no such control exists), so the breach still fires early based on the predicted point.

**System-information field**

```
Linux 6.8.0-101-generic #101-Ubuntu SMP PREEMPT_DYNAMIC Mon Feb  9 10:15:05
    UTC 2026 x86_64
```

#### Safety & fail-safe behavior: patch analysis

**Task:** PX4 #17795 **Model:** Claude Opus 5

**Evaluation requirements.**

- ☐ **C1.** Build/fly SITL: 60 m fence, 10 m/s, warning-only action.
- ☐ **C2.** With GF_PREDICT=0, observe a geofence warning.
- ☐ **C3.** First warning occurs at distance $\geq$45 m.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/src/modules/navigator/geofence_params.c
+++ b/src/modules/navigator/geofence_params.c
@@ -129,3 +129,20 @@ PARAM_DEFINE_FLOAT(GF_MAX_HOR_DIST, 0);
  * @group Geofence
  */
 PARAM_DEFINE_FLOAT(GF_MAX_VER_DIST, 0);
+
+/**
+ * Geofence predicted breach check
+ *
+ * If enabled, the geofence breach is not checked against the current vehicle
+ * position but against a position which is predicted based on the current
+ * velocity and the estimated braking distance. This allows to trigger the
+ * geofence action early enough such that the vehicle does not overshoot the
+ * fence boundary.
+ *
+ * If disabled, the geofence breach is only triggered once the current vehicle
+ * position violates the fence.
+ *
+ * @boolean
+ * @group Geofence
+ */
+PARAM_DEFINE_INT32(GF_PREDICT, 1);
```

**Gold comparison.** `src/modules/navigator/navigator_main.cpp`

```
fence_violation_test_point =
    matrix::Vector2d(_global_pos.lat, _global_pos.lon);
```

**Diagnosis.** Safety settings remain ineffective when parameter definitions are disconnected from the state-dependent protection logic. In PX4, Opus 5 declares `GF_PREDICT` but never reads it or changes the predicted-position check. The reference repair uses the setting to select current or predicted position, connecting configuration to the geofence warning decision during flight. The agent's patch leaves prediction unconditional and fails to implement the requested switch in protection behavior (Figure 7, left).

### C.8 Long-horizon robustness

**Task:** Drake #19920 **Model:** Muse Spark 1.2

**Complete task description**

> In Drake's multibody subsystem, a MultibodyPlant exposes a 'reaction_forces' output port that reports the spatial reaction force acting at each joint. These reaction forces are obtained as a side effect of the plant's dynamics: the plant reconstructs the set of forces that produced the current accelerations and feeds them, together with those accelerations, through inverse dynamics so that the residual per−joint force is the reaction force. For the result to be physically correct, the reconstructed force set must be exactly the set of forces the plant actually used to advance the state, so that the reported reactions balance the applied loads (i.e., agree with an independent free−body−diagram calculation).
>
> Observed behavior: When the plant is configured in discrete−time mode, the reaction−forces computation fails to account for all of the forces used during the discrete update. Contact and constraint contributions are dropped: in particular, hydroelastic contact forces are not included at all (they are only added in continuous mode), and other constraint−type forces such as joint−limit forces and general solver constraints are likewise not reflected. As a consequence, for a discrete plant that reaches a static or steady configuration under contact, the reaction forces reported on the port do not balance the applied loads and disagree with a free−body−diagram computation. This is most clearly seen with hydroelastic contact in discrete mode, where the missing contact contribution leaves the reported reactions wrong. The continuous−time mode does not exhibit this problem.

Expected behavior: In discrete mode, the reaction forces reported on the port should account for every force the discrete solver applied to advance the state −−− including force elements, joint−limit forces, general constraints, and both point and hydroelastic contact −−− regardless of which discrete contact solver is in use (SAP or TAMSI). The reported joint reaction forces should then balance the applied loads and match an independent free−body−diagram calculation, matching the correctness already achieved in continuous mode.

Affected function: MultibodyPlant's reaction−forces output−port computation (CalcReactionForces) in the multibody subsystem, together with the discrete−update path that determines which forces are aggregated for a discrete step.

Trigger condition: Simulate a discrete−time MultibodyPlant that comes to rest (or steady state) while in contact −−− especially hydroelastic contact −−− and read the 'reaction_forces' output port. The reported reactions are incorrect because contact/constraint forces from the discrete update are omitted.

**System-information field**

```
Linux 6.17.0-42-generic #42-Ubuntu SMP PREEMPT_DYNAMIC Thu Jul 23 19:56:28
    UTC 2026 x86_64
```

LONG-HORIZON ROBUSTNESS: PATCH ANALYSIS

**Task:** Drake #19920 **Model:** Muse Spark 1.2

**Evaluation requirements.**

- ☐ **C1.** Uncached Bazel builds/exits 0; rod/box regressions pass.
- ☐ **C2.** 5 s ladder force/torque vs free-body solution: rel. tol. $10^{-9}$.
- ☐ **C3.** Continuous + SAP/TAMSI; point/hydroelastic; welds/joint limits.

**Agent patch.** All diff hunks are shown; Git index metadata is omitted.

```
--- a/multibody/plant/discrete_update_manager.h
+++ b/multibody/plant/discrete_update_manager.h
@@ -160,6 +160,15 @@ class DiscreteUpdateManager : public
    ScalarConvertibleComponent<T> {
                            bool include_joint_limit_penalty_forces,
                            MultibodyForces<T>* forces) const;

+  // Adds discrete contact and constraint forces to `forces`. For SAP, this
+  // includes point and hydroelastic contact, joint limits, and other
+  // constraints. For TAMSI, this includes point and hydroelastic contact.
+  void AddDiscreteForces(const drake::systems::Context<T>& context,
+                         MultibodyForces<T>* forces) const {
+    DRAKE_DEMAND(forces != nullptr);
+    DoAddDiscreteForces(context, forces);
+  }
+
   // TODO(amcastro-tri): Consider replacing with more specific APIs with the
   // resolution of #16955. E.g., APIs to obtain generalized forces due to
   // constraints, rather than raw solver results.
@@ -284,6 +293,9 @@ class DiscreteUpdateManager : public
    ScalarConvertibleComponent<T> {
       const systems::Context<T>& context,
       ContactResults<T>* contact_results) const = 0;

+  virtual void DoAddDiscreteForces(const systems::Context<T>& context,
+                                   MultibodyForces<T>* forces) const = 0;
+
   // Struct used to conglomerate the indexes of cache entries declared by the
   // manager.
   struct CacheIndexes {
```

**Gold comparison.** `multibody/plant/multibody_plant.cc`

```
discrete_update_manager_->
    EvalDiscreteUpdateMultibodyForces(context);
```

**Diagnosis.** Incomplete solver-to-output integration leaves sustained physical invariants unimplemented in the reported quantities. In Drake, Muse Spark 1.2 adds a pure virtual force-aggregation method without implementing it in the contact manager or connecting it to reaction-force output. The reference repair connects solver forces to reaction-force output, completing the computation checked by the steady-state force-balance regression. The agent's changes stop at the interface declaration, leaving the physical quantity reported to downstream components unimplemented (Figure 7, right).

## D Construct Validity Checklist

We use the authors' checklist template (Bean et al., 2025); each item is followed by a concise response.

**Define the phenomenon**

- ☐ Provide a precise and operational definition for the phenomenon being measured
  **Yes.** Sections 3.1 and 3.3 define repository repair and execution-based task success.
- ☐ Specify the scope of the phenomenon being covered and acknowledge any excluded aspects
  **Yes.** The Ethics Statement and Appendix B delimit software-only evaluation and the execution scope of behavioral checks.
- ☐ Identify if the phenomenon has sub-components and ensure they are measured separately
  **Yes.** Table 1 reports the overall score and all eight capability scores.

**Measure only the phenomenon**

- ☐ Control for unrelated tasks that may affect the results
  **Partial.** Task curation excludes unrelated changes; the reported evaluation focuses on CPS repairs.
- ☐ Assess the impact of format constraints on model performance
  **No.** All leaderboard runs use the fixed response format in Appendix C.
- ☐ Validate any automated output parsing techniques for accuracy, consistency and bias
  **Partial.** Base/gold execution checks verify verdict separation; validation is based on reference outcomes.

**Construct a representative dataset for the task**

- ☐ Employ sampling strategies to ensure task items are representative of the overall task space
  **Partial.** Tasks cover several repositories and capabilities, but reconstructability introduces selection bias.
- ☐ Verify the quality and relevance of all task items, especially for large or automatically generated datasets
  **Partial.** Task verification uses base/gold executions and structured statement review.
- ☐ Include task items that test known LLM sensitivities (e.g. input permutations or variations)
  **No.** The evaluation uses the original task statements and scenario inputs.

**Acknowledge limitations of reusing datasets**

- ☐ Document whether the benchmark adapts a previous dataset or benchmark
  **Yes.** We document the adaptation of the issue–patch benchmark format in Sections 3.1 and 3.2.

- ☐ If so, analyse and report the relevant strengths and limitations of the adapted prior work
  **Yes.** Section 2 discusses repository-test strengths and their limits for CPS behavior.
- ☐ If so, report and compare performance on the new benchmark against the original
  **No.** The experiments evaluate the new CPS task collection.
- ☐ Explain modifications to reused datasets and how they improve construct validity
  **Yes.** Section 3.3 explains the behavioral checks added to repository-level evaluation.

**Prepare for contamination**

- ☐ Implement tests to detect data contamination and apply them to the benchmark
  **No.** The study does not test model-training contamination.
- ☐ Maintain a held-out set of task items to facilitate ongoing, uncontaminated evaluation
  **Partial.** Evaluation assets are held out; source issues and patches originate in public repositories.
- ☐ Investigate the potential pre-exposure of benchmark source materials or similar data in common LLM training corpora
  **No.** Pretraining exposure is outside the reported evaluation.

**Use statistical methods to compare models**

- ☐ Report the benchmark's sample size and justify its statistical power
  **Partial.** The release has 400 tasks; suites range from 12 (Res.) to 74 (Frame) tasks, so scores for the smallest suites are less precise. Statistical power is not formally analyzed.
- ☐ Report uncertainty estimates for all primary scores to enable robust model comparisons
  **No.** Headline and capability scores are reported as point estimates.
- ☐ If using human raters, describe their demographics and mitigate potential demographic biases in rater recruitment and instructions
  **No.** The manuscript does not characterize reviewer demographics or recruitment.
- ☐ Use metrics that capture the inherent variability of any subjective labels, without relying on single-point aggregation or exact matching.
  **Partial.** Task success uses executable checks; capability annotations use one primary label per task.

**Conduct an error analysis**

- ☐ Conduct a qualitative and quantitative analysis of common failure modes
  **Partial.** Section 4.2 presents qualitative analysis of one case per capability.
- ☐ Investigate whether failure modes correlate with non-targeted phenomena (confounders) rather than the intended construct
  **Partial.** The case analysis identifies interface, integration, and behavioral defects in the submitted patches.
- ☐ If so, identify and discuss any potential scoring biases revealed in the error analysis
  **Yes.** Appendix C explains composite verdicts and the scope of evidence-based case selection.
- ☐ Conduct experiments or propose new directions to improve model scores on the benchmark
  **Yes.** Section 4.3 evaluates a second scaffold, and Section 4.2 identifies the repair failures that future agents should address.

**Justify construct validity**

- ☐ Justify the relevance of the benchmark for the phenomenon with real-world applications
  **Yes.** The benchmark uses engineering issues from deployed simulator and autonomy projects.
- ☐ Provide a clear rationale for the choice of tasks and metrics, connected to the operational definition of the phenomenon
  **Yes.** Section 3.3 ties executable requirements to task-level success and aggregate scores.

- ☐ Compare similarities and differences between the benchmark and existing evaluations of similar phenomena
  **Yes.** Section 2 compares the benchmark with repository repair and CPS evaluations.
- ☐ Discuss the limitations and design trade-offs of the benchmark concerning construct validity
  **Yes.** Section 2 discusses oracle validity; the contamination items above and the Ethics Statement address source exposure and hardware transfer.

## E AGENTIC BENCHMARK CHECKLIST

We follow the published ABC template (Zhu et al., 2025), retaining every item ID and grouping; item labels are abbreviated.

**Task validity**

- ☐ **T.1.** Tool versions specified.
  **Partial.** Section 4 identifies Valkyrie and Daytona; software versions are not fully specified.
- ☐ **T.2.** Required APIs available.
  **Partial.** Task tools execute in provisioned environments; model inference uses external APIs.
- ☐ **T.3.** API interruptions handled.
  **Partial.** Each model call is retried up to four times (Section 4); an attempt without an evaluation verdict counts as unresolved (Appendix B).
- ☐ **T.4.** State cleared between runs.
  **Yes.** Each attempt starts from the task environment and base revision.
- ☐ **T.5.** Ground truth isolated.
  **Yes.** The agent-facing task excludes the gold patch and hidden reproducer.
- ☐ **T.6.** Environment frozen.
  **Partial.** Evaluation uses fixed task revisions and prepared container environments.
- ☐ **T.7.** Annotations verified.
  **Partial.** Base/gold execution verifies reference behavior; the analysis uses the resulting task annotations.
- ☐ **T.8.** Tasks solvable.
  **Yes.** Admission requires the maintainer patch to pass the task reproducer.
- ☐ **T.9.** Oracle solver available.
  **Yes.** The evaluator can apply and execute the stored maintainer patch as a reference solution.
- ☐ **T.10.** Evaluation shortcuts excluded.
  **Partial.** Task isolation and reproducer review address solution leakage and implementation-specific checks.

**Outcome validity**

- ☐ **O.a.1.** Equivalent answers accepted.
  **N/A.** Patch success is not scored by whole-string matching.
- ☐ **O.a.2.** Redundant wording handled.
  **N/A.** Patch success is not scored by answer wording.
- ☐ **O.b.1.** Negation handled.
  **N/A.** The evaluator does not use substring answer matching.
- ☐ **O.b.2.** Answer enumeration resisted.
  **N/A.** Listing candidate answers cannot satisfy executable checks.
- ☐ **O.b.3.** Guessing prevented.
  **N/A.** No substring-based answer key is used.

- ☐ **O.c.1.** Judge reliability validated.
  **N/A.** Final patch verdicts do not use an LLM judge.
- ☐ **O.c.2.** Judge attacks resisted.
  **N/A.** Final patch verdicts do not use an LLM judge.
- ☐ **O.d.1.** Test correctness verified.
  **Partial.** Base/gold execution and statement review check alignment with the reported issue.
- ☐ **O.d.2.** Test quality measured.
  **No.** Test verification uses reference executions rather than coverage or mutation scores.
- ☐ **O.e.1.** Fuzzer edge cases covered.
  **N/A.** The reported protocol does not use randomized fuzz testing.
- ☐ **O.e.2.** Fuzzer inputs covered.
  **N/A.** The reported protocol does not use randomized fuzz testing.
- ☐ **O.e.3.** Fuzzer sensitivity checked.
  **N/A.** The reported protocol does not use randomized fuzz testing.
- ☐ **O.f.1.** Workflow branches covered.
  **Partial.** Reproducers exercise the triggering conditions specified by each task.
- ☐ **O.f.2.** Flaky verdicts controlled.
  **Partial.** Testbenches specify scenarios and numerical tolerances; the evaluation uses one attempt per task.
- ☐ **O.g.1.** Valid final states accepted.
  **Partial.** Oracles accept patches by executable requirements; testing of alternative correct repairs is not quantified.
- ☐ **O.g.2.** Unrelated state preserved.
  **Partial.** Declared P2P checks protect the existing behaviors they cover.
- ☐ **O.g.3.** Trivial state changes rejected.
  **Partial.** Base-to-fix verification rejects the unmodified baseline; the study does not systematically evaluate state-manipulation attacks.
- ☐ **O.h.1.** Answer format specified.
  **N/A.** The submission is a source patch, not a matched answer string.
- ☐ **O.h.2.** Random answers rejected.
  **N/A.** The evaluator does not score guessed textual answers.
- ☐ **O.I.1.** Metric exploitation checked.
  **Partial.** Hidden execution checks isolate scoring from agent-visible instructions; systematic reward hacking is outside this study.

**Benchmark reporting**

- ☐ **R.1.** Benchmark open-sourced.
  **Partial.** The public split will be released; the remaining tasks stay held out (Reproducibility Statement).
- ☐ **R.2.** Harness open-sourced.
  **Partial.** The toolbox, including the evaluation suite, will be released (Section 3.4 and Reproducibility Statement).
- ☐ **R.3.** Contamination mitigated.
  **Partial.** Hidden tests are separated from agent inputs; source-history exposure remains possible.
- ☐ **R.4.** Task updates planned.
  **Partial.** Versioned construction supports adding tasks and revising evaluation artifacts.
- ☐ **R.5.** Capabilities linked to outcomes.
  **Yes.** Sections 3.1 and 3.3 connect the suites to executable requirements.
- ☐ **R.6.** Evaluation subject specified.
  **Yes.** Sections 4 and 4.1 identify the evaluated model and scaffold combinations.

- ☐ **R.7.** Flaws mitigated.
  **Yes.** Section 3.2 describes base/gold checks, statement review, and task rejection.
- ☐ **R.8.** Flaw impact discussed.
  **Yes.** The Ethics Statement discusses imperfect oracles and overgeneralization from selected repositories and scenarios.
- ☐ **R.9.** Flaw impact quantified.
  **No.** The study does not estimate score corrections for oracle or annotation errors.
- ☐ **R.10.** Uncertainty reported.
  **No.** Leaderboard and scaffold results are single-attempt point estimates.
- ☐ **R.11.** Interpretation guidance provided.
  **Yes.** Appendices B and C specify the scope of score and case-study conclusions.
- ☐ **R.12.** Non-AI baselines reported.
  **Partial.** Maintainer patches provide verified human-written reference solutions.
- ☐ **R.13.** Trivial baselines reported.
  **Partial.** Admission checks the unmodified base against each task reproducer.

# F Repository Descriptions

The repositories in CPSE-Bench are grouped here by the embodied domain they serve: driving, robotics, and aerial autonomy. Within each domain a project plays one of two roles. A *simulator* models the world forward, so its faults live in the physics, rendering, sensing, and determinism machinery itself and are validated by inspecting the world state it produces. An *autonomy stack* consumes that world state to perceive, plan, and act, so its faults live in perception, planning, control, or state estimation and are validated by executing the stack against a simulator. We list each project's role and the simulation backend against which its tasks are reproduced.

## F.1 Driving

**CARLA (Dosovitskiy et al., 2017).** An Unreal Engine–based simulator for autonomous-driving research that exposes sensors, traffic, and vehicle dynamics through a programmatic client API. *Role:* simulator. *Backend:* CARLA/Unreal Engine.

**MetaDrive (Li et al., 2023).** A lightweight, compositional driving simulator built on Panda3D that supports procedurally generated scenarios and fast headless rollouts, making it well suited to deterministic, repeated-seed behavioral verification. *Role:* simulator. *Backend:* Panda3D.

**openpilot (comma.ai, 2016).** An open-source driver-assistance stack (perception, planning, and longitudinal/ lateral control) maintained by comma.ai. *Role:* autonomy stack. *Backend:* MetaDrive/CARLA.

## F.2 Robotics

**Webots (Michel, 2004).** A mature open-source robotics simulator with an ODE-based physics engine and a broad library of robot and sensor models. *Role:* simulator. *Backend:* Webots/ODE.

**Isaac Lab (Mittal et al., 2023).** NVIDIA's GPU-accelerated robot-learning framework layered on Isaac Sim, used for large-scale physics-based training and evaluation. *Role:* simulator. *Backend:* Isaac Sim.

**gz-sim (Koenig & Howard, 2004).** The modern Gazebo simulator, a widely used robotics simulation platform for sensor, actuator, and multi-body dynamics modeling. *Role:* simulator. *Backend:* Gazebo.

**Drake (Tedrake & the Drake Development Team, 2019).** A model-based toolbox for robotics with a high-fidelity multibody-dynamics engine and rigorous contact modeling, exposing programmatic simulation and state readback. *Role:* simulator. *Backend:* Drake.

**Navigation2 (Macenski et al., 2020).** The ROS 2 navigation stack covering planning, control, and recovery behaviors for mobile robots, validated by executing against a simulator. *Role:* autonomy stack. *Backend:* Gazebo.

**MoveIt 2 (MoveIt Development Team, 2022).** The standard ROS 2 manipulation-planning framework (motion planning, kinematics, collision checking) for robot arms, validated against a simulated robot. *Role:* autonomy stack. *Backend:* Gazebo.

**ManiSkill (Tao et al., 2025).** A GPU-accelerated manipulation simulation framework built on the SAPIEN physics engine, supporting large-scale, repeatable contact-rich rollouts. *Role:* simulator. *Backend:* SAPIEN.

### F.3 Aerial Autonomy

**PX4 (Meier et al., 2015).** An open-source flight-control stack for uncrewed aerial vehicles, validated in software-in-the-loop (SITL) simulation against Gazebo. *Role:* autonomy stack. *Backend:* SITL/Gazebo.

**ArduPilot (ArduPilot Development Team, 2009).** A long-standing open-source autopilot for aerial and ground vehicles, with its own software-in-the-loop simulation harness. *Role:* autonomy stack. *Backend:* SITL.

**Betaflight (Betaflight Development Team, 2024).** A widely deployed open-source flight-control firmware for multirotors, exercising attitude/rate control loops and validated in software-in-the-loop simulation. *Role:* autonomy stack. *Backend:* SITL.