PCBWorld: A Benchmark Environment for Engine-Grounded PCB Design Automation

Hyungseok Song*
LG AI Research
Seoul, Republic of Korea
hyungseok.song@lgresearch.ai

Junseok Park*
LG AI Research
Seoul, Republic of Korea
frank.park227@lgresearch.ai

Won-Seok Choi*
LG AI Research
Seoul, Republic of Korea
wonseok.choi@lgresearch.ai

Seohui Bae
LG AI Research
Seoul, Republic of Korea
seohui.bae@lgresearch.ai

Han-Seul Jeong
LG AI Research
Seoul, Republic of Korea
hanseul.jeong@lgresearch.ai

Youngjoon Park†
LG AI Research
Seoul, Republic of Korea
yj.park@lgresearch.ai

Soonyoung Lee†
LG AI Research
Seoul, Republic of Korea
soonyoung.lee@lgresearch.ai

Abstract

PCB routing is the task of connecting the nets of a board with copper traces under strict design rules, yet learning-based methods still lag behind rule-based routers. We introduce PCBWorld, an open-source engine-grounded PCB routing environment built on the KiCad EDA engine. As a human engineer does, agents in PCBWorld interactively route a board through the engine's native operations, using its Design Rule Check (DRC) feedback to keep the routing within the design rules. The environment supports both RL policies and tool-using LLM agents. Alongside the environment, PCBWorld-Bench provides three dataset families in KiCad's native board format (`.kicad_pcb`), covering two types of controllable synthetic instances and 679 real open-source boards. It scores any completed board with eight engine-checked evaluation metrics, regardless of the routing method. In our experiments, agents in PCBWorld consistently outperformed grid-action RL policies and open-loop LLM baselines, and an RL policy trained only on synthetic boards transferred zero-shot to real boards, approaching rule-based routers. These results position the engine-grounded, interactive approach of PCBWorld as a promising foundation for advancing the routing ability of both RL and LLM agents.

Keywords

Agentic AI, PCB Routing Benchmark, Reinforcement Learning

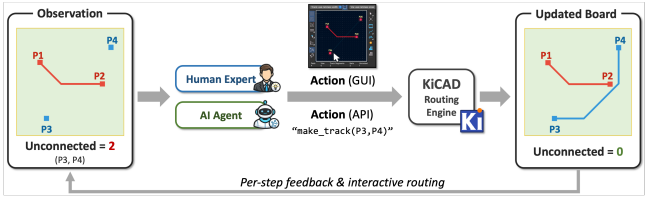
1 Introduction

Printed circuit boards (PCBs) are the physical boards that mount and interconnect electronic components, forming the backbone of nearly every electronic product [11, 27, 29]. Turning a circuit design into a manufacturable board centers on *routing*, the task of drawing copper traces that connect the pads of each net while keeping distinct nets electrically isolated under strict design rules [61]. As modern boards grow denser, routing every net without conflicts among tightly packed traces becomes increasingly challenging. Yet industry still relies mainly on rule-based routers rooted in decades-old heuristics [15, 35, 44]. They reliably route routine portions of a board, but rarely complete diverse, complex production boards end-to-end. Learning-based methods have been studied extensively to overcome this limit, but unlike in language, vision, and games, they are not yet competitive even with rule-based routers [5, 47].

We attribute this limitation to how the routing problem is modeled. Existing RL formulations cast routing as cell-by-cell movement on a discretized grid, whose search space grows rapidly with grid size [5, 26, 47], or delegate routing to an auto-router and restrict the agent to auxiliary decisions such as net ordering [40, 73, 74]. More recent LLM-based methods generate scripts or complete board files in a single open-loop pass [4, 66, 75]; without engine feedback, they struggle to satisfy the strict design rules. The only workflow that reliably completes production boards end-to-end remains that of a human engineer. An engineer neither leaves the whole board to an auto-router nor draws every trace entirely by hand. Instead, the engineer repeatedly reads the geometry of the current board, judges where the next connection can feasibly run, and draws it with the engine's native operations, which keep the new trace within the design rules.

To place agents in this same loop, we introduce PCBWorld, an *engine-grounded* PCB environment in which agents complete boards by directly invoking an EDA engine's native routing operations. Built on the open-source KiCad EDA engine [32, 62], PCBWorld exposes 58 Python APIs over the engine, ranging from the step-level routing operations an engineer uses to the engine's Design Rule Check (DRC). At every step, the agent receives the engine-updated board state, the admissible next actions, and DRC feedback, and routes the board one operation at a time through this interactive loop. Unlike GUI-driven agent environments [68], PCBWorld also provides headless and vectorized gym environments, supporting efficient large-scale RL training. On top of a standard Gym interface,

*Equal contribution.
†Corresponding author.



	Task	Geom.	DRC	API	RL	LLM
PCBWorld (ours)	PCB	Gridless	✓	✓	✓	✓
TREND [74]	Chip	Grid	✓	–	✓	–
XRoute-Env [73]	Chip	Grid	✓	✓	✓	–
Jumanji [5]	Abstract	Grid	–	–	✓	–
PCB-Bench [37]	PCB	–	–	–	–	✓

Figure 1: Overview of PCBWorld and PCBWorld-Bench. The KiCad PCB routing engine is exposed as the agent's interaction environment: agents invoke its native API and condition each step on engine-computed feedback (DRC status, connectivity, wirelength, via count); rewards and metrics are computed from these same engine-checked outcomes.

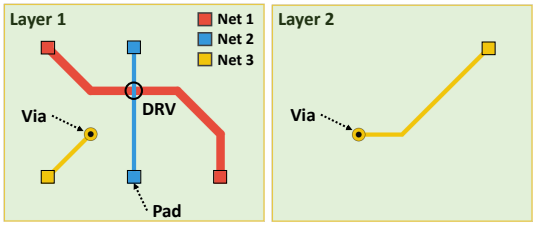


Figure 2: Core concepts of PCB routing. Pads belonging to the same net must be electrically connected; traces of different nets that come too close incur a design rule violation (DRV); and on multi-layer boards, vias allow traces to switch layers to avoid such conflicts.

PCBWorld provides tailored wrappers for RL policies and tool-using LLM agents, so both families of agents train and act in the same environment.

Alongside PCBWorld, we release PCBWorld-Bench, a benchmark that evaluates *geometric feasibility reasoning*, the capability to produce a routed board in which every net is connected and every trace satisfies the strict design rules. This capability remains largely untested in existing LLM benchmarks, which focus on software engineering [28], board interpretation [37], or 3D-CAD generation [64, 67]. PCBWorld-Bench combines two synthetic board generators, one grid-based (D1) and one gridless (D2), with a curated set of 679 real open-source boards (D3) derived from PCBench's open-source PCB corpus [22]. Every instance is provided in KiCad's native board format (`.kicad_pcb`) together with its own design rules, so it loads directly into PCBWorld for training and evaluation. A completed board is scored by the engine's DRC across 35 stock error-level design-rule checks, whose full severity mapping is detailed in Appendix F. Because the evaluation depends only on the completed board file, all routing methods, whether or not they route through PCBWorld, are scored the same way and compared fairly. The larger D3 boards, which reach hundreds of nets, remain a challenging target for routing research.

On PCBWorld-Bench, we evaluated three families of routing methods under the same protocol: rule-based routers (Freerouting [15], OrthoRoute [3], KiCadRoutingTools [19]), RL agents (PPO [58], GRPO [59], A2C [5], and Sable [47]), and LLM agents (the GPT-5.4 family [51, 52] and Qwen3.5-397B [57]). RL models trained on existing grid-routing benchmarks grow each trace one adjacent cell at a time; as the grid grew finer, even Sable, the state of the art on those benchmarks, collapsed, while PPO, which acts through segment-level KiCad-API actions, maintained its performance across all grid sizes. Trained only on simple synthetic boards, PPO transferred zero-shot to real boards, where it outperformed every other method, including the rule-based routers, on the small boards and remained second only to the strongest rule-based router on the larger ones. The LLM-agent results show that interactive routing substantially outperforms one-shot planning and track-level generation, consistently demonstrating the value of iterative interaction with the routing engine. Nevertheless, the slow inference time and poor scalability to complex boards reveal the difficulty of PCB routing for current LLMs, while leaving open a clear opportunity for future progress. We release PCBWorld and PCBWorld-Bench as open source. We hope they serve as a shared foundation for evaluating the practical routing ability of agents and for advancing learning-based PCB routing.

2 Background

2.1 PCB Routing Problem

A printed circuit board (PCB) is a multi-layer substrate whose components expose metal contacts called *pads*. A *net* is a set of pads that must be electrically connected to share the same signal, and PCB routing is the task of drawing copper *traces* that connect all pads of every net while keeping distinct nets electrically isolated. A trace is represented as a sequence of straight *track* segments, each lying on a single layer. A routing scheme is *grid-based* if track endpoints must lie on a fixed lattice, and *gridless* if they may take arbitrary positions on the canvas. A trace may contain the layer switching through a *via*, a plated hole that joins tracks across layers. A routed board is valid only if it passes a *Design Rule Check* (DRC); any *design rule violation* (DRV), for example traces of distinct nets running too close (a *clearance* violation) or intersecting, blocks fabrication. Such a DRV-free routing is *geometrically feasible*; a single violation invalidates an otherwise plausible board.

Among DRV-free routings, quality is commonly measured along two primary axes: total *wirelength*, which correlates with signal delay and material cost, and *via count*, which correlates with fabrication cost and reliability. For the routing state s, let $n_{\mathrm{drv}}(s)$, $\ell(s)$, and $n_{\mathrm{via}}(s)$ denote the number of DRVs, the total wirelength, and the via count, respectively. PCB routing task is then formulated as the constrained optimization problem:

$$s^\star = \arg\min_{s\in\mathcal{R}} \lambda_w \ell(s) + \lambda_v n_{\mathrm{via}}(s) \quad \text{subject to} \quad n_{\mathrm{drv}}(s) = 0, \quad (1)$$

where $\lambda_w, \lambda_v \geq 0$ are user-specified weights balancing wirelength against via count.

2.2 KiCad: An Open, Programmable EDA Suite

KiCad [32] is an open-source EDA suite maintained under the Linux Foundation [62], with sustained contributions from CERN [7, 13, 14]. It provides the end-to-end PCB design flow within a unified engine, encompassing component placement, trace routing, net editing, and design-rule checking [33]. The engine is written in C++ and exposes a programmable Python API through extensible bindings [30, 31], while its open, human-readable file format facilitates interoperability with other EDA tools. Together, these properties render KiCad's capabilities comparable to those of commercial EDA suites such as Cadence [6] and Altium [1]. The broader KiCad ecosystem also includes open PCB-routing resources such as PCBench [22] and plugins such as Freerouting [15], OrthoRoute [3], and KiCadRoutingTools [19].

3 PCBWorld: An Engine-Grounded PCB Routing Environment

PCBWorld wraps KiCad's routing engine as a Gym environment that faithfully reproduces a real-world PCB routing workflow. The environment is organized as three layers. The bottom layer is the C++ KiCad engine, which performs the actual computations, such as routing and design rule checks (§3.1). The middle layer is the Gym environment, which defines an MDP whose state, action, and reward are built from the engine's native operations (§3.2). On top, two tailored wrappers re-encode this MDP for RL and LLM agents (§3.3). More details are provided in Appendix J.

3.1 Python API via Bindings to the KiCad Engine

The bottom layer exposes the KiCad engine (§2.2) through a set of Python bindings. PCBWorld provides 58 *Python APIs* built on bindings to the engine: 14 reproduce the core routing API of `PNS::ROUTER`, 28 directly wrap KiCad's DRC and board-state extraction APIs, and 16 provide auxiliary utilities.[1] All higher layers interact with the engine exclusively through these headless Python APIs, free of GUI dependence and interactive-speed bottlenecks. Every agent action is therefore executed as a native KiCad engine operation, as in a human engineer's workflow (Figure 1). Because each environment holds its own engine instance, PCBWorld also runs as a vectorized environment, stepping many boards in parallel to sustain the rollout throughput that RL training requires.

3.2 MDP Formulation

The Gym environment formalizes the routing workflow as an MDP. An episode loads a real PCB file (`.kicad_pcb`) as its initial state, a *bare board* whose components are fixed and whose nets are unrouted. The agent then routes the board through successive API calls, observing the result of each, until every net is routed or a step limit is reached.

State. At each time step $t \in \{0, 1, \ldots, T\}$, the state s_t captures a snapshot of the board, where $t = 0$ corresponds to the bare board and $t = T$ to the terminal state at which the episode ends. A KiCad board is naturally described as a composition of heterogeneous geometric objects organized in a hierarchical structure. For example, each `pad` belongs to a `net` and carries its own `coordinate` (`net` → `pad` → `coordinate`). We introduce a **nested dictionary** that faithfully represents the exact numeric values and hierarchy of KiCad (Figure 3). Unlike abstracted representations such as a grid or a rendered image, this lets the agent observe the board's exact state, as the EDA domain's strict design rules demand. The dictionary splits by mutability during routing. **Board_static** holds immutable objects such as pad geometries, internal obstacles, and design rules. **Routing_geometry** holds objects added during routing, such as tracks, vias, and the per-pad connectivity status. Further details are provided in Appendix B.

Table 1: Routing actions.

Action	Arguments	Description
`net_select`	`net_id`	Select `net_id`; subsequent tracks and vias belong to the selected net.
`net_end`	–	Release the current net.
`start_route`	`p_start`	Begin a route from `p_start`, a pad center or existing track/via endpoint of the net.
`make_line`	`p_end`, `routing_mode`	Extend the route to `p_end` on the current layer. `routing_mode` sets how it interacts with existing traces.
`make_via`	`p_end`, `routing_mode`	Execute `make_line` to `p_end`, then place a via at `p_end`.
`finish`	`routing_mode`	Auto-complete the route to the Euclidean-nearest unconnected pad on the current layer.

Action. An action is a pair of an action type and its arguments, composed from the APIs exposed in §3.1. Table 1 summarizes the six action types. In the example of Figure 4, `make_line` takes arguments (`p_end`, `routing_mode`) and draws a track to `p_end`. The `routing_mode` argument controls how the engine realizes this track, either pushing blocking wires aside (`push_n_shove`) or detouring around them (`walkaround`), while satisfying strict design rules such as clearance. The KiCad engine exposes a finite set of candidate points for `p_end`, such as pad centers and the endpoints of existing tracks and vias. This reduces the action space from a continuous plane to a few meaningful points. The agent thus selects a `p_end` and a `routing_mode`, rather than generating primitive wire segments itself in the gridless space. The RL agent selects `p_end` from this candidate set. The LLM agent is prompted toward the same points but may emit free coordinates for detour waypoints (Appendix B).

Reward. The goal of the constrained optimization in Equation (1) is to reach a state s with low wirelength $\ell(s)$, low via count $n_{\text{via}}(s)$, and no DRVs, i.e., $n_{\text{drv}}(s) = 0$. We relax its hard DRV constraint into a penalty and define a potential function $\Phi(s)$ that measures the board quality,

$$\Phi(s) \;=\; -\big(f_d(n_{\text{drv}}(s)) + \lambda_w\, \ell(s) + \lambda_v\, n_{\text{via}}(s)\big).$$

The DRV count $n_{\text{drv}}(s)$ comes from KiCad's DRC API, which evaluates violations across the 35 stock error-level design-rule checks catalogued in Appendix F, including unconnected pads and clearance violations between distinct nets. The two weights $\lambda_w, \lambda_v \geq 0$

[1] The full list is provided in Appendix D. PCBWorld is built against KiCad 9.0.8 with `kicad-python` 0.6.0.

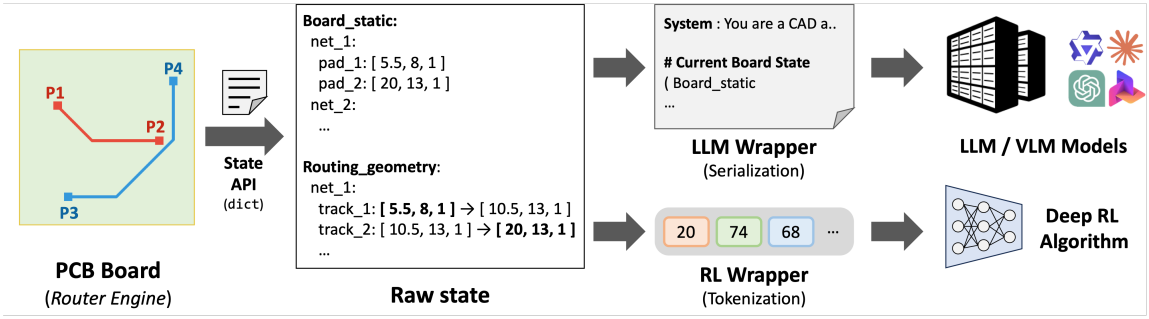


Figure 3: Two Gym wrappers communicate with the shared engine through a unified state dictionary, which they re-encode into agent-specific formats: tokenized sequences for the RL wrapper and serialized tool-call schemas for the LLM wrapper.

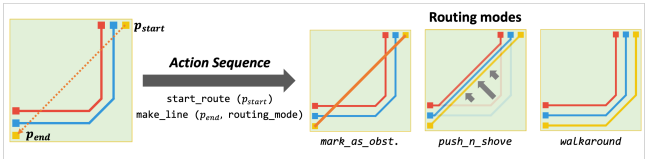


Figure 4: Example Action of PCBWorld. An example of `make_line`, where the `routing_mode` argument alters routing behavior through the KiCad routing engine, resulting in substantially different trace geometries across `walkaround`, `push_n_shove`, and `mark_as_obstacles`.

reuse those of Equation (1), and f_d is a monotone-increasing, concave penalty that assigns a large marginal cost to the first violation on each net. On PCBWorld-Bench this penalty dominates the typical wirelength and via savings between rollouts, so maximizing Φ drives the policy toward DRV-free routings that approximate the optimum of Equation (1).

The MDP's reward is the *terminal* reward, with $r_t = 0$ for $t < T$ and $r_T = \Phi(s_T) - \Phi(s_0)$ at termination, scoring the completed board against the bare board. For dense training we also expose a *per-step* form, $r_t = \Phi(s_{t+1}) - \Phi(s_t)$. With an undiscounted return ($\gamma = 1$), the per-step rewards telescope to $\sum_{t=0}^{T-1} r_t = \Phi(s_T) - \Phi(s_0)$, so the per-step form is a potential-based shaping of the terminal reward [49] and shares its optimal policy. In practice we train with $\gamma = 0.995$ (Appendix K.1), for which this equivalence holds approximately. Since GRPO [59] uses only the total return of each episode, the two reward forms coincide for it. PPO [58] works naturally with both forms. Because it estimates per-step advantages with a value function, the per-step form gives it a denser signal for credit assignment. PCBWorld exposes Φ as a pluggable objective rather than a fixed one. Since the equivalence above holds for any potential, a user can replace its quality terms with any criterion evaluable on the board state and learn the corresponding objective under either reward form.

3.3 Environment Wrappers

We implement separate wrappers that expose the nested-dictionary raw state and structured actions through interfaces tailored to RL and LLM tasks. For the LLM wrapper, we adopt an S-expression representation that preserves the nested hierarchy and key–value relations while minimizing token count, retaining only routing-relevant fields (coordinates, connectivity, layers, effective DRC constraints) and dropping internal ones. All numerical values are normalized to a uniform decimal precision, with special sentinels (e.g., the through-hole layer) rendered as distinct tokens to prevent confusion with integers. The prompt distills the task into a concise set of core guidelines—target identification, coordinate usage, obstacle avoidance, layer-change detours, and explicit net release—enabling efficient prompting. Finally, the wrapper supplies a compact action-history block with a "last rejected attempt" hint, so the agent recognizes no-effect or refused actions and avoids repeating them.

The RL wrapper tokenizes each geometric object (pad, track, via, point) into an individual token by combining a Fourier feature map of its coordinates with an entity-type embedding, yielding a flat sequence that preserves the native object hierarchy. The action is decoded autoregressively: the head emits an action type, then only the parameter tokens that type requires (e.g., a point selected by a pointer network from the engine-provided candidate set and/or a routing mode). This keeps every emitted action syntactically complete and within the engine's valid-action set by construction. Full tokenization, candidate-pool, and masking details are deferred to Appendix J.2.

4 PCBWorld-Bench

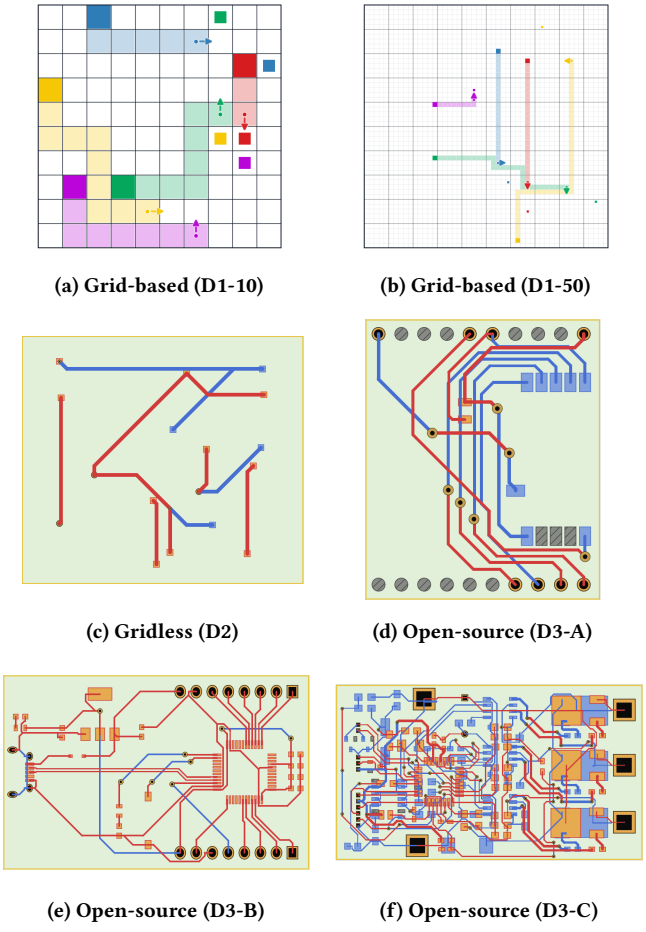
We introduce PCBWorld-Bench, a benchmark for geometric feasibility reasoning in PCB routing. It provides a unified evaluation protocol across diverse routing methods, together with three board datasets and eight evaluation metrics.

4.1 Task and Evaluation Protocol

Each instance is a KiCad board in the native `.kicad_pcb` format with its routes removed. Given the component placement and design rules stored in the board, the agent completes the routing in the same file. The PCBWorld-Bench evaluator scores the completed board from the outputs of the KiCad engine. Since the evaluation depends only on the completed board, routing methods that do not use PCBWorld and the reference solutions are scored the same way as our agents. Both the datasets and the metrics are pluggable. Users

Table 2: Dataset statistics: number of instances, and ranges of nets, pads, and layers per board.

Dataset	# inst. (train/valid/test)	# nets	# pads	# layers
D1 (Grid)	10,000 / 128 / 128	5	10	1
D2 (Gridless)	10,000 / 128 / 128	4–6	8–21	2
D3-A (Small)	– / – / 100	2–13	6–31	2–4
D3-B (Medium)	– / – / 287	5–42	31–100	2–4
D3-C (Large)	– / – / 292	6–451	101–2,103	2–8



(a) Grid-based (D1-10) (b) Grid-based (D1-50)

(c) Gridless (D2) (d) Open-source (D3-A)

(e) Open-source (D3-B) (f) Open-source (D3-C)

Figure 5: Example boards from PCBWorld-Bench. Representative examples from the three board types, synthetic grid-based boards (D1), synthetic gridless boards (D2), and real-world open-source boards (D3).

can add new `.kicad_pcb` boards as evaluation instances or define new metrics from any quantity the KiCad engine can compute.

4.2 Board Datasets

Table 2 summarizes the instance counts and key statistics of the three board datasets, and Figure 5 presents representative examples from each. Further details are provided in the datasheet and the license information in Appendix O and P, respectively.

(D1) Synthetic Grid-based Boards. D1 replicates the grid-routing task of Jumanji's *Connector* environment [5, 26] in PCBWorld. Boards are generated in KiCad format from the same distribution as Connector instances. Agents route on the same grid, and each wire occupies one grid cell. Five grid sizes are provided, $G \in \{10, 50, 100, 200, 500\}$, denoted D1-10 through D1-500. D1 enables fair comparison between RL policies trained in PCBWorld and state-of-the-art methods developed on Connector, such as Sable [47].

(D2) Synthetic Gridless Boards. D2 is a synthetic gridless routing dataset. We release the board generator that produced it, which controls the number of nets and pads per net, so routing difficulty is adjusted directly through these parameters. The released D2 samples 4–6 nets and 8–21 total pads on 2-layer boards, allowing vias for layer transitions. We split D2 into D2-train/valid/test.

(D3) Open-Source Boards. D3 is a curated set of 679 boards derived from PCBench's open-source PCB corpus [22]. We convert these boards to the KiCad 9 format while preserving their existing board information, including design-rule settings. We retain only boards that remain fully connected and free of DRVs after conversion. Each board retains its original routed traces, which serve as reference solutions for evaluation. These boards are partitioned into three subsets by pad count, D3-A, D3-B, and D3-C, ordered from smallest to largest. For zero-shot evaluation, all methods share a common evaluation set of 99 D3-A and 10 D3-B boards. The smaller D3-B sample reflects the high per-board evaluation cost of large boards for LLM agents (Appendix L).

4.3 Evaluation Metrics

For each board we draw five rollouts and select the single rollout with the largest potential gain. All metrics are computed on this selected rollout, except Time, which is averaged over all five rollouts (Appendix H). We use **Clean Pass** (CP) as the primary metric. CP counts a board as successful only if the selected rollout completes routing with full connectivity and the resulting board satisfies all given design rules, i.e., DRV=0. We further report **Potential Gain** (Pot.) as a routing-quality metric, defined as the change in board potential from the bare-board state to the completed routed state. Additional diagnostics include **Routability** (Rout.), the fraction of target connections successfully routed; **wallclock time** (Time), measured until completion or exhaustion of the step budget; **DRV count** (DRV), the aggregate number of KiCad-detected design-rule violations; and physical cost metrics such as **total wirelength** (WL) and **via count** (Via). For LLM agents we also report **Parse-fail**, the fraction of steps in the selected rollout whose response fails to parse into a valid action; such steps leave the board unchanged. We write @5 for this best-of-five protocol, the default for all reported metrics, and @1 when a metric scores a single rollout without selection. Deterministic methods are run once, as selection does not apply. Formal definitions and implementation details are provided in Appendix H.

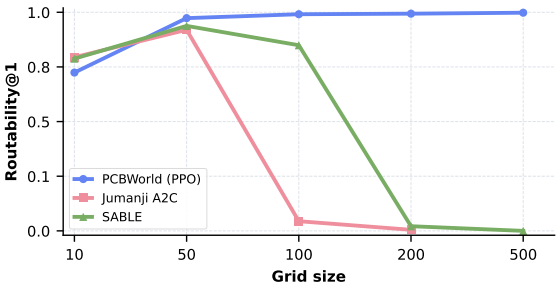


Figure 6: Simulation Results of Synthetic Grid-based Boards (D1). Single-rollout routability (Rout.@1↑) across grid sizes.

5 Experiments

We evaluate on each task suite described in Section 4: Synthetic Grid-based Board (D1), Synthetic Gridless Board (D2), and Open-Source Board (D3). Through D1, we show that the KiCad-API actions of PCBWorld scale to fine grid resolutions where grid-action methods break down. Through D2 and D3, we evaluate PCBWorld and both its LLM and RL agents on PCB routing against diverse baselines. The RL agents are trained on D2-train split, while the LLM agents are run without any additional fine-tuning.

5.1 Experimental Setup

PCBWorld agents. We build both LLM and RL agents that act through PCBWorld in a closed loop, observing the updated board state and engine-checked feedback after each action. We refer to this *interactive* mode throughout. This mirrors how we assess an engineer's routing ability by giving them an EDA tool. For **LLM agents**, we examine how geometric reasoning varies with model capacity across GPT-5.4, GPT-5.4-mini, GPT-5.4-nano [51, 52], and Qwen3.5-397B [57]. For **RL agents**, we train a small Transformer policy from scratch over the PCBWorld interface and report three variants, PPO (per-step), PPO (terminal), and GRPO. PPO (per-step) uses the per-step reward, while PPO (terminal) and GRPO use the terminal reward defined in Section 3.2. Since PPO (per-step) is our default policy, we write it as plain PPO unless noted otherwise. Further implementation and RL training details are provided in Appendices J and K.1.

Baselines. We evaluate three families of existing PCB routing approaches, none of which interact with PCBWorld: rule-based routers, grid-action RL agents, and open-loop LLMs. The `.kicad_pcb` files they route are scored by PCBWorld under the same metrics. **Rule-based routers** are among the most widely used open-source tools for PCB routing. We evaluate three rule-based routers: Freerouting [15], OrthoRoute [3], and KiCadRoutingTools (KRT) [19]. We train two **grid-action RL baselines** on Jumanji-Connector v2 [26], a recent cooperative multi-agent routing benchmark equivalent: A2C (Jumanji) [5] and Sable [47]. Both agents act on the grid, routing each net one cell at a time. We compare two **open-loop LLM baselines** on D2 and D3, *plan-only* and *engine-free*. Both produce their full routing in a single pass, without observing any intermediate board state. The plan-only baseline observes only the initial board and generates the full sequence of PCBWorld's routing actions in one pass. This complete sequence is then executed in PCBWorld to carry out the routing, mirroring prior LLM generation of operation sequences [55, 65]. The engine-free baseline instead bypasses PCBWorld and directly emits the routed `.kicad_pcb` itself, inserting wire segments and vias, in line with prior LLM generation of design artifacts [2, 39, 46].

5.2 Scalability Limits of Grid-Action Routing

On the synthetic grid-based board (D1), we trained and evaluated each method independently at each grid resolution. To match the training objective of A2C (Jumanji) and Sable, we trained PPO to maximize Rout.@1 (Figure 6). At grid 10, PPO is marginally below the grid-action baselines; it reaches near-perfect routability from grid 50 onward, while A2C collapses by grid 100 and Sable by grid 200. As the grid grows finer (Figs. 5a and 5b), grid-action baselines that route one cell at a time inevitably face a long horizon that weakens credit assignment. By contrast, PPO issues segment-level KiCad-API actions, so its decision horizon is tied to routed segments rather than grid cells and does not grow with grid resolution. Additional results are provided in Appendix L.

5.3 Comparing Routing Agents in PCBWorld

We compare all methods in Table 3 using CP as the primary metric, with Pot., Rout., and Time reported alongside. Additional per-metric DRV/WL/Via breakdowns with per-seed standard deviations are provided in Table 20. Figure 8 shows qualitative results produced by each method on D3-A.

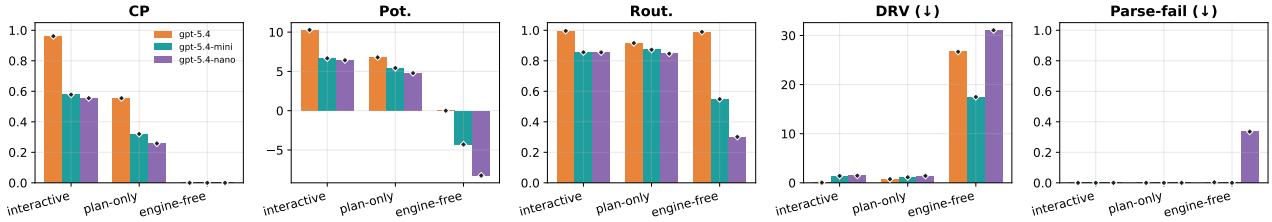
Trained from scratch on synthetic D2-train, a compact PPO Transformer outperforms all LLM agents and the remaining rule-based routers, with only Freerouting, the SOTA baseline, remaining competitive. On in-distribution D2-test, PPO matches Freerouting's perfect CP and Rout. (both 1.00) and leads on Pot. (10.52 vs. 9.76) and Time (0.43s vs. 2.69s). On out-of-distribution D3-A, it still attains higher CP (0.86 vs. 0.80), indicating strong generalization. On the larger and more complex D3-B split, however, Freerouting outperforms PPO (0.78 vs. 0.45), revealing the limits of this generalization on boards whose net and pad counts far exceed the D2-train distribution (Table 2).

LLM agents exhibit geometric reasoning that effectively leverages the core operations of PCBWorld without any routing-specific training, and this reasoning scales with model capacity. Specifically, GPT-5.4 reaches a CP of 0.96 on D2 and consistently outperforms its smaller mini (0.58) and nano (0.55) variants. All LLM agents nonetheless obtain 0.00 CP on D3-B, showing that their current geometric reasoning does not yet extend to larger boards. This gap positions large-board routing in PCBWorld-Bench as an open challenge for current LLMs, probing a form of spatial reasoning that today's models have yet to acquire. The strong performance of a compact PPO policy suggests RL fine-tuning of LLMs as a possible direction for future work toward expert routing agents.

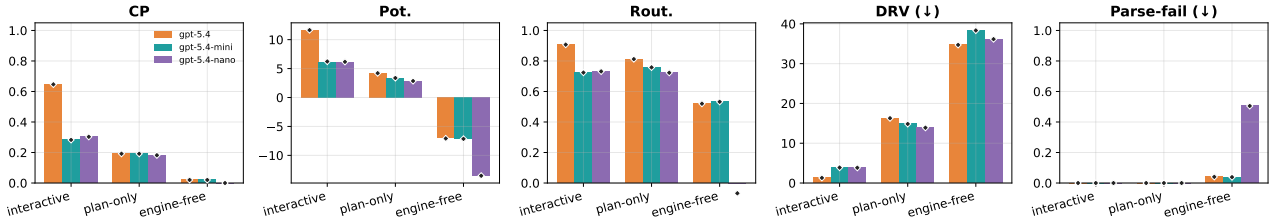
In terms of runtime efficiency, PPO (0.43s/1.83s on D2/D3-A) runs faster than Freerouting (2.69s/7.09s) on smaller boards and remains comparable on the largest, while the LLM agents are far slower throughout (94.04s/231.15s for GPT-5.4). The compact PPO policy generates a solution in a single learned rollout, whereas Freerouting,

Table 3: Routing quality on D2, D3-A, and D3-B. For each board we draw 5 rollouts and select the best with the highest Pot.; reported metrics are then computed on this one selected rollout, except Time, which is averaged over all 5 rollouts. We report clean pass (CP↑), potential gain (Pot.↑), and routability (Rout.↑), with wallclock time (Time↓) in seconds. Best CP and Pot. per split (excluding the Reference oracle) are in bold. Stochastic methods (Freerouting and the RL agents) report the mean over 4 seeds; LLM agents are evaluated with a single run (5 rollouts); deterministic methods are run once. Per-metric DRV/WL/Via breakdowns with per-seed std are in Table 20.

	D2				D3-A (99 Boards)				D3-B (10 Boards)			
Method	CP↑	Pot.↑	Rout.↑	Time↓	CP↑	Pot.↑	Rout.↑	Time↓	CP↑	Pot.↑	Rout.↑	Time↓
Reference	-	-	-	-	1.00	15.33	1.00	-	1.00	47.47	1.00	-
Freerouting [15]	**1.00**	9.76	1.00	2.69	0.80	**14.80**	0.91	7.09	**0.78**	**44.18**	1.00	9.94
OrthoRoute [3]	0.01	-4.58	0.30	2.54	0.02	-11.50	0.51	2.20	-	-	-	-
KiCadRoutingTools [19]	**1.00**	10.34	1.00	0.82	0.74	12.37	0.94	0.65	0.20	23.36	0.86	3.27
PCBWorld: Inference on different LLM backbones												
GPT-5.4	0.96	10.28	1.00	94.04	0.65	11.65	0.91	231.15	0.00	17.47	0.62	865.83
GPT-5.4-mini	0.58	6.66	0.86	33.93	0.28	6.22	0.72	56.85	0.00	13.76	0.61	422.06
GPT-5.4-nano	0.55	6.41	0.85	63.39	0.30	6.17	0.73	100.76	0.00	13.32	0.59	510.25
Qwen3.5-397B	0.33	3.85	0.74	47.58	0.34	6.88	0.75	81.82	0.00	8.82	0.51	222.29
PCBWorld: RL trained on Transformer												
PPO	**1.00**	**10.52**	1.00	0.43	**0.86**	13.59	0.95	1.83	0.45	33.85	0.85	10.77
GRPO	**1.00**	9.16	1.00	1.16	0.85	12.09	0.98	3.54	0.10	16.47	0.69	11.20
PPO (terminal)	0.00	-5.24	0.26	5.53	0.01	-5.81	0.23	5.97	0.00	-14.98	0.09	9.02



(a) D2 (Synthetic Gridless Board).



(b) D3-A (Open-source Board).

Figure 7: Engine engagement in the LLM agent across interactive, plan-only, and engine-free modes. Table 20 in Appendix L reports full results. (Negative routability is caused by unnecessary floating traces that are unrelated to net connectivity.)

though also lightweight, iteratively searches and compares routing candidates before committing. The LLM agents are slowest, as each action is costly to generate and the model often wanders without completing the routing.

5.4 Impact of Engine Engagement in LLM

Figure 7 compares three engine-engagement levels of the LLM agent on D2 and D3-A, *interactive*/*plan-only*/*engine-free*, where *interactive* is our PCBWorld agent and *plan-only*/*engine-free* are the two open-loop baselines. On our primary metrics CP and Pot., the three levels rank in this order consistently across all GPT models and both splits. For GPT-5.4, CP is 0.96/0.55/0.00 on D2 and 0.65/0.19/0.02 on D3-A.

Leveraging the native API is essential for producing design-rule-clean routing. The engine-free agent records far worse DRV than the interactive and plan-only agents. Generating wire segments

without the API requires exacting geometric reasoning to keep every track within the design rules, an ability current models largely lack. Smaller models degrade further, and under engine-free generation GPT-5.4-nano often fails to produce even valid KiCad syntax for 34–51% of outputs across D2 and D3-A.

Although the plan-only and interactive agents leverage the same PCBWorld action space and both form connections, only the interactive agent keeps them design-rule-clean, so DRV (1.29 vs. 16.3) and CP (0.65 vs. 0.19) diverge sharply on D3-A. As in an engineer's routing workflow, both acting through the native API and accurate observation of the board state are essential for clean PCB routing.

5.5 Impact of Reward Design in RL

Controllability through penalty weights. We test whether the policy responds to fine-grained routing objectives by sweeping the wirelength weight $\lambda_w \in \{0, 0.001, 0.002\}$ and the via weight $\lambda_v \in \{0, 0.05, 0.1\}$ in a 3×3 factorial design (Fig. 9). We analyze the full factorial results by aggregating them along each axis. Averaged over all via-weight settings, increasing λ_w consistently reduces wirelength from 484.8 mm to 468.4 mm and 462.4 mm. Likewise, averaged over all wirelength-weight settings, increasing λ_v reduces the via count from 4.06 to 2.82 and 2.27. Although wirelength and via count are coupled in PCB routing, these marginal trends show that each penalty acts as a consistent optimization signal for its associated cost. Overall, PCBWorld adapts its routing behavior to the specified objective rather than collapsing to a single operating point.

Reward shaping interacts with the optimizer. We compare PPO (per-step) against the terminal-reward variants PPO (terminal) and GRPO (Table 3). PPO (per-step) performs best overall, matching the top methods on D2 and achieving the highest non-reference CP on D3-A. Its advantage is clearest on the longer-horizon D3-B split, where it reaches CP 0.45, versus 0.10 for GRPO and 0.00 for PPO (terminal). This highlights the importance of dense intermediate feedback for credit assignment in long-horizon PCB routing. Meanwhile, under the same terminal-reward formulation, GRPO substantially outperforms PPO (terminal), suggesting that different reward granularities may require different optimization methods. Overall, PPO (per-step) provides the most effective combination in our experiments.

6 Related Work

Rule-based PCB routers. Automatic PCB routing has a long algorithmic history, spanning grid-based maze search since Lee's algorithm [35] and gridless techniques [61], with rip-up and reroute as the central iterative heuristic [12, 44]. Academic refinements continue this line [9, 21, 43]. Rule-based routers remain central in practice, with open-source routers such as Freerouting, OrthoRoute, and KiCadRoutingTools [3, 15, 19] and commercial EDA suites as the production standard [1, 6]. Since this prior knowledge is encoded through manually designed rules and heuristics, performance depends on objective- and board-specific tuning.

RL for PCB routing. Existing RL work narrows the agent's action interface in one of two ways. The first family casts routing as cell-by-cell movement on a discretized grid [5, 23, 26, 41, 47]; at the grid resolution production boards require, the search space grows impractically large. The second family routes only indirectly. The agent selects high-level options for a fixed auto-router, such as which net to route next [40, 42, 73, 74], which routing pattern to follow [8], or which fanout locations to route from [36], and the router draws all geometry. Routing quality is then capped by the ceiling of the underlying router. In neither family does the agent draw board geometry through the engine's native operations. PCBWorld instead exposes the engine's native routing operations, the ones an engineer uses in the actual routing workflow, as the agent's action space [31, 32], and derives the learning signal from the engine's design-rule check rather than from a hand-crafted geometric proxy.

LLM agents for EDA. LLM agents that pair reasoning with tool calls [72] are beginning to reach EDA, and existing work divides into two groups by feedback structure. One group generates EDA scripts or HDL code in a single open-loop pass [45, 46, 60, 66]. The other uses an EDA tool as a verifier and refines the artifact iteratively. The agent repairs HDL with compilation and simulation feedback [4, 24, 63, 69], generates schematics under an ERC check (a stage before board routing) [75], or tunes flow parameters on reported metrics [17]. In both groups, however, the agent decides at the level of the artifact. It emits a whole script, a whole code file, or a parameter set, and feedback arrives at that same granularity. In neither group do the engine's native operations serve as the agent's tools. Within PCB, PCB-Bench [37] evaluates an LLM's ability to interpret a board but does not score design actions themselves. In PCBWorld, the agent acts through the engine's native operations and receives stepwise feedback on the resulting board state.

Agents in other domains. Outside EDA, LLM-based design generation has been studied most actively in 3D CAD. Building on sequence representations of CAD models [67], these methods generate CAD operation sequences or parametric code from text and images [18, 38, 39, 50, 65], and benchmarks score the reconstruction of a labeled target shape [64], a feasibility standard softer than design rules where a single violation invalidates a board. Agent benchmarks in general software domains pose tasks over code repositories, API calls, and user interactions, and verify outcomes by test runs or state comparison [28, 53, 56, 70, 71]. These tasks involve no engine that adjudicates geometry and design rules. Computer-use environments and GUI agents [10, 25, 68] are general enough to host an EDA tool in principle. Pixel-level GUI control, however, is unlikely to reach the precision and speed that micrometer-scale routing and large-scale training demand. PCBWorld instead provides a PCB-specific headless and vectorized environment that directly supports RL policy training, and PCBWorld-Bench scores only the completed board file, so baselines operating outside PCBWorld are evaluated under the same engine verification. PCBWorld-Bench is the first integrated PCB *design* benchmark under a single engine-verified evaluator.

7 Discussion

Towards explainable, reasoning-based routing. We explicitly configure our LLM agents to generate a `<think>` block to improve their inference performance. This process not only improves reasoning

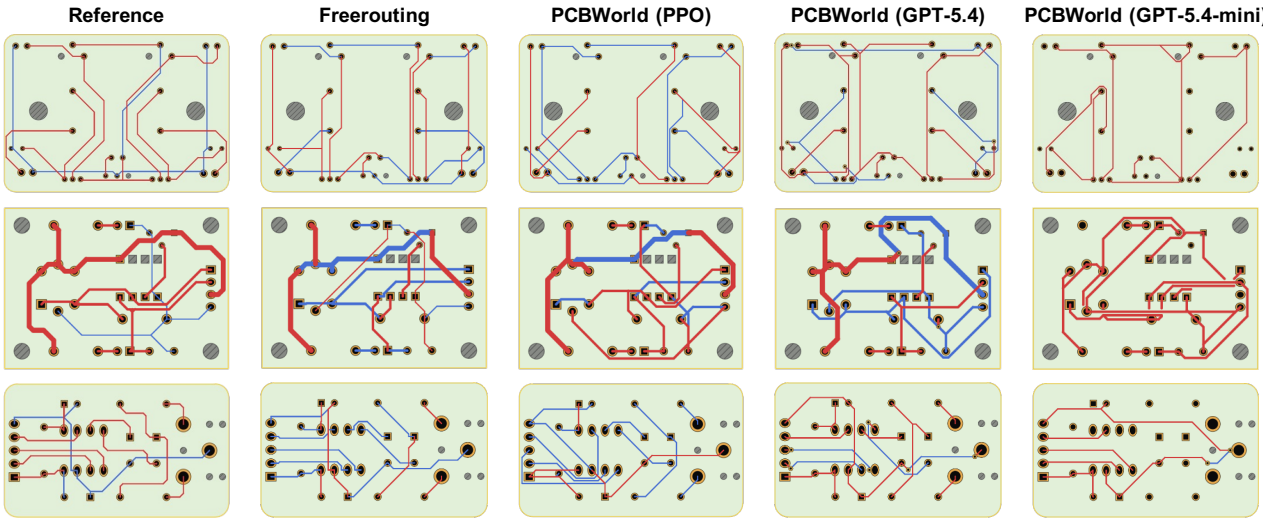


Figure 8: Examples of routed boards (D3-A). Wire traces in different colors (red and blue) represent wire segments on distinct copper layers. The results demonstrate that PCBWorld enables effective routing that closely resembles the reference solution while satisfying various constraints, including track width and other design rules.

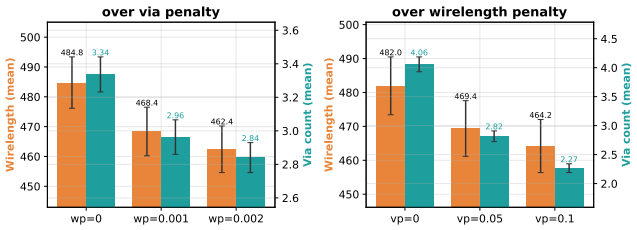


Figure 9: Reward weight sweep (3×3 factorial). Bars show marginal means with error bars. Wirelength: left axis (orange); via count: right axis (teal).

robustness by accumulating the model's intermediate thoughts, but also enables explainable reasoning behind the final action. In practice, by analyzing the model's <think></think> blocks in routing tasks with PCBWorld, we identify several key observations, which are presented in Appendix N. We expect that deeper analysis of the reasoning traces produced by specialized routing agents can further strengthen their routing reasoning and lead to routing policies that are explainable in natural language.

Action and API coverage. Track-removal APIs such as `delete_track_near` are exposed in PCBWorld but excluded from the agent's action space, precluding iterative refinement such as rip-up and reroute, which high-quality routing typically relies on [12, 15, 44]. We plan to expose the engine's remaining routing APIs, add such corrective actions to the agent's action space, and extend the task to component placement, where component locations are jointly optimized rather than fixed.

Evaluation realism. PCBWorld-Bench scores boards using engine-checked geometric and connectivity outcomes (DRV, wirelength, via count), whereas industrial practice requires signal integrity [20], electromagnetic interference [54], and thermal objectives [34]. Incorporating these signals into the potential Φ to shape both evaluation and RL training is a natural next step.

8 Conclusion

We present PCBWorld, a Gym environment that wraps the open-source KiCad EDA engine through a Python API and formulates PCB routing as an MDP without any surrogate abstraction. Two lightweight wrappers over this single MDP re-encode the same state, action, and reward for RL policies and tool-using LLM agents, allowing both families to train and act in the same environment. Built on PCBWorld, PCBWorld-Bench provides a KiCad port of Jumanji's Connector across diverse grid scales (D1), a synthetic gridless board generator (D2), and a re-curated set of PCBench-derived open-source boards [22] reserved for zero-shot evaluation (D3), forming a benchmark for geometric feasibility reasoning in which only the component placement and design rules are given. Empirically, we find that (i) acting as a human engineer does, the KiCad-API action interface outperforms both grid-action RL and engine-free LLM generation of `.kicad_pcb` files; (ii) a multi-axis evaluation of LLM agents across model scale and engine engagement charts how far current LLMs can serve as routing agents; and (iii) a compact Transformer policy trained from scratch with PPO approaches a rule-based router refined over decades and transfers zero-shot to open-source boards, offering a practical recipe for training routing policies. By releasing PCBWorld and PCBWorld-Bench together with code, baselines, and the evaluation harness, we aim to provide a common engine-grounded foundation on which learning-based EDA and tool-augmented LLM research can advance.

References

[1] Altium. 2026. Altium Designer: The Industry's Leading PCB Design Software. https://www.altium.com/altium-designer Accessed: 2026-06-11.

[2] Akshay Badagabettu, Sai Sravan Yarlagadda, and Amir Barati Farimani. 2024. Query2CAD: Generating CAD models using natural language queries. arXiv preprint arXiv:2406.00144. doi:10.48550/arXiv.2406.00144

[3] Brian Benchoff. 2025. OrthoRoute: A GPU-accelerated PCB autorouter for KiCad. https://github.com/bbenchoff/OrthoRoute

[4] Jason Blocklove, Shailja Thakur, Benjamin Tan, Hammond Pearce, Siddharth Garg, and Ramesh Karri. 2025. Automatically Improving LLM-based Verilog Generation using EDA Tool Feedback. *ACM Transactions on Design Automation of Electronic Systems* 30, 6 (2025), 1–26. doi:10.1145/3723876 arXiv:2411.11856.

[5] Clément Bonnet, Daniel Luo, Donal John Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence Illing Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries Petrus Smit, Nathan Grinsztajn, Raphael Boige, Cemlyn Neil Waters, Mohamed Ali Ali Mimouni, Ulrich Armel Mbou Sob, Ruan John de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. 2024. Jumanji: A Diverse Suite of Scalable Reinforcement Learning Environments in JAX. In *International Conference on Learning Representations (ICLR)*. arXiv:2306.09884.

[6] Cadence Design Systems. 2026. Allegro X Design Platform: PCB and System Design. https://www.cadence.com/en_US/home/tools/pcb-design-and-analysis/allegro-x-design-platform.html Accessed: 2026-06-11.

[7] CERN BE-CO-HT. 2026. CERN BE-CO-HT contribution to KiCad. https://ohwr.org/projects/cern-kicad/

[8] Hao Chen, Kai-Chieh Hsu, Walker J. Turner, Po-Hsuan Wei, Keren Zhu, David Z. Pan, and Haoxing Ren. 2023. Reinforcement Learning Guided Detailed Routing for Custom Circuits. In *Proceedings of the 2023 International Symposium on Physical Design (ISPD)*. 26–34. doi:10.1145/3569052.3571874

[9] Jiarui Chen, Yujing Zhou, Qinghai Liu, and Xinhong Zhang. 2023. A Novel Global Routing Algorithm for Printed Circuit Boards Based on Triangular Grid. *Electronics* 12, 24 (2023), 4942. doi:10.3390/electronics12244942

[10] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. 2024. SeeClick: Harnessing GUI Grounding for Advanced Visual GUI Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Bangkok, Thailand, 9313–9332. doi:10.18653/v1/2024.acl-long.505 arXiv:2401.10935.

[11] Clyde F. Coombs, Jr. and Happy T. Holden (Eds.). 2016. *Printed Circuits Handbook* (7 ed.). McGraw Hill. https://www.mheducation.com/highered/mhp/product/printed-circuits-handbook-seventh-edition.html

[12] W. A. Dees, Jr. and P. G. Karger. 1982. Automated rip-up and reroute techniques. In *Proceedings of the 19th Design Automation Conference (DAC)*. 432–439. doi:10.1145/800263.809241

[13] Antonella Del Rosso. 2015. KiCad Software Gets the CERN Treatment. https://cds.cern.ch/record/2000246 CERN Document Server, 2015-02-17.

[14] Roberto Fernandez Bautista. 2025. KiCad: a Free and Open Source Software (FOSS) tool for Printed Circuit Board (PCB) design. https://videos.cern.ch/record/3023308 CERN Videos, record 3023308.

[15] Freerouting Contributors. 2026. Freerouting: Open-source PCB Autorouter. https://github.com/freerouting/freerouting

[16] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. 2021. Datasheets for datasets. *Commun. ACM* 64, 12 (2021), 86–92. doi:10.1145/3458723

[17] Amur Ghose, Andrew B. Kahng, Sayak Kundu, and Zhiang Wang. 2025. ORFS-agent: Tool-Using Agents for Chip Design Optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*. 1–13. doi:10.1109/MLCAD65511.2025.11189204 arXiv:2506.08332.

[18] Yandong Guan, Xilin Wang, XiMing Xing, Jing Zhang, Dong Xu, and Qian Yu. 2025. CAD-Coder: Text-to-CAD Generation with Chain-of-Thought and Geometric Reward. In *Advances in Neural Information Processing Systems*, Vol. 38. 59765–59789. arXiv:2505.19713.

[19] Andy Haas. 2026. KiCad Routing Tools. https://github.com/drandyhaas/KiCadRoutingTools

[20] Stephen H. Hall and Howard L. Heck. 2009. *Advanced Signal Integrity for High-Speed Digital Designs*. Wiley-IEEE Press. doi:10.1002/9780470423899

[21] Youbiao He. 2024. *Towards Automated PCB Routing: Leveraging Machine Learning and Heuristic Techniques*. Ph. D. Dissertation. Iowa State University. doi:10.31274/td-20240617-74

[22] Youbiao He, Jacob Frieden, Hebi Li, Roba Abbajabal, Ge Luo, and Forrest Sheng Bao. 2024. PCBench: A Dataset for Printed Circuit Board Routing. https://github.com/PCBench/PCBench DAC 2024 work-in-progress poster.

[23] Youbiao He, Hebi Li, Jin Tian, and Forrest Sheng Bao. 2022. Circuit Routing Using Monte Carlo Tree Search and Deep Reinforcement Learning. In *2022 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 1–5. doi:10.1109/VLSI-DAT54769.2022.9768074

[24] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 39. 300–307. doi:10.1609/aaai.v39i1.32007 arXiv:2408.08927.

[25] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. 2024. CogAgent: A Visual Language Model for GUI Agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 14281–14290. doi:10.1109/CVPR52733.2024.01354 arXiv:2312.08914.

[26] InstaDeep. 2026. Connector Environment — Jumanji Documentation. https://instadeepai.github.io/jumanji/environments/connector/ Last updated: 2026-06-02.

[27] IPC. 2003. IPC-2221A-2003: Generic Standard on Printed Board Design. https://webstore.ansi.org/standards/ipc/ipc2221a2003 Association Connecting Electronics Industries.

[28] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *ICLR*. https://openreview.net/forum?id=VTF8yNQM66

[29] R. S. Khandpur. 2006. *Printed Circuit Boards: Design, Fabrication, Assembly and Testing*. McGraw-Hill. https://search.worldcat.org/title/62032512

[30] KiCad Project. 2024. KiCad Developer Documentation: APIs and Bindings. https://dev-docs.kicad.org/en/apis-and-binding/

[31] KiCad Project. 2026. KiCad API Python Bindings. https://docs.kicad.org/kicad-python-main/

[32] KiCad Project. 2026. KiCad EDA Suite. https://www.kicad.org Accessed: 2026-06-11.

[33] KiCad Project. 2026. KiCad PCB Editor Documentation. https://docs.kicad.org/9.0/en/pcbnew/pcbnew.html Documentation revision 152cd19e; accessed: 2026-06-11.

[34] Pradeep Lall, Michael G. Pecht, and Edward B. Hakim. 1997. *Influence of Temperature on Microelectronics and System Reliability: A Physics of Failure Approach* (1 ed.). CRC Press.

[35] C. Y. Lee. 1961. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers* EC-10, 3 (1961), 346–365. doi:10.1109/TEC.1961.5219222

[36] Haiyun Li, Jixin Zhang, Ning Xu, and Mingyu Liu. 2023. FanoutNet: A Neuralized PCB Fanout Automation Method Using Deep Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 37. 8554–8561. doi:10.1609/aaai.v37i7.26030

[37] Jindong Li, Lianrong Chen, Bin Yang, Jiadong Zhu, Ying Wang, Yuzhe Ma, and Menglin Yang. 2026. PCB-Bench: Benchmarking LLMs for Printed Circuit Board Placement and Routing. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=Q5QLu7XTWx

[38] Jiahao Li, Yusheng Luo, Yunzhong Lou, and Xiangdong Zhou. 2026. ReCAD: Reinforcement Learning Enhanced Parametric CAD Model Generation with Vision-Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 40. 6190–6198. doi:10.1609/aaai.v40i8.37544 arXiv:2512.06328.

[39] Xingang Li, Yuewan Sun, and Zhenghui Sha. 2025. LLM4CAD: Multimodal Large Language Models for Three-Dimensional Computer-Aided Design Generation. *Journal of Computing and Information Science in Engineering* 25, 2 (2025), 021005. doi:10.1115/1.4067085

[40] Haiguang Liao, Qingyi Dong, Xuliang Dong, Wentai Zhang, Wangyang Zhang, Weiyi Qi, Elias Fallon, and Levent Burak Kara. 2020. Attention Routing: Track-Assignment Detailed Routing Using Attention-Based Reinforcement Learning. arXiv preprint arXiv:2004.09473. doi:10.48550/arXiv.2004.09473

[41] Haiguang Liao, Wentai Zhang, Xuliang Dong, Barnabás Póczos, Kenji Shimada, and Levent Burak Kara. 2020. A Deep Reinforcement Learning Approach for Global Routing. *Journal of Mechanical Design* 142, 6 (2020), 061701. doi:10.1115/1.4045044 arXiv:1906.08809.

[42] Yin-Chi Liao, Sheng-Xin Pan, and Po-Jui Chiang. 2026. Automation of PCB Autorouting via World-Model Reinforcement Learning and Freerouting Integration. *Expert Systems with Applications* 311 (2026), 131424. doi:10.1016/j.eswa.2026.131424

[43] Ting-Chou Lin, Devon Merrill, Yen-Yi Wu, Chester Holtz, and Chung-Kuan Cheng. 2021. A Unified Printed Circuit Board Routing Algorithm With Complicated Constraints and Differential Pairs. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 170–175. doi:10.1145/3394885.3431568

[44] Ralph Linsker. 1984. An iterative-improvement penalty-function-driven wire routing system. *IBM Journal of Research and Development* 28, 5 (sep 1984), 613–624. doi:10.1147/RD.285.0613

[45] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, Bonita Bhaskaran, Bryan Catanzaro, Arjun Chaudhuri, Sharon Clay, Bill Dally, Laura Dang, Parikshit Deshpande, Siddhanth Dhodhi, Sameer Halepete, Eric Hill, Jiashang Hu, Sumit Jain, Ankit Jindal, Brucek Khailany, George

Kokai, Kishor Kunal, Xiaowei Li, Charley Lind, Hao Liu, Stuart Oberman, Sujeet Omar, Ghasem Pasandi, Sreedhar Pratty, Jonathan Raiman, Ambar Sarkar, Zhengjiang Shao, Hanfei Sun, Pratik P. Suthar, Varun Tej, Walker Turner, Kaizhe Xu, and Haoxing Ren. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. arXiv preprint arXiv:2311.00176. doi:10.48550/arXiv.2311.00176

[46] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Invited Paper: VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–8. doi:10.1109/ICCAD57390.2023.10323812 arXiv:2309.07544.

[47] Omayma Mahjoub, Sasha Abramowitz, Ruan John De Kock, Wiem Khlifi, Simon Verster Du Toit, Jemma Daniel, Louay Ben Nessir, Louise Beyers, Juan Claude Formanek, Liam Clark, and Arnu Pretorius. 2025. Sable: A Performant, Efficient and Scalable Sequence Model for MARL. In *Proceedings of the 42nd International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 267)*. 42579–42614. arXiv:2410.01706.

[48] Larry McMurchie and Carl Ebeling. 1995. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays (FPGA95)*. ACM, 111–117. doi:10.1145/201310.201328

[49] Andrew Y. Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML)*. 278–287. https://dl.acm.org/doi/10.5555/645528.657613

[50] Ke Niu, Haiyang Yu, Zhuofan Chen, Mengyang Zhao, Teng Fu, Bin Li, and Xiangyang Xue. 2026. From Intent to Execution: Multimodal Chain-of-Thought Reinforcement Learning for Precise CAD Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 40. 8160–8167. doi:10.1609/aaai.v40i10.37763 arXiv:2508.10118.

[51] OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/ Accessed: 2026-06-11.

[52] OpenAI. 2026. Introducing GPT-5.4 mini and nano. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/ Accessed: 2026-06-11.

[53] Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 48371–48392. https://proceedings.mlr.press/v267/patil25a.html

[54] Clayton R. Paul. 2006. *Introduction to Electromagnetic Compatibility* (2 ed.). Wiley-Interscience. doi:10.1002/0471758159

[55] Dacheng Qi, Chenyu Wang, Jingwei Xu, Tianzhe Chu, Zibo Zhao, Wen Liu, Wenrui Ding, Yi Ma, and Shenghua Gao. 2026. Pointer-CAD: Unifying B-Rep and Command Sequences via Pointer-based Edges & Faces Selection. arXiv preprint arXiv:2603.04337. doi:10.48550/arXiv.2603.04337 Accepted by CVPR 2026.

[56] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=dHng2O0Jjr

[57] Qwen Team. 2026. Qwen3.5: Towards Native Multimodal Agents. https://qwen.ai/blog?id=qwen3.5

[58] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347. doi:10.48550/arXiv.1707.06347

[59] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv preprint arXiv:2402.03300. doi:10.48550/arXiv.2402.03300

[60] Utsav Sharma, Bing-Yue Wu, Sai Rahul Dhanvi Kankipati, Vidya A. Chhabria, and Austin Rovinski. 2024. OpenROAD-Assistant: An Open-Source Large Language Model for Physical Design Tasks. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*. 1–7. doi:10.1145/3670474.3685960

[61] Naveed A. Sherwani. 1999. *Algorithms for VLSI Physical Design Automation* (3rd ed.). Kluwer Academic Publishers. doi:10.1007/b116436

[62] The Linux Foundation. 2019. KiCad Joins Linux Foundation to Advance Electronic Design Automation. https://www.linuxfoundation.org/press/press-release/kicad-joins-linux-foundation-to-advance-electronic-design-automation Press release, 2019-11-22.

[63] Yunda Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1145/3649329.3657353 arXiv:2311.16543.

[64] Liang Wang, Heng Meng, Zekai Xiang, Jin Liu, Pingyi Zhou, Litao Chen, and Yongqiang Tang. 2026. Text2CAD-Bench: A Benchmark for LLM-based Text-to-Parametric CAD Generation. arXiv preprint arXiv:2605.18430. doi:10.48550/arXiv.2605.18430

[65] Siyu Wang, Cailian Chen, Xinyi Le, Qimin Xu, Lei Xu, Yanzhou Zhang, and Jie Yang. 2025. CAD-GPT: Synthesising CAD Construction Sequence with Spatial Reasoning-Enhanced Multimodal LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 39. 7880–7888. doi:10.1609/aaai.v39i8.32849 arXiv:2412.19663.

[66] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2024. ChatEDA: A Large Language Model Powered Autonomous Agent for EDA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 10 (2024), 3184–3197. doi:10.1109/TCAD.2024.3383347 arXiv:2308.10204.

[67] Rundi Wu, Chang Xiao, and Changxi Zheng. 2021. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 6752–6762. doi:10.1109/ICCV48922.2021.00670 arXiv:2105.09492.

[68] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J. Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems 37*, Vol. 37. 52040–52094. doi:10.52202/079017-1650 arXiv:2404.07972.

[69] Ke Xu, Jialin Sun, Yuchen Hu, Xinwei Fang, Weiwei Shan, Xi Wang, and Zhe Jiang. 2024. MEIC: Re-thinking RTL Debug Automation Using LLMs. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9. doi:10.1145/3676536.3676801 arXiv:2405.06840.

[70] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. 50528–50652. doi:10.52202/079017-1601 arXiv:2405.15793.

[71] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. 2025. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. In *The Thirteenth International Conference on Learning Representations*. arXiv:2406.12045.

[72] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. https://openreview.net/forum?id=WE_vluYUL-X

[73] Zhanwen Zhou, Hankz Hankui Zhuo, Xiaowu Zhang, and Qiyuan Deng. 2023. XRoute Environment: A Novel Reinforcement Learning Environment for Routing. arXiv preprint arXiv:2305.13823. doi:10.48550/arXiv.2305.13823

[74] Zhanwen Zhou, Hankz Hankui Zhuo, Jinghua Zhou, and Wushao Wen. 2025. Transformer-based Reinforcement Learning for Net Ordering in Detailed Routing. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*. 9492–9500. doi:10.24963/ijcai.2025/1055

[75] Huanghaohe Zou, Peng Han, Emad Nazerian, and Alex Q. Huang. 2026. PCB-SchemaGen: Constraint-Guided Schematic Design via LLM for Printed Circuit Boards (PCB). arXiv preprint arXiv:2602.00510. doi:10.48550/arXiv.2602.00510

A Notation Glossary

Table 4: Notation.

Symbol	Description
$n_{\text{drv}}(\cdot)$	Number of design-rule violations (DRVs)
$n_{\text{via}}(\cdot)$	Number of vias
$\ell(\cdot)$	Total wire length
$\mathcal{R}$	Design space
s	Board state, represented as a state dictionary
s_t	Board state at step t
$\Phi(\cdot)$	Potential function
$r(\cdot)$	Reward function
G	Grid size

B State-dictionary schema

The environment projects the raw KiCad `.kicad_pcb` S-expression through four progressively narrower views: the C++ *engine* (`KiCadEngine`, the authoritative live state), a JSON *Gym observation* (`KiCadHLEnv`), an *RL token stream* (`BatchedStateTokenizer`), and an *LLM text prompt* (S-expression / XML rendering). Table 5 lists every logical field once and marks where it appears in the chain. The chain is *largely lossless in the sense that matters for routing*: every field the agent does not see is one the engine still retains and enforces on the agent's behalf.

Lossless where it matters. Every row of Table 5 that drops out below the engine column does so for one of three reasons, none of which compromises the agent's ability to plan a legal route.

(1) The engine still enforces it. Via drill and pad rotation disappear from the agent's view but are used by the C++ DRC engine when it scores the agent's track attempts; the agent decides "place a via here", not "place a via with this drill". Per-netclass design rules are collapsed for the agent into a single strictest envelope, but on `net_select` the environment pushes the resolved class's track width, via diameter and drill into PNS, so the engine routes at the correct per-class dimensions even though the agent only sees the envelope.

(2) The field is derivable from what the agent does see. The `state_code` duplicates `is_routing` plus the phase flag; `is_dragging` is non-zero only in interactive sessions the policy never triggers; track and via widths in the LLM prompt are recoverable from `board_constraints` for routing-quality reasoning.

(3) The field is not relevant to routing decisions. Pad names, footprint references, layer-stack-up dielectrics, 3D models, UUIDs and timestamps are dropped throughout; the schema synthesizes its own short, episode-local IDs (`P0`, `T0`, ...) for addressing.

The genuine losses are limited to two: copper pours (`(zone ...)`), which the engine itself does not represent in the PNS routable space, and per-class rule differentiation as visible to the agent. Both are recoverable upgrades rather than architectural barriers – the file's information already reaches the engine; surfacing it would extend the projection rather than replace it.

Differences between the RL and LLM views are representational, not informational. The two agent-facing layers see the same underlying state but consume it differently. The RL stream emits a discrete `(action_type, pointer_idx, routing_mode)` triple over a candidate pool synthesized from the current net's pads, vias, track endpoints and a directional grid; coordinates are therefore selected from a finite snap-set. The LLM emits free-form text constrained by a guided-decoding regex, allowing it to invent detour waypoints anywhere within the bounding box. The RL stream additionally encodes the top-32 DRC violations and an augmentation context for coordinate symmetries; the LLM compensates with multi-turn history, a no-effect retrofit marker on `start_route`+empty-effect pairs, and a rejection-streak slot that surfaces consecutive identical mask rejections. In both cases the final filter is the engine: mask rejection, pointer-out-of-pool (RL only) and parse failure (LLM only) all collapse to an `idle` fallback, so PNS never sees illegal input regardless of which consumer is driving.

C System Architecture (PCBWorld)

This appendix distinguishes carefully among policy decisions, Gym-env mediation, the wrapper that surfaces 58 engine APIs, and the C++ binding underneath. Confusing these four layers is the most common source of misreading the state, action, and reward decomposition in Section D, so we fix the layered picture here. Figure 10 gives the agent–environment view of the stack and the RL signals connecting the two sides.

C.1 Layering the stack

At the top, three policy classes share an identical observation and action interface: a deterministic rule-based router as a baseline, an RL agent built on a from-scratch Transformer, and an LLM agent that issues tool calls. They all enter the stack through `KiCadHLEnv` at L4, the

Table 5: Field-by-field schema across the projection layers. ✓ = present, ○ = present in reduced form, X = absent.

Field	`.kicad_pcb` origin	Engine	JSON	RL	LLM
Geometry					
Board bbox / copper layer count	derived from outline	✓	✓	✓	✓
Board outline (Edge.Cuts)	`(gr_line/arc/...)`	○[a]	✓	✓	○[b]
Pad geometry (xy, w, h, layer)	`(pad ...)`	✓	✓	✓	○[c]
Pad metadata (name, ref, rotation, shape)	`(pad/footprint ...)`	✓	○[d]	X	○[d]
Track segment	`(segment ...)`	✓	✓	✓	○[b]
Via outer diameter / layer span	`(via ...)`	✓	✓	✓	○[e]
Via drill diameter	`(via ...)`	✓	X	X	X
Zone / copper pour	`(zone ...)`	X	X	X	X
Net / connectivity					
Net membership	`(net N)`	✓	✓	✓[f]	✓
Unconnected / NPTH pads	`net 0`, `np_thru_hole`	✓	✓	X	○[g]
Ratsnest (unrouted edges)	— (computed)	✓	✓	✓	✓
Design rules					
Per-netclass distribution	`(setup)` / `.kicad_pro`	✓	X	X	X
BDS minima, presets	`(setup ...)`	✓	X	X	X
Strictest effective constraints	— (max over BDS + classes)	✓	✓	X	✓[h]
Router state					
Head xy / layer / net / phase / mode	in-memory	✓	✓	✓	✓
`is_dragging`, `state_code`, `routing_target`	in-memory	✓	✓	X	X
`step`, `step_ratio`, prev-action	env-synthesized	—	✓	✓	○[i]
DRC					
Top-k violation list	— (`run_drc`)	✓	✓(top-32)	✓	X[j]
Per-net counts, error/warning split	—	✓	info only	X	X
Action interface					
6-way action mask	env-synthesized	—	info	logit mask	valid-action listing + grammar
Coordinate emission	—	—	—	discrete pointer pool	continuous text + regex grammar
Self-feedback (no-effect, streak)	—	—	X	X	✓[k]

[a] curves linearized once at the C++ binding; [b] segment / edge widths dropped from the textual prompt; [c] sexpr keeps `xy+layer` only, xml keeps full geometry; [d] shape only; [e] diameter dropped from the textual prompt; [f] via slot id, name dropped; [g] NPTH only as obstacles, unconnected pads omitted; [h] emitted as `board_constraints`, omitting unset fields; [i] history rendered as multi-turn text; [j] reward signal only, not echoed in the prompt; [k] LLM-specific (no parameter-side memory).

Gymnasium environment facade that owns four sub-modules: an action dispatcher for command routing, an observation builder, a reward potential composer, and an action mask. The dispatcher is the only L4 sub-module that mutates state; the other three are read paths. L4 holds a single `KiCadEngine` instance at L3, the Python wrapper around the C++ router that exposes the 58 step-level APIs the rest of the appendices catalogue. L3 is the only Python call site that reaches the Python ↔ C++ binding at L2, behind which sits the unmodified KiCad C++ engine at L1.

C.2 Building observations and rewards

This subsection and the next together specify the L3↔L4 signal exchange that synthesises an MDP interface from the unmodified engine, which is what makes the stack an RL environment rather than a Python SDK over the 58 APIs. Every read crossing back from L3 into Python is packaged into one of three dataclasses. `BoardSnapshot` is the geometry bundle that the observation builder consumes. `RewardSnapshot` is the unrouted, wirelength, and DRC bundle that the reward potential composer folds into $\Phi(s)$. `RoutingSessionState` is the routing FSM bundle that lets the action mask and the action dispatcher decide which actions are admissible. Because these three are the only read channels, the L3↔L4 coupling reduces to which engine method populates which dataclass field, which is exactly the reading the dual-tag analysis in Section D.2 formalises.

C.3 Dispatching actions

Policies do not call the 58 step-level APIs directly. They emit a single high-level action through the action dispatcher at L4, which translates that action into a small subset of L3 mutators. Section D catalogues the 58 APIs and their MDP components, and the high-level action surface

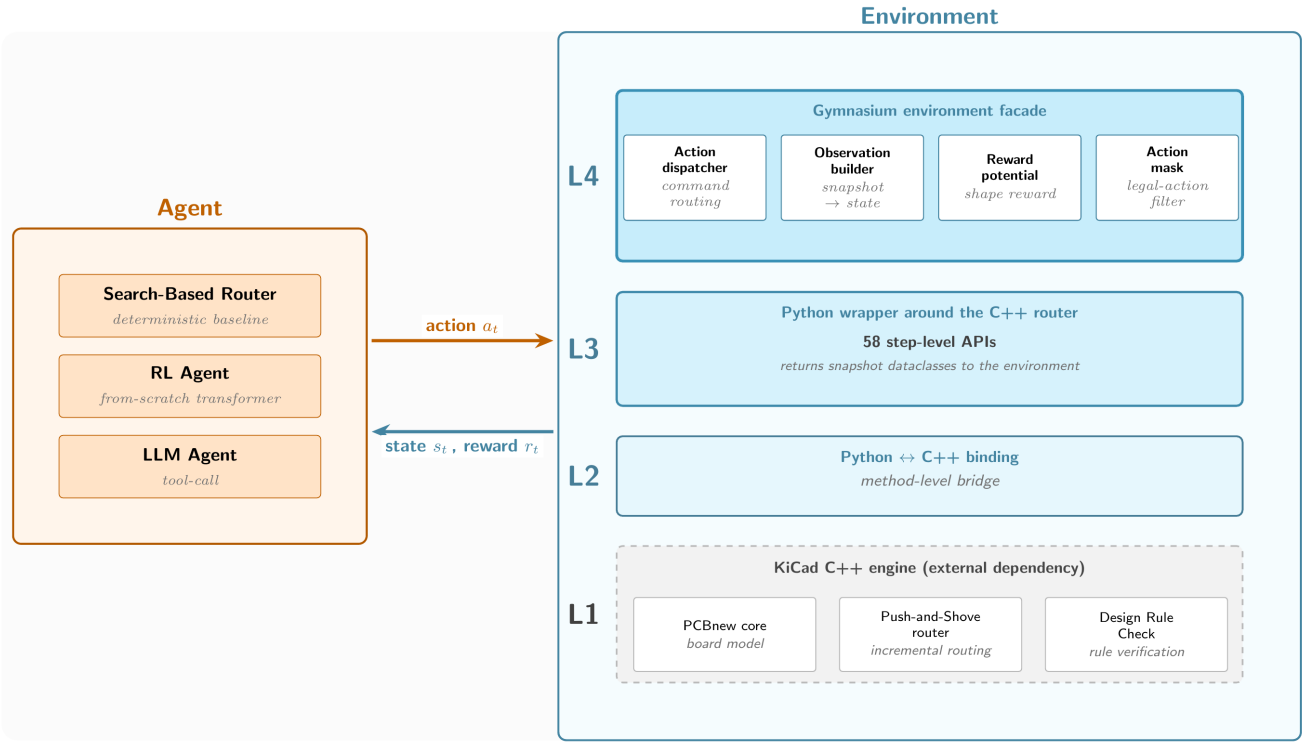


Figure 10: PCBWorld architecture. A reinforcement-learning view of the proposed environment. Three policy classes, a rule-based router as a deterministic baseline, an RL Agent built on a from-scratch Transformer, and an LLM Agent that issues tool calls, share a single Gymnasium environment through the standard action a_t and the state s_t, reward r_t signals. Inside the environment, our contribution spans L4 (the Gymnasium environment facade with four sub-modules: action dispatcher, observation builder, reward potential, and action mask), L3 (the Python wrapper exposing 58 step-level APIs), and L2 (the Python↔C++ binding); layer-box heights encode relative scope, while sub-boxes detail the internal components. L1 is KiCad's unmodified C++ engine, namely PCBnew core, Push-and-Shove router, and Design Rule Check, drawn with a dashed border as an external dependency.

exposed to the policies is documented in the main text under Section 3.2. The data contract above fixes the state, reward, and admissibility channels that the policy reads, while this hilevel action surface fixes the action space the policy emits, completing the MDP interface that L4 exposes upward.

D Mapping the API to MDP

The agent reaches KiCad's PNS router through a thin Python wrapper whose public surface exposes 58 step-level methods, partitioned by the wrapper source into 14 groups and mapped to 25 Action, 25 State, 3 Reward, and 5 Utility components of the MDP. Table 6 catalogues every method by MDP component and source-code group, and is the central artefact of this appendix. Four wrapper-only helpers are excluded: `close` is episode teardown, `get_native` returns a raw C++ handle for renderers, and `get_drc_result`, `get_board_snapshot` are dataclass aggregators whose underlying getters surface individually in the assignment rule below. Section C situates this wrapper as the L3 layer of the overall PCBWorld stack.

For a step-level method m at L3, component membership is fixed by which L4 call-sites reach m, not by what the method returns. The two subsections below organise the 58 APIs of Table 6: the Routing MDP collects the per-step State, Reward, Action, and transition components, while the miscellaneous group covers episode-level Utility methods and seven dual-tagged overlaps.

Group	*n*	Methods
***Action**: drive PCB routing through copper placement, parameter changes, and rework (25 methods).*		
Configuration	6	`set_routing_mode()`, `set_corner_mode()`, `set_track_width()`, `set_via_diameter()`, `set_via_drill()`, `reset_via_mode()`
Routing	9	`start_route()`, `move()`, `fix_route()`, `cancel_route()`, `finish()`, `undo_last_segment()`, `flip_posture()`, `toggle_via()`, `switch_layer()`
Dragging	3	`start_drag()`, `fix_drag()`, `cancel_drag()`
Rework	4	`delete_track_by_index()`, `delete_track_near()`, `delete_via_by_index()`, `delete_via_near()`
Refresh, *a/r*	1	`run_drc()`
Refresh, *a/u*	2	`build_connectivity()`, `set_design_rules()`
***State**: read the current PCB layout, routing session FSM, and design rules (25 methods).*		
Board meta	3	`get_board_bbox()`, `get_board_net_count()`, `get_copper_layer_count()`
Elements	7	`get_tracks()`, `get_vias()`, `get_pads()`, `get_points()`, `get_ratsnest()`, `get_board_outline()`, `get_net_names()`
Elements, *s/r*	3	`get_track_count()`, `get_via_count()`, `get_unrouted_count()`
Session	8	`get_router_state_code()`, `is_routing()`, `is_dragging()`, `is_placing_via()`, `get_current_layer()`, `get_route_head()`, `get_current_net_code()`, `get_routing_target()`
Design rules	2	`get_design_rules()`, `get_netclass_for_net()`
Snapshots	2	`get_board_meta()`, `get_routing_session_state()`
***Reward**: score routing quality through DRC violations and routing progress (3 methods).*		
Diagnostics	2	`get_drc_violation_count()`, `get_drc_violations()`
Diagnostics, *s/r*	1	`get_reward_snapshot()`
***Utility**: save the routed PCB and reset engine caches between episodes (5 methods).*		
DRC cache	1	`clear_drc_cache()`
I/O	1	`save()`
Provenance	3	`get_project_path()`, `was_project_loaded_from_file()`, `was_legacy_design_settings_loaded()`

Table 6: Step-level API inventory. The 58 APIs map to 25 Action, 25 State, 3 Reward, and 5 Utility methods, each further refined into one of 14 source-code groups. Dual-component rows append the secondary tag in italic after the group name, for example *a/r*; these seven rows carry component overlap forced by the underlying C++ binding. The Refresh group covers engine-cache refresh operations called at step or episode boundaries, which is why its rows are dual-tagged rather than purely Action.

D.1 Routing MDP

State. Methods that feed the observation builder, either directly or through the `BoardSnapshot` aggregator. The aggregator bundles seven read-only getters spanning board geometry, ratsnest, and metadata into a single dataclass, but is excluded from the 58-method inventory because its underlying getters appear individually. The Board meta, Elements, Session, Design rules, and Snapshots groups are State.

Reward. Methods that populate fields of the `RewardSnapshot` dataclass that the potential composer folds into $\Phi(s)$ in Section 3.2. The Diagnostics group exhausts this tag, with the `get_reward_snapshot` aggregator joining the two raw DRC counters.

Action. Methods that the per-step dispatcher invokes to mutate board or router state, covering reset cleanup, per-step configuration, and within-episode action handling. The Configuration, Routing, Dragging, and Rework groups fall here.

State transition. The transition kernel $P(s_{t+1} \mid s_t, a_t)$ is realised entirely by the unmodified C++ engine. An Action method submits the requested mutation, the engine applies, modifies, or rejects it according to its routing and DRC logic, and the L4 facade then harvests s_{t+1} from the resulting `BoardSnapshot` and `RoutingSessionState` while `RewardSnapshot` supplies r_t on the same C++ pass. Inadmissible actions are pre-filtered by the action mask reading `RoutingSessionState`, so the engine's reject path is defensive rather than routine, and no Python code computes the transition itself.

D.2 Miscellaneous

Utility. Methods reached only from reset or end-of-episode bookkeeping, never from the per-step decision path. The DRC cache, I/O, and Provenance groups are pure Utility.

Dual-tagged exceptions. The component assignment above admits seven dual-tagged exceptions, all forced by the underlying C++ binding rather than by taxonomic ambiguity. The three element-count getters and `get_reward_snapshot` share an s/r tag because one C++ pass populates both a State and a Reward field. `run_drc` carries a/r because its single invocation mutates the DRC cache and also refreshes the violation list the reward consumes. `build_connectivity` and `set_design_rules` carry a/u because they mutate engine caches only at episode boundaries and never commit copper. With these seven exceptions, 51 of 58 methods carry a single tag, so the per-component totals of 25 Action, 26 State, 7 Reward, and 7 Utility sum to 65 and exceed the unique count of 58 by exactly the dual-tagged rows.

E Visual Demonstration for Routing APIs

This appendix complements the API table of Section D with a visual reference for the 22 routing-related entries. Every figure shares a single template: a *scenario panel* on the left fixes the starting board and overlays a dashed plan for the intended route, an *action sequence* in the middle prints the literal `kicad_engine.py` call sequence, and *variant panels* on the right show the resulting copper under different parameter choices. Net colours are computed deterministically from the engine's `net_code` so each net keeps a consistent colour across figures: NET1 stays red, NET2 stays blue, and NET3 is rendered as the dashed plan. Layer is colour-coded with *Top* in red and *Btm* in blue, following the KiCad GUI convention. Coordinates use the KiCad board frame, with the origin at the top-left and the y-axis pointing downward, and all dimensions are given in millimetres. The visualisation covers twenty-one of the twenty-two APIs in five thematic groups; the remaining one, `undo_last_segment`, only mutates the in-flight head and never modifies committed copper, so a static figure cannot show its effect, and its semantics are verified by the unit test `test_undo_does_not_commit_tracks`.

Routing primitives. A routing session is bracketed by `start_route` and one of three completion calls: `fix_route`, `cancel_route`, or `finish`. Within the session, `move` re-positions the head while only `fix_route` commits a segment to copper. Fig. 11 visualizes this commit-versus-no-commit boundary across a four-step session, and Fig. 12 contrasts the three completion calls on a three-pad scenario. Two further session-time controls shape the geometry of the committed trace: `flip_posture` in Fig. 13 mirrors the L-shape orientation chosen for asymmetric endpoints, and the layer-transition pair `toggle_via`/`switch_layer` in Fig. 14 either drops a stand-alone via or inserts one automatically when the active layer changes.

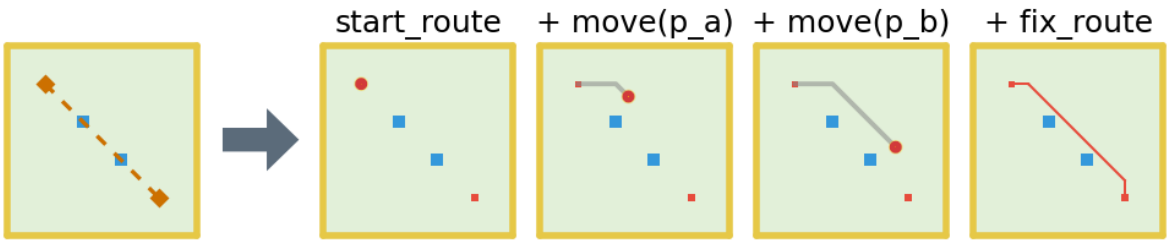


Figure 11: Virtual move versus physical commit. Within a routing session, `move` relocates the virtual head that reflects the designer's intent and draws only a transient hint-line preview without modifying the board. `fix_route` commits the current virtual path as a physical copper object in the database and converts the preview into a permanent trace.

Routing-mode controls. Two session-level switches choose how the router treats existing copper and how it draws corners. Fig. 15 contrasts the three strategies of `set_routing_mode`, namely *MarkObstacles*, *Shove*, and *Walkaround*, on a board pre-routed with two obstacle nets. Fig. 16 contrasts the two settings of `set_corner_mode`, 45° mitering against 90° rectilinear corners, on identical diagonal endpoints. The default `MITERED_45` mode is the recommended PCB practice, while the `MITERED_90` mode is exposed for layout-tool compatibility but is generally avoided in modern design.

Track and via geometry. Width and via geometry are exposed as session-time setters that override the design-rule defaults for the next committed segment or via. Fig. 17 sweeps `set_track_width` across three target widths on identical endpoints, the same control used in practice to differentiate power, clock, and signal nets. Fig. 18 factorizes via geometry into three independent calls, `set_via_diameter`, `set_via_drill`, and `reset_via_mode`, with a final reset variant verifying that the per-call override is not sticky.

Drag-to-modify. Once a track is committed, the engine permits non-destructive geometric modification through a drag handle on its midpoint. Fig. 19 shows the three primitives `start_drag`, `fix_drag`, and `cancel_drag` on a single pre-routed diagonal, contrasting the committed and aborted shapes side by side. The drag family complements the deletion primitives below by providing an in-place modification path that does not require removal followed by re-routing.

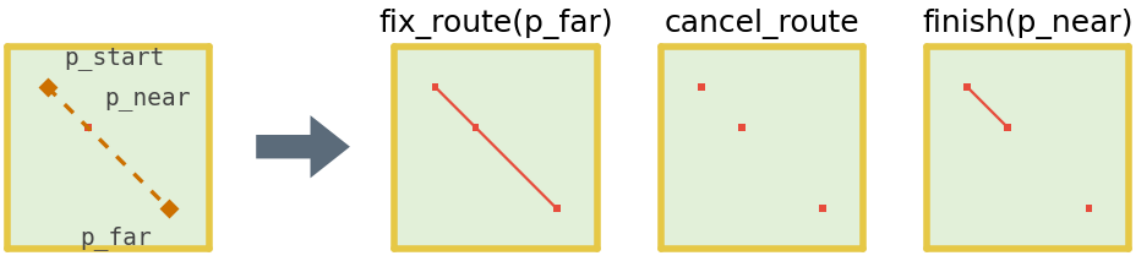


Figure 12: Session termination modes. Three distinct primitives close an open routing session. `fix_route` commits the trace at the designer-specified coordinate and ends the session. `cancel_route` discards every virtual change in the current session and restores the board to its pre-session state. `finish` hands control to the PNS router, which auto-completes the shortest remaining path to the nearest unrouted pad and ends the session.

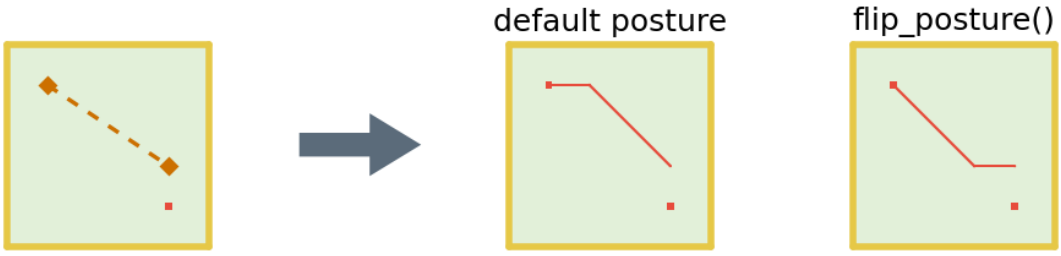


Figure 13: Corner posture control. When two endpoints are joined by an L-shape, the router must choose between leaving the start pad horizontally first or vertically first. `flip_posture` reverses this corner direction and allows the routing efficiency to be tuned dynamically within the available design space.

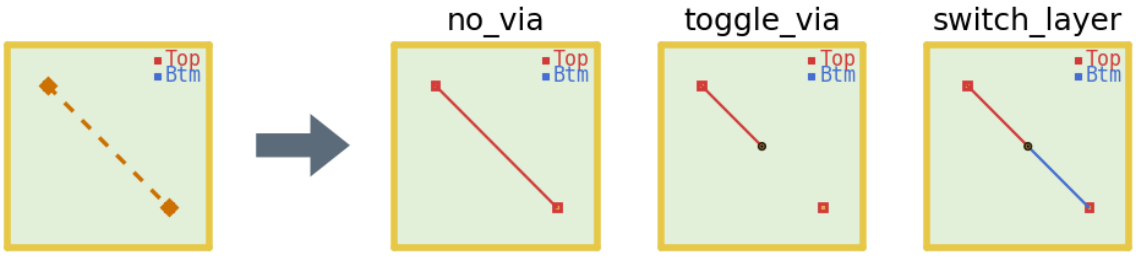


Figure 14: Via placement and layer transition. In a multilayer board, electrical connections between different layers are mediated by vias. `toggle_via` drops a vertical connection at the current location, while `switch_layer` changes the active routing layer and automatically inserts a transition via to connect the two layers.

Track and via removal. Tracks and vias can be removed either by integer position in the engine's internal list or by spatial coordinate within a tolerance. Fig. 20 contrasts `delete_track_by_index` and `delete_track_near` on a chained three-segment L, and Fig. 21 contrasts `delete_via_by_index` and `delete_via_near` on a multi-via trace. The two addressing modes are functionally equivalent at the engine level but differ in agent ergonomics: index addressing is convenient for replay or rollback patterns, while coordinate addressing is convenient when the agent identifies the target by spatial reasoning.

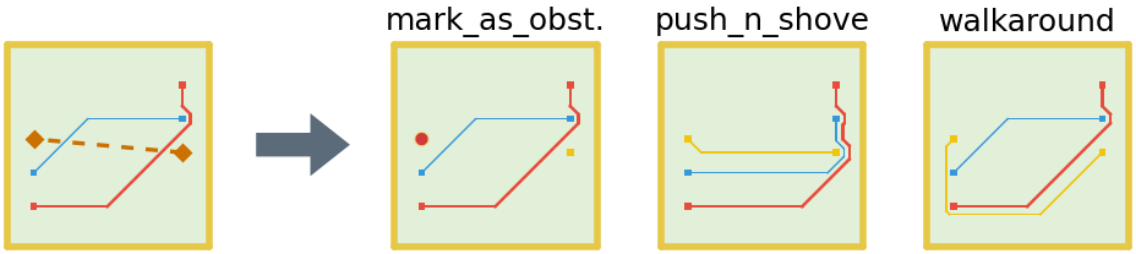


Figure 15: Obstacle interaction policies. `set_routing_mode` chooses one of three policies for interacting with existing copper. *MarkObstacles* performs collision detection and aborts trace generation when a clearance constraint would be violated. *Shove* pushes neighbouring tracks aside to dynamically clear space for the new path. *Walkaround* preserves the existing copper geometry and searches for an optimal detour around it.

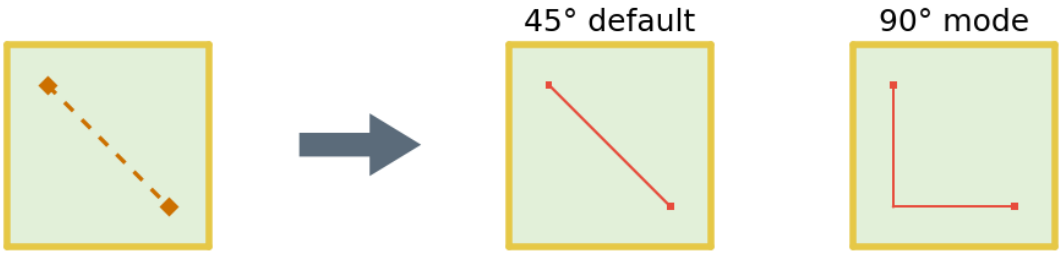


Figure 16: Trace corner geometry. `set_corner_mode` controls the angular format at trace corners. The industry-standard 45° miter, `MITERED_45`, is optimized to minimize signal loss and avoid manufacturing defects. The 90° rectilinear corner, `MITERED_90`, is reserved for cases that require special geometric alignment.

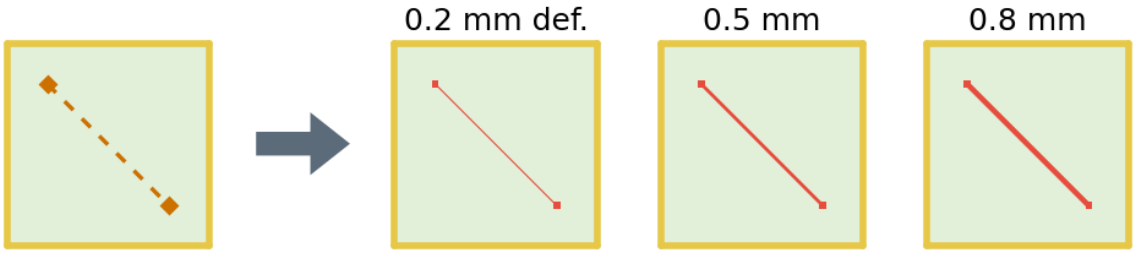


Figure 17: Trace width control. `set_track_width` explicitly sets the physical conductor width. Width is the key parameter that determines characteristic impedance and current-carrying capacity, distinguishing wide power traces that carry large currents from thin signal traces that carry fine signals.

F Design Rule Check Catalog

F.1 DRC Catalog and Severity Mapping

Source of truth. KiCad enumerates every Design Rule Check it can emit in `drc_item.h`: 62 distinct `DRCE_*` codes spanning copper-clearance, geometric (track width, annular ring, hole-to-hole), connectivity (unconnected, shorts, dangling), schematic-parity, courtyard, silkscreen, footprint-library and tuning (length / skew / diff-pair) checks. The default severity for each code is set in `board_design_settings.cpp`: every code starts at Error and a small set of overrides demotes 22 codes to Warning (mostly silkscreen, schematic and library checks

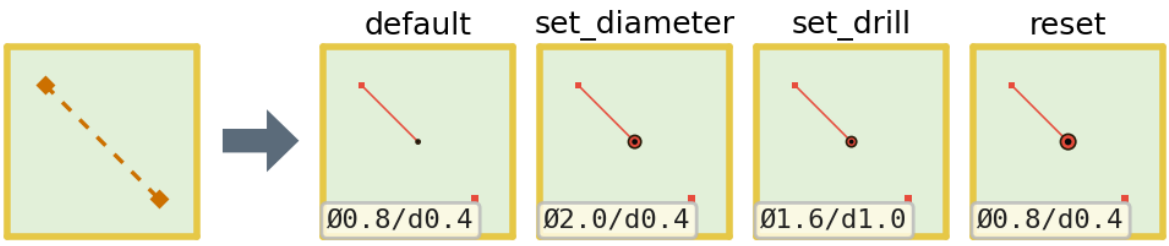


Figure 18: Via geometry and reset. The outer diameter and inner drill of a via are adjusted individually and independently of the design rules. `reset_via_mode` clears such temporary overrides and restores the system's default design standard, preserving design consistency.

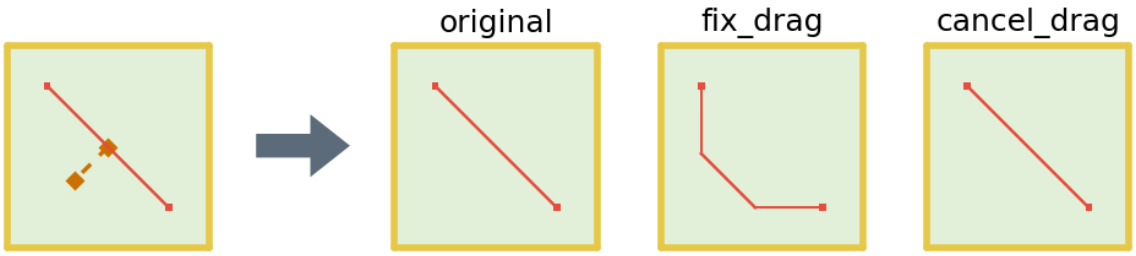


Figure 19: Trace drag-to-modify. Pulling the midpoint of an already committed trace allows its geometry to be reshaped flexibly. This is a non-destructive editing path used when local route optimization is needed without disturbing the existing connectivity.

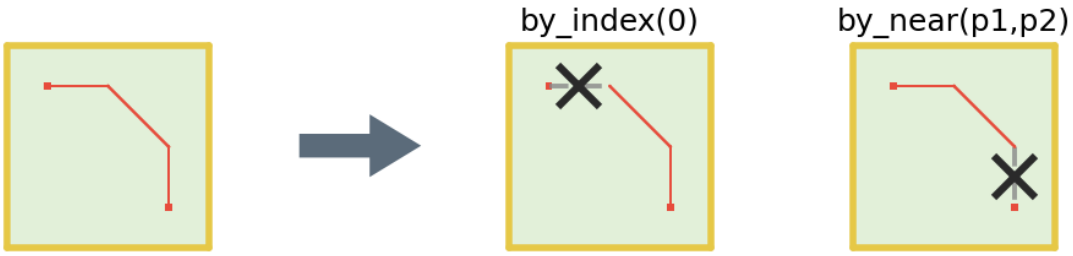


Figure 20: Track deletion methods. A track object can be identified and removed by two complementary logical approaches. `delete_track_by_index` addresses the engine's internal list by integer position to identify the target precisely. `delete_track_near` addresses the same target by physical coordinate, providing an intuitive interface for an agent that reasons about spatial layout.

plus the dangling/colocated-hole pair) and 5 codes to Ignore (courtyard membership and footprint-filter/type checks), leaving 35 stock error-level checks.

Severity surface seen by our environment. Our pipeline never edits the per-code severity table – KiCad's defaults are used verbatim – and collapses the resulting violation list into a two-way severity surface:

- **Error.** Any violation whose KiCad severity is Error.
- **Other.** Every warning or ignored code: silkscreen and text overlaps, courtyard membership, library/schematic parity, dangling tracks and vias, colocated holes, copper slivers, isolated copper, padstack questions, and so on. These never enter the penalty or the headline DRV count.

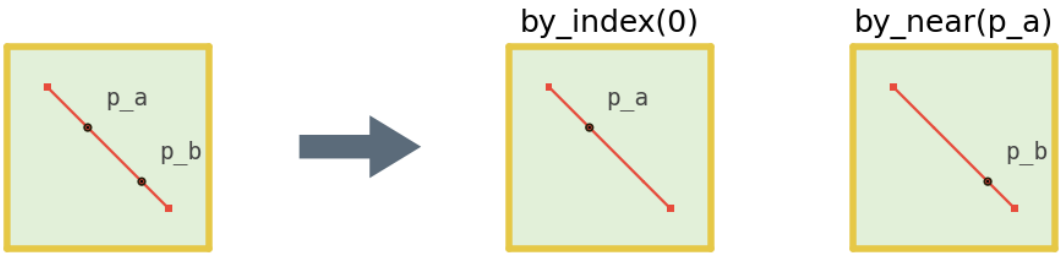


Figure 21: Via deletion methods. By the same logic as track removal, the vertical connection object is selectively removed either by engine index or by spatial coordinate.

Both the reward penalty and the evaluation DRV therefore count the same 35 stock error-level checks. The full enumeration with our classification is in Tables 7–8; the count-to-penalty mapping is in Table 9.

The catalog splits naturally into routing-relevant codes (the agent's actions can produce or fix them) and non-routing codes (footprint / silk / text / library / schematic / courtyard / zone-fill issues that the agent has no actuators for). Table 7 lists the routing-relevant subset, and Table 8 the rest. In both tables *Default* is the KiCad stock severity and *Env* is the bucket that drives the reward penalty and the DRV count (E = Error; W = Warning; I = Ignore); since the defaults are used verbatim, Env coincides with Default for every code.

Table 7: Routing-relevant DRC codes – the agent's actions can directly produce or clean these.

ID	DRCE_* name	Settings key	Message	Default	Env
1	UNCONNECTED_ITEMS	unconnected_items	Missing connection between items	E	E
2	SHORTING_ITEMS	shorting_items	Items shorting two nets	E	E
3	ALLOWED_ITEMS	items_not_allowed	Items not allowed (custom rule)	E	E
5	CLEARANCE	clearance	Clearance violation	E	E
6	CREEPAGE	creepage	Creepage violation	E	E
7	TRACKS_CROSSING	tracks_crossing	Tracks crossing	E	E
8	EDGE_CLEARANCE	copper_edge_clearance	Board edge clearance	E	E
12	DANGLING_VIA	via_dangling	Via not / partly connected	W	W
13	DANGLING_TRACK	track_dangling	Track has unconnected end	W	W
16	HOLE_CLEARANCE	hole_clearance	Hole clearance violation	E	E
17	TRACK_WIDTH	track_width	Track width out of range	E	E
18	TRACK_ANGLE	track_angle	Track angle out of range	E	E
19	TRACK_SEGMENT_LENGTH	track_segment_length	Track segment length out of range	E	E
20	ANNULAR_WIDTH	annular_width	Annular ring too small	E	E
22	DRILL_OUT_OF_RANGE	drill_out_of_range	Hole size out of range	E	E
23	VIA_DIAMETER	via_diameter	Via diameter out of range	E	E
26	MICROVIA_DRILL_OUT_OF_RANGE	microvia_drill_out_of_range	Microvia hole out of range	E	E
32	DISABLED_LAYER_ITEM	item_on_disabled_layer	Item on disabled copper layer	E	E
37	NET_CONFLICT	net_conflict	Pad net mismatches schematic	W	W
46	ASSERTION_FAILURE	assertion_failure	Custom-rule assertion	E	E
47	GENERIC_WARNING	generic_warning	Custom-rule warning	E	E
48	GENERIC_ERROR	generic_error	Custom-rule error	E	E
50	SOLDERMASK_BRIDGE	solder_mask_bridge	Solder-mask bridges different nets	E	E
56	LENGTH_OUT_OF_RANGE	length_out_of_range	Track length out of range	E	E
57	SKEW_OUT_OF_RANGE	skew_out_of_range	Skew between tracks out of range	E	E
58	VIA_COUNT_OUT_OF_RANGE	too_many_vias	Too many / few vias on connection	E	E
59	DIFF_PAIR_GAP_OUT_OF_RANGE	diff_pair_gap_out_of_range	Diff-pair gap out of range	E	E
60	DIFF_PAIR_UNCOUPLED_LENGTH_TOO_LONG	diff_pair_uncoupled_length_too_long	Diff-pair uncoupled too long	E	E

Count-to-penalty mapping. The reward potential and the eval-time DRV count both count error-level violations, as follows.

Why this subset. Of the 62 catalogued codes, the 28 in Table 7 are routing-relevant in the sense that the agent's actions can directly produce or fix them; the rest (Table 8) concern schematic parity, library state, courtyards, silkscreen and text, which the agent has no actuators for. Of

Table 8: Non-routing DRC codes – footprint, silkscreen, text, library, schematic-parity, courtyard and zone-fill checks. Listed for completeness; none of these enter our reward penalty or DRV count.

ID	DRCE_* name	Settings key	Message	Default	Env
4	TEXT_ON_EDGECUTS	text_on_edge_cuts	Text on Edge.Cuts layer	E	E
9	ZONES_INTERSECT	zones_intersect	Copper zones intersect	E	E
10	ISOLATED_COPPER	isolated_copper	Isolated copper fill	W	W
11	STARVED_THERMAL	starved_thermal	Thermal relief incomplete	E	E
14	DRILLED_HOLES_TOO_CLOSE	hole_to_hole	Drilled hole too close to other	W	W
15	DRILLED_HOLES_COLOCATED	holes_co_located	Drilled holes co-located	W	W
21	CONNECTION_WIDTH	connection_width	Copper connection too narrow	W	W
24	PADSTACK	padstack	Padstack questionable	W	W
25	PADSTACK_INVALID	padstack_invalid	Padstack not valid	E	E
27	OVERLAPPING_FOOTPRINTS	courtyards_overlap	Courtyards overlap	E	E
28	MISSING_COURTYARD	missing_courtyard	Footprint has no courtyard	I	I
29	MALFORMED_COURTYARD	malformed_courtyard	Malformed courtyard	E	E
30	PTH_IN_COURTYARD	pth_inside_courtyard	PTH inside courtyard	I	I
31	NPTH_IN_COURTYARD	npth_inside_courtyard	NPTH inside courtyard	I	I
33	INVALID_OUTLINE	invalid_outline	Malformed board outline	E	E
34	MISSING_FOOTPRINT	missing_footprint	Missing footprint	W	W
35	DUPLICATE_FOOTPRINT	duplicate_footprints	Duplicate footprints	W	W
36	EXTRA_FOOTPRINT	extra_footprint	Extra footprint	W	W
38	SCHEMATIC_PARITY	footprint_symbol_mismatch	Footprint attrs don't match symbol	W	W
39	FOOTPRINT_FILTERS	footprint_filters_mismatch	Footprint outside symbol filters	I	I
40	FOOTPRINT_TYPE_MISMATCH	footprint_type_mismatch	Footprint type vs. pads mismatch	I	I
41	LIB_FOOTPRINT_ISSUES	lib_footprint_issues	Footprint not in libraries	W	W
42	LIB_FOOTPRINT_MISMATCH	lib_footprint_mismatch	Doesn't match library copy	W	W
43	PAD_TH_WITH_NO_HOLE	through_hole_pad_without_hole	Through-hole pad without hole	E	E
44	FOOTPRINT	footprint	Footprint not valid	E	E
45	UNRESOLVED_VARIABLE	unresolved_variable	Unresolved text variable	E	E
49	COPPER_SLIVER	copper_sliver	Copper sliver	W	W
51	SILK_CLEARANCE	silk_over_copper	Silkscreen clipped by mask	W	W
52	SILK_EDGE_CLEARANCE	silk_edge_clearance	Silkscreen clipped by board edge	W	W
53	TEXT_HEIGHT	text_height	Text height out of range	W	W
54	TEXT_THICKNESS	text_thickness	Text thickness out of range	W	W
55	OVERLAPPING_SILK	silk_overlap	Silkscreen overlap	W	W
61	MIRRORED_TEXT_ON_FRONT_LAYER	mirrored_text_on_front_layer	Mirrored text on front layer	W	W
62	NONMIRRORED_TEXT_ON_BACK_LAYER	nonmirrored_text_on_back_layer	Non-mirrored text on back layer	W	W

Table 9: How each violation bucket enters the DRC penalty f_d used in the shaping potential Φ (Equation (2)) and the evaluation count. The training penalty f_d takes the log-per-net shape of Equation (5); here x is the number of distinct nets carrying at least one in-bucket violation (breadth) and x_i is the number of in-bucket violations on net i (depth). The 3:1 ratio is the breadth-vs-depth weighting, not error-vs-warning.

Bucket	Source	Reward penalty f_d	Eval DRV
Error	KiCad Error severity	counted (see below)	+1
Warning	KiCad Warning severity	0	0
Ignore	KiCad Ignore severity	0	0

* both sums are taken over the Error bucket (drc_errors). The CP metric reported in the benchmark is the joint indicator $\mathbf{1}\{\text{routability} = 1\} \cdot \mathbf{1}\{\texttt{drc\_errors} = 0\}$. The penalty f_d enters Φ with a minus sign (Equation (2)), so larger f_d means smaller reward.

the 28 routing-relevant codes, all but the dangling/net-conflict trio carry stock Error severity and are therefore counted; the trio are stock warnings and, like all other warnings, stay outside the penalty and the DRV count. The non-routing warnings are deliberately left out of the penalty so that training signal is concentrated on outcomes the policy can affect, and left out of the headline DRV count so that comparable boards routed by different agents are not penalised for upstream artefacts they share.

Mapping LLM failure modes to DRC buckets. Table 10 gives representative examples of how LLM routing failures surface in the DRC reports, or as non-DRC outcomes such as mask reject, parse fail, or no-effect steps. Failures that never reach the engine appear with a *none* entry in the DRC column.

Table 10: Representative mapping from LLM failure modes to DRC buckets. Rows whose failures never reach the engine are assigned to the non-DRC column.

Failure mode	Primary DRC code(s)	Secondary / non-DRC	Bucket
(via on pad)	ANNULAR_WIDTH, HOLE_CLEARANCE	—	E
(wrong-layer endpoint)	DANGLING_TRACK	no-effect step	W
(cross-net contact)	SHORTING_ITEMS, CLEARANCE	—	E
(stranded segment)	DANGLING_TRACK, DANGLING_VIA	—	W
(detour over board edge)	EDGE_CLEARANCE	—	E
(invalid coordinate / phantom waypoint)	—	parse fail / mask reject	none
(repeated identical attempt)	—	rejection streak	none

G Reward potential and DRC penalty

This section instantiates the shaping potential Φ defined in §3.2, which we restate here:

$$\Phi(s) \;=\; -\big(f_d(n_{\mathrm{drv}}(s)) + \lambda_w\,\ell(s) + \lambda_v\,n_{\mathrm{via}}(s)\big), \tag{2}$$

and specifies the concrete choices used by our learned baselines: the DRC counting convention n_{drv}, the concave shape of f_d, the quality-term weights (λ_w, λ_v), and the per-step form of the reward.

DRC counting convention. Let $\mathcal{V}(s)$ be the multiset of DRC violations whose KiCad default severity is Error, i.e., the 35 stock error-level checks catalogued in Appendix F. We take $n_{\mathrm{drv}}(s) = |\mathcal{V}(s)|$, and additionally decompose it per offending net i:

$$x_i(s) \;=\; \big|\{v \in \mathcal{V}(s) \,:\, \mathrm{net}(v) = i\}\big| \qquad (\textit{depth}\text{: per-net violation count}), \tag{3}$$

$$x(s) \;=\; \big|\{i \,:\, x_i(s) > 0\}\big| \qquad (\textit{breadth}\text{: number of dirtied nets}), \tag{4}$$

so that $n_{\mathrm{drv}}(s) = \sum_i x_i(s)$.

Shape of the DRC penalty f_d. Real industrial quality requires producing boards with DRV = 0: a board with one residual violation is no more shippable than a board with ten, so reward improvements only matter insofar as they push the policy toward the clean-board boundary. We therefore adopt a log-concave shape for f_d, which assigns a large marginal penalty to the *first* violation on a clean net and rapidly diminishing marginal penalty to each additional violation. This concentrates the learning signal on closing out the last violations—where industrial quality is decided—rather than on incremental reductions on already dirty boards. Concretely, we instantiate the concave penalty in Equation (2) as a log shape that is separately concave in breadth and depth and unbounded above:

$$f_d(n_{\mathrm{drv}}(s)) \;=\; s_{\mathrm{agg}}\,\ln\!\Big(1 + \tfrac{x(s)}{o}\Big) \;+\; s_{\mathrm{pn}} \sum_{i:\,x_i(s)>0} \ln\!\Big(1 + \tfrac{x_i(s)}{o}\Big), \tag{5}$$

with hyperparameters $(s_{\mathrm{agg}}, s_{\mathrm{pn}}, o) = (3,\ 1,\ 2)$. The first (aggregate) term penalizes *breadth*—how many distinct nets the policy has dirtied—while the per-net sum penalizes *depth*—how many violations pile up on each individual net. The shared offset o acts as the log-curve knee, so a single new violation on a clean net contributes a bounded $\ln(3/2)$ even when many other nets are already dirty, preventing the gradient from collapsing on heavily violated boards. The 3:1 ratio biases the policy toward routing fewer dirty nets rather than minimizing violations on a single dirty net.

Quality-term weights during training. Unless otherwise noted, the reported policies are trained with the default quality weights $(\lambda_w, \lambda_v) = (0.002, 0.1)$, so the full potential of Equation (2)—including the wirelength and via terms—is optimized during training. The controllability study (§5.5) sweeps these weights over a 3×3 grid, $\lambda_w \in \{0, 0.001, 0.002\}$ and $\lambda_v \in \{0, 0.05, 0.1\}$; its feasibility-only corner $(\lambda_w, \lambda_v) = (0, 0)$ trains against $\Phi(s) = -f_d(n_{\mathrm{drv}}(s))$ alone.

Severity-mode coupling. The reward penalty f_d, the DRC tokens emitted into the agent's state (Appendix B), and the headline evaluation DRV (Appendix H) all count the same error-level violations, so the agent is never trained against violations it cannot observe, nor evaluated on a different set than it was rewarded for.

H Evaluation Metrics

We evaluate PCB routing quality from multiple complementary angles, combining board-level aggregates with sample-level statistics. For each board $b \in \mathcal{B}$, we generate k independent routing samples under the same budget; we use $k = 5$ throughout. We denote the i-th routed sample for board b as $s_{b,i}$ for $i \in \{1, \dots, k\}$, and reserve $s_{b,0}$ for the bare-board state before any routing. We use $\mathbf{1}[\cdot]$ for the indicator function.

Potential Gain (Pot.↑). Our central routing-quality measure is the *potential gain* from the bare-board state. Let $\Phi(s)$ be the board-level potential function introduced in Section 3.2, which jointly penalizes design-rule violations, wire length, and via count (and thereby captures net-level routing quality; see Appendix G). For a routed sample $s_{b,i}$, the potential gain is

$$\Delta\Phi(s_{b,i}) = \Phi(s_{b,i}) - \Phi(s_{b,0}),$$

which measures how much routing progress has been made relative to the bare-board state $s_{b,0}$: a larger $\Delta\Phi$ reflects a cleaner, shorter, and more complete routing solution. As a reported metric, Pot. is the potential gain of the per-board selected sample (defined next), averaged over boards:

$$\text{Pot.} = \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \Delta\Phi(s_b^\star).$$

Sample Selection. Potential gain also defines how a single representative rollout is chosen per board, which all subsequent sample-level metrics are computed on. For each board b we select the routed sample with the largest potential gain,

$$I_b = \arg\max_{i \in \{1,\dots,k\}} \Delta\Phi(s_{b,i}), \qquad s_b^\star = s_{b,I_b},$$

and collect the selected pairs into

$$\mathcal{B}^\star = \left\{ (b, s_b^\star) : b \in \mathcal{B} \right\}.$$

Unless stated otherwise, every per-board diagnostic metric defined below is reported on this potential-selected sample $s_b^\star$, so that all metrics describe the same representative rollout.

Routability (Rout.↑). We measure connectivity using KiCad's *ratsnest* connections: the set of pad-to-pad connections that the netlist mandates but that have not yet been physically routed, rendered in KiCad as the residual "rats" lines. For a sample $s_{b,i}$, let $R(s_{b,i})$ denote the number of remaining ratsnest edges. Since $s_{b,0}$ denotes the bare-board state before any routing, the ratsnest-based routability of $s_{b,i}$ is

$$\text{rout}(s_{b,i}) = \frac{R(s_{b,0}) - R(s_{b,i})}{R(s_{b,0})} \le 1,$$

i.e., the fraction of required ratsnest connections resolved relative to the bare-board state. The value is negative when spurious floating traces introduce ratsnest edges beyond those of the bare board, as under engine-free LLM generation in Figure 7. A sample is *fully routed* iff $R(s_{b,i}) = 0$, equivalently $\text{rout}(s_{b,i}) = 1$. We report Rout. on the per-board best samples selected by potential gain:

$$\text{Rout.} = \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \text{rout}(s_b^\star) = \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \frac{R(s_{b,0}) - R(s_b^\star)}{R(s_{b,0})}.$$

Clean Pass (CP↑). Connectivity alone is insufficient as a measure of manufacturability. We therefore define a stricter criterion: a sample $s_{b,i}$ is *clean* iff (i) the residual ratsnest is empty, $R(s_{b,i}) = 0$, and (ii) the number of KiCad error-level design-rule violations is zero, $\text{drv}(s_{b,i}) = 0$. Consistent with the other metrics, we evaluate cleanliness on the per-board selected sample $s_b^\star$ (the rollout of largest potential gain):

$$\text{CP} \;=\; \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \mathbf{1}\left[\text{rout}(s_b^\star) = 1 \;\wedge\; \text{drv}(s_b^\star) = 0 \right].$$

This is the closest proxy in our evaluation to "ready-to-fabricate" output. We draw $k = 5$ rollouts per board and report CP on the selected sample $s_b^\star$.

DRV count (Design-Rule Violation, DRV↓). We quantify the severity of rule violations by the KiCad error count $\text{drv}(s_{b,i}) = \texttt{drc\_errors}(s_{b,i})$. Following the main-text convention in Section 4.3, we report DRV on the per-board best samples selected by potential gain:

$$\text{DRV} \;=\; \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \text{drv}(s_b^\star).$$

Physical Cost Metrics (WL↓, Via↓). We report two physical cost metrics on the per-board best samples selected by potential gain. WL is the total routed wire length in millimeters, summed over all nets, and Via is the number of vias placed by the router. Both are common secondary objectives in PCB routing, as shorter traces and fewer vias generally reduce routing cost and manufacturing complexity. They are aggregated over $\mathcal{B}^\star$ as

$$\text{WL} = \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \text{wl}(s_b^\star), \qquad \text{Via} = \frac{1}{|\mathcal{B}^\star|} \sum_{(b, s_b^\star) \in \mathcal{B}^\star} \text{via}(s_b^\star).$$

Wallclock time (Time↓). We report wall-clock routing time per sample (in seconds), measured from the first API call of a rollout until the agent terminates that rollout. Unlike DRV, WL, and Via, the runtime is computed over *all* k samples on *every* board, regardless of routability: we take the sample-mean per board and then the mean across boards,

$$\text{Time} \;=\; \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}}\frac{1}{k}\sum_{i=1}^{k}\text{time}(s_{b,i}).$$

Restricting to fully-routed samples would bias the cost in favor of methods that abort early on hard boards; the formulation above charges every rollout for the time it actually consumed. For stochastic methods (Freerouting and our RL agents) we report the seed mean of this quantity; the LLM agents are evaluated with a single run, and for deterministic baselines (OrthoRoute and KRT) a single value is reported.

Parse failure rate (Parse-fail↓). Parse-fail applies to the LLM agents, whose model responses are funnelled through a strict parser before they can reach the engine; a rejected response leaves the board unchanged (Table 12). For a sample $s_{b,i}$, let $n(s_{b,i})$ be the number of model responses issued in the rollout and $n_{\text{rej}}(s_{b,i})$ the number rejected by the parser. Consistent with the other diagnostics, we report the rejected fraction on the per-board selected samples,

$$\text{Parse-fail} \;=\; \frac{1}{|\mathcal{B}^\star|}\sum_{(b,s_b^\star)\in\mathcal{B}^\star}\frac{n_{\text{rej}}(s_b^\star)}{n(s_b^\star)}.$$

Methods without a text interface (the RL agents and the rule-based routers) have no parse step, so the metric is not reported for them.

I Experimental Setup and Model Serving

I.1 Proprietary LLM and Open-Source LLM

We benchmark the LLM agent on three OpenAI cloud-API models and one open-weights Qwen model served through Together AI. Table 11 summarises the deployment configuration for each: serving infrastructure, hardware when exposed by the provider, weight quantisation as served, reasoning mode ("hidden" for closed models with internal chain-of-thought, `think-off` for the Qwen variant where the explicit reasoning channel is disabled), the per-turn token budget for generation, and the datasets each model is evaluated on (D2, D3-A, and D3-B).

Table 11: Deployment configuration of every LLM in the benchmark. "–" = not applicable / not exposed by the provider.

Model	Infrastructure	Hardware	Quant.	Thinking	Max tok.	Datasets
gpt-5.4	API (OpenAI)	cloud	–	hidden	256	D2, D3-A, D3-B
gpt-5.4-mini	API (OpenAI)	cloud	–	hidden	256	D2, D3-A, D3-B
gpt-5.4-nano	API (OpenAI)	cloud	–	hidden	256	D2, D3-A, D3-B
Qwen3.5-397B-A17B	API (Together)	–	FP4	think-off	512	D2, D3-A, D3-B

J Implementation Detail

J.1 LLM Wrapper

The wrapper builds each prompt by instantiating a fixed template with runtime values. We partition the prompt into a static *system* message and a dynamic per-turn *user* message (refer Section M.1). The system message contains episode-invariant content: the agent role and priority, routing guidelines, the response-format contract, the board-state schema description, and the `board_static` block (footprints, pads, nets, and design rules). By placing these invariant fields in the system message, the wrapper keeps the prompt prefix stable across steps, making the prompt layout compatible with prefix caching.

The user message is regenerated at every environment step from the current routing state and bookkeeping variables. It contains the 1-indexed step counter, the freshly serialized observation obtained by concatenating the `routing_geometry` and `router_head` blocks, the number of accepted actions so far, a rolling window of the recent action history, an optional rejection-streak line, and the list of action verbs allowed by the current router-phase mask. Coordinates in the dynamic observation are rounded to three decimal places in millimeters. The rejection slot is empty in the nominal case; after a parse failure or mask veto, consecutive identical rejections are collapsed into a single "`(rejected ×N)`" line and cleared once the next valid action is accepted.

Every model response is funnelled through a strict parser before it can touch the router. The parser requires a single `<think></think><action></action>` pair, a known verb, the right arity, and ASCII-only content; failures fall back to an explicit IDLE action (action index 6). After parsing, the action index is intersected with the current phase's mask (NET_SELECT → START_ROUTE → ROUTING); a mask miss is also rerouted to `idle` and the offending body is pushed onto the rejection streak so the next turn's user prompt can surface it. Successfully dispatched actions that nonetheless leave the unrouted-pin count unchanged are kept but tagged `[no effect]` in history, and any preceding `start_route` that they exposed as wasted is retroactively re-tagged. The reward consequences of each class are summarised in Table 12.

Table 12: Parser and action-mask fallback policy. The *Action taken* column is what the underlying KiCad router actually executes.

Failure mode	Trigger	Action taken
Missing <action> tag	no/empty action block	IDLE (idx 6)
Malformed body	unknown verb, wrong arity, conv. error	IDLE (idx 6)
Bad <think> block	missing/duplicated reasoning channel	IDLE (idx 6)
Non-ASCII content	e.g. CJK characters in output	IDLE (idx 6)
Mask reject	valid parse, disallowed in current phase	IDLE (idx 6)
No-effect step	dispatched but unrouted unchanged	as emitted, tagged

J.2 RL Wrapper

The RL wrapper turns the state dictionary of §3.2 into a batched token sequence and exposes the action space of §3.2 as a typed autoregressive head over the engine-provided candidate set.

Tokenization. Each geometric or structural object becomes a single token. We support fourteen entity types: BOARD, EDGE, NET, PAD, TRACK, VIA, RAT, HEAD, four CAND_* variants for the action-time candidate pool, and DRC_VIOLATION. For a token i of type $\tau(i)$ with raw feature vector $\mathbf{f}_i$, the token embedding is the sum of an entity-type embedding and a per-type linear projection of its features:

$$\mathbf{x}_i \;=\; \mathbf{e}_{\tau(i)} \;+\; \mathbf{W}_{\tau(i)}\,\mathbf{f}_i.$$

Coordinates and dimensions are first normalized by the board center and a reference scale quantized onto a predefined discrete set, so identical physical quantities yield identical tokens regardless of board size. The normalized continuous fields (point coordinates, segment widths, via diameters) then pass through a sin/cos Fourier feature map with $n_{\text{freq}} = 32$ and geometric base 1.20; copper layers are encoded as the (dist_top, dist_bot) pair. TRACK tokens use symmetric endpoint pooling so that swapping the two endpoints yields the same embedding. Every geometric token also receives the head-relative distance to the current routing head as an inductive feature. A learned slot embedding distinguishes net-internal sub-objects, and the full sequence is finally normalized with LayerNorm. With $d_{\text{model}} = 128$ and $n_{\text{layers}} = 4$, the resulting flat sequence preserves the native object hierarchy of the state dictionary while remaining permutation-equivariant within each entity group.

Autoregressive action head. At step t, the action a_t of §3.2 is decoded as a triple $(\alpha,\ k,\ m)$ comprising an action type α, a pointer index k into the candidate set, and a routing mode m. We follow the slot table in Table 13: net_select and start_route use only a pointer; make_line and make_via use both pointer and mode; finish uses only mode; net_end uses neither. Decoding proceeds in two Transformer passes. In pass one the state sequence is extended with a single SOD token; the action-type logits are produced by a tied action_type_head embedding, sampled, and re-injected as a typed token to obtain a hidden state $\mathbf{h}_\alpha$. The pointer distribution scores $\mathbf{h}_\alpha$ against the per-token states of the eligible candidate slice (net tokens for net_select, candidate tokens otherwise) by scaled dot-product. In pass two the chosen candidate's state token is re-emitted as point_tok to obtain $\mathbf{h}_k$; the routing-mode distribution then dot-products $\mathbf{h}_k$ against the routing-mode embedding table reused from the tokenizer vocabulary.

Candidate pool. The candidate set is built per-step from the active net. It contains the net's pads (thru-hole pads expanded to one entry per copper layer), the endpoints of its already-drawn tracks, its via centers, and an eight-way directional grid of 0.5 mm offsets around the current head (a four-way grid snapped to the underlying lattice when running a D1 grid instance). Candidates are deduplicated by (x, y, layer) and truncated to 64 slots. Pointer selection therefore operates on a finite pool emitted by the engine; out-of-pool indices are masked to $-\infty$ before sampling.

Masking and stability. The same FSM-based action mask of §3.2 (net_select $\rightarrow$ start_route $\rightarrow$ routing) is applied to action-type logits; per-net pointer masks block out-of-scope candidates, and an additional pointer mask excludes the freshly started route's own coordinate across all layers, preventing self-loops and the resulting performance collapse; routing-mode masks honor the configured rule set. Action-type and pointer logits are passed through $10 \cdot \tanh(\cdot)$ to prevent early-training logit blow-up. Unused slots in the emitted action triple are set to $-\inf$, so every emitted action is syntactically complete and lies inside the engine's valid-action set by construction.

K Hyperparameters and compute

Table 15 reports the optimization and architecture settings needed to reproduce the learned baselines for the D1 grid-size/action-abstraction scalability experiment.

K.1 Main-result RL policy hyperparameters

The PPO, PPO (terminal), and GRPO entries of Table 3 all train the same decoder-only Transformer policy from scratch on D2-train, selecting checkpoints on D2-valid; the policy is then evaluated by rollout on the held-out test boards of both D2-test and the zero-shot D3 open-source

Action type	Needs pointer	Needs mode
`net_select`	✓	
`start_route`	✓	
`net_end`		
`make_line`	✓	✓
`make_via`	✓	✓
`finish`		✓
`idle`		

Table 13: Slot usage per action type. Unused slots are emitted as -1 in the action triple and contribute no log-prob term.

Table 14: Configuration of the main-result RL policy (trained on D2-train; evaluated on D2-test and zero-shot D3 test boards), shared by the PPO / PPO (terminal) / GRPO entries of Table 3.

Policy network (shared)	
Architecture	decoder-only Transformer
d_{model} / layers / heads / FFN	128 / 4 / 8 / 512
Coordinate encoding	Fourier (n_{freq}=32), MLP width 128
Activation / residual	GELU / ReZero
Action / value head	pointer network (logit clip 10) / MLP critic (PPO only)
Optimization (shared)	
Optimizer	AdamW (ϵ=10^{-5}, weight decay 10^{-4})
Learning rate	10^{-4} (20-iteration warmup)
Discount γ / GAE λ	0.995 / 0.95
PPO clip ϵ	0.2
Entropy / value coef.	0.01 / 0.5
Max gradient norm	0.5
Batch size / update epochs	256 / 4
Parallel envs / rollout horizon	32 / 512
Episode step limit	256
Reward (shared)	
Wirelength weight λ_w / via weight λ_v	0.002 / 0.1
DRC penalty f_d	log-per-net, error-level violations
Per-variant	
PPO	dense per-step reward, 300 iterations
PPO (terminal)	sparse terminal reward, 300 iterations
GRPO	sparse reward, group size 16, 1800 iterations
Seeds / hardware	42–45 / 1× NVIDIA L40 48 GB

set. The three variants share the architecture, optimizer, and reward weights (Table 14), and differ only in the reward form (per-step dense vs. terminal sparse) and the number of training iterations.

K.2 D1 grid-size scalability learned-policy hyperparameters.

The D1 scalability experiment compares PPO against grid-action A2C (Jumanji) and Sable on matched Connector-v2 tasks: PPO consumes KiCad board files, while Jumanji/Sable consume fixed Connector-v2 NPZ instances exported from the same splits. Table 15 lists the training settings that materially affect the RL results, with one hyperparameter family per row. Final reporting evaluates each selected checkpoint on the exact 128 held-out test boards per grid and seed, with five rollouts per board and one selected rollout retained before aggregation. For the grid-action baselines, training uses `train.npz`, training-time validation uses `val.npz`, and the final reported metric is recomputed on `test.npz`. This split policy is important because the Connector-v2 NPZ files are exported from the same generated KiCad board splits used by the PPO agent. The main figure reports the routability trend; Table 21 retains the per-seed routability breakdown. For those diagnostics, Jumanji/Sable wirelength is converted from grid steps to millimeters using the grid pitch, 100/grid mm per cell.

Table 15: D1 learned-policy hyperparameters. RL settings used for the learned-methods. Each row isolates one training, architecture, or reward setting; "–" denotes a setting that is not used by that method.

Setting	PPO	A2C (Jumanji)	Sable / Mava
Data and budget			
Grids	10, 50, 100, 200, 500	10, 50, 100, 200; 500 OOM	10, 50, 100, 200, 500
Seeds	42–45	42–45	42–45
Train split	10K KiCad boards/grid	10K Connector-v2 NPZ instances/grid	10K Connector-v2 NPZ instances/grid
Eval protocol	128 exact test boards/grid × 5 rollouts; selected rollout/board	128 exact test NPZ instances/grid × 5 rollouts; selected rollout/board	128 exact test NPZ instances/grid × 5 rollouts; selected rollout/board
Episode horizon	256	256	256
Native endpoint	300 PPO iterations	D1-10: 8300 epochs; D1-50: 1600 epochs; D1-100: 5000 epochs; D1-200: 2200 epochs	D1-10: 1.58M updates; D1-50: 1.76M; D1-100: 1.46M; D1-200: 0.8125M; D1-500: 0.18M
Optimization			
Parallelism	32 envs	total batch 256	16 envs
Rollout length	512	10	128
Update epochs	4 PPO epochs	100 learner steps/epoch	4 PPO epochs
Minibatch/update batch	minibatch 256	total batch 256	update batch 2; 2 minibatches
Learning rate	10^{-4} with 20-iter warmup	2×10^{-4} for D1-10/D1-50; 1.25×10^{-5} for D1-100; 6.25×10^{-6} for D1-200	actor lr 2.5×10^{-4}
Discount γ	0.995	1.0	0.99
GAE / trace	GAE $\lambda = 0.95$	bootstrap factor 0.95	GAE $\lambda = 0.95$
Policy clip	0.2	–	0.2
Entropy coefficient	0.01	0.01	0.01
Value coefficient	0.5	TD loss weight 1.0	0.5
Gradient clip	0.5	–	0.5
Architecture			
Observation	KiCad token stream	Connector grid tensor	vector Connector observation
Backbone	token Transformer	Jumanji Connector A2C	feed-forward Sable with retention memory
Width	$d_{\text{model}} = 128$, FFN 512	conv channels 32, MLP 512	embedding 64
Depth / heads	4 layers, 8 heads	4 encoder blocks, 8 heads, key size 16	1 block, 1 head
Action constraint	action masking with same-net bias	grid-action mask	grid-action policy
Reward and accounting			
Reward form	newly connected nets − movement cost	newly connected nets − movement cost	newly connected nets − movement cost
Movement unit	wirelength increment in mm	grid-action step	grid-action step
Movement cost	$0.003 \times \Delta$WL(mm)	D1-10: 0.03; D1-50: 0.006; D1-100: 0.003; D1-200: 0.0015	D1-10: 0.03; D1-50: 0.006; D1-100: 0.003; D1-200: 0.0015; D1-500: 0.0006
Extra step cost	0	–	–
WL accounting unit	native mm	grid steps × $100/G$ mm	grid steps × $100/G$ mm

K.3 D1 training-time validation budgets.

The D1 learning curves are diagnostic W&B validation logs, not the held-out rollout protocol used for the main results. The PPO runs use the recent 300-iteration checkpoint batch and complete in roughly 13–19 hours per seed/grid in W&B runtime. The corrected A2C (Jumanji) and Sable sweeps use fixed native endpoints chosen to match roughly one day of active training for the corresponding grid scale. Table 16 records these endpoints so that the cropped axes in Fig. 22 are auditable.

K.4 Training-time validation curves for D1 grid-size scalability.

Figure 22 visualizes the validation diagnostics logged during training for the learned policies in Fig. 6. These curves are not the final reporting protocol: the main figure evaluates the selected checkpoints on the exact 128 held-out test boards per grid and seed. The plots instead show optimization dynamics under each method's native training axis, with seed mean and standard-deviation bands.

Table 16: D1 W&B validation-log endpoints. Runtime is W&B `_runtime`, summarized as mean [min,max] hours across the listed seeds. The native endpoint is the largest logged training coordinate used by the validation curves. The displayed endpoint is the plotted coordinate after the figure's display-only clipping/scaling; for Sable this is the logged update counter divided by 100.

Method	Subset	Seeds	Runtime (h)	Native endpoint	Displayed endpoint	State / note
PPO	D1-10	42–45	19.5 [17.7, 22.0]	300 iter	300	finished
PPO	D1-50	42–45	14.0 [13.7, 14.3]	300 iter	300	finished
PPO	D1-100	42–45	13.3 [12.3, 15.9]	300 iter	300	finished
PPO	D1-200	42–45	13.0 [11.6, 15.7]	300 iter	300	finished
PPO	D1-500	42–45	13.7 [12.0, 16.7]	300 iter	300	finished
A2C (Jumanji)	D1-10	42–45	approx. 24	8300 epochs	5000 (clipped)	finished
A2C (Jumanji)	D1-50	42–45	approx. 24	1600 epochs	1600	finished
A2C (Jumanji)	D1-100	42–45	approx. 24	5000 epochs	5000	finished
A2C (Jumanji)	D1-200	42–45	approx. 24	2200 epochs	2200	finished
A2C (Jumanji)	D1-500	–	–	–	–	OOM
Sable	D1-10	42–45	approx. 24	1.58M updates	15.8k	finished
Sable	D1-50	42–45	approx. 24	1.76M updates	17.6k	finished
Sable	D1-100	42–45	approx. 24	1.46M updates	14.6k	finished
Sable	D1-200	42–45	approx. 24	0.8125M updates	8.125k	finished
Sable	D1-500	42–45	approx. 24	0.18M updates	1.8k	finished

K.5 Baseline Router Hyperparameters

We compare our learned policy against three classical PCB routers: KiCadRoutingTools (KRT), OrthoRoute (a PathFinder-style negotiation router), and Freerouting 2.1.0. Each router consumes its own preferred input format, so every input board is converted accordingly (Freerouting: DSN, OrthoRoute: ORP, KRT: native KiCad PCB) before the router is invoked, and the hyperparameters used for each board are recorded alongside the routing result. All settings are taken from each router's *default* configuration; no tuning is performed. The *Tunable* column indicates whether a parameter is freely user-configurable (✓) or hard-coded inside the router (✗).

Table 17: KiCadRoutingTools (KRT) hyperparameters. Search-algorithm parameters, cost-function weights, and geometry parameters are all exposed as CLI arguments of `route.py` and are therefore user-configurable. Geometry parameters are set to match the design rules of each input board.

Group	Parameter	Value	Tunable
Search	`grid_step` (mm)	0.1	✓
	`max_iterations`	200,000	✓
	`max_probe_iterations`	5,000	✓
	`max_rip_up_count`	3	✓
	`heuristic_weight`	1.9	✓
	`ordering_strategy`	`mps`	✓
Cost	`via_cost`	50	✓
	`via_proximity_cost`	10	✓
	`turn_cost`	1,000	✓
	`direction_preference_cost`	50	✓
Geometry	`layers`	{F.Cu, B.Cu}	✓
	`track_width` (mm)	matched to input board's design rules	✓
	`clearance` (mm)	matched to input board's design rules	✓
	`via_size` (mm)	matched to input board's design rules	✓
	`via_drill` (mm)	matched to input board's design rules	✓

Grid resolution and design rules in OrthoRoute. PathFinder does not explicitly evaluate trace width or clearance; the grid spacing acts as an implicit minimum trace-to-trace distance. We use the upstream default `grid_pitch` = 0.4 mm, and we do *not* enforce the board's design rule (`track_width` + `clearance`) on `grid_pitch`. PathFinder also internally tunes a subset of its parameters (`pres_fac_*`, `hist_*`, `max_iterations`) on the fly based on per-board flexibility.

A note on iteration semantics. The three routers use mutually incomparable termination conditions: KRT's `max_iterations` counts A* expansions across all rip-up attempts, OrthoRoute's `max_iterations` counts PathFinder negotiation rounds, and Freerouting's `-mp` sets the maximum number of full ripup-and-reroute passes. We therefore do not normalise these budgets and instead use each router's default termination condition.

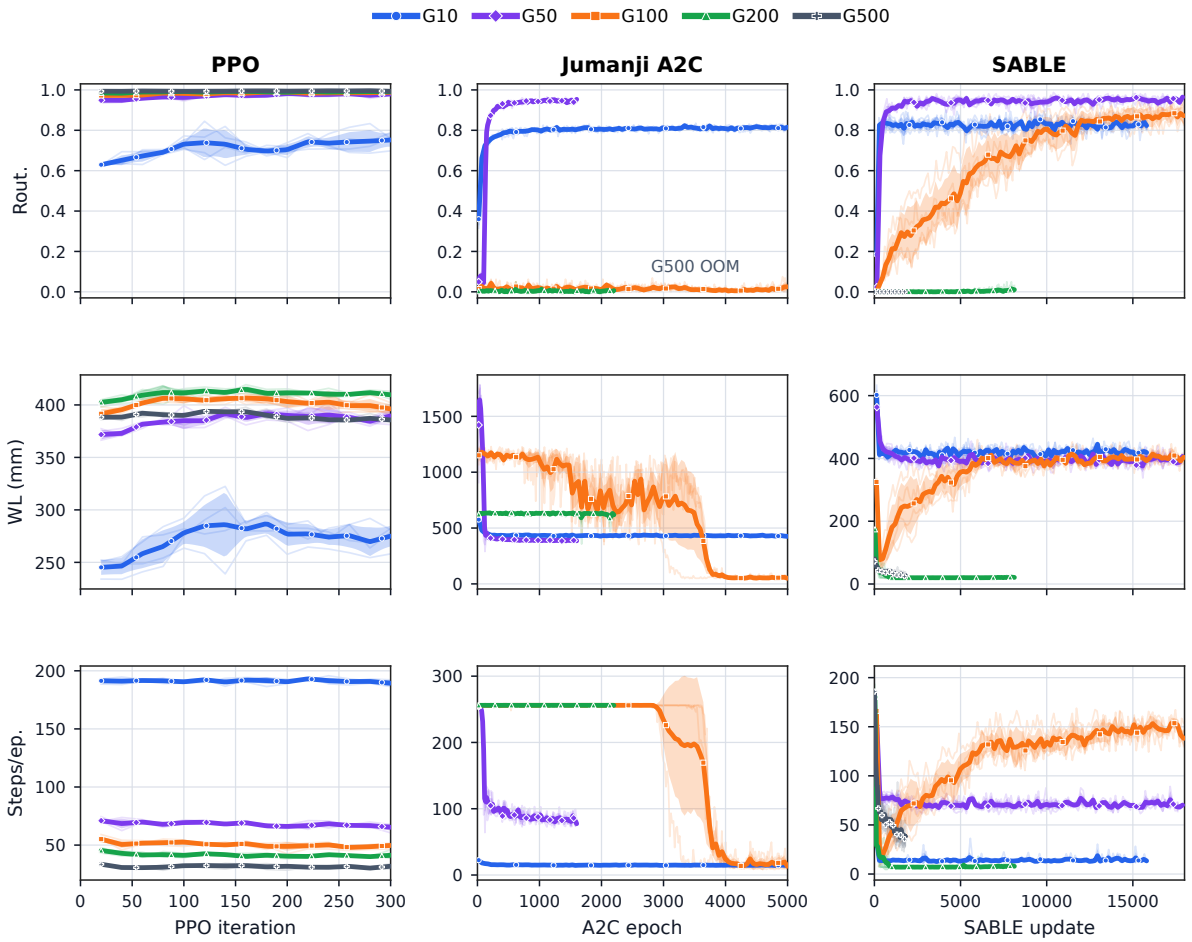


Figure 22: D1 grid-size scalability training-time validation diagnostics. Columns separate PPO, Jumanji A2C, and SABLE runs; rows show routability, wirelength, and episode length. This corrected four-seed plot uses the fixed Jumanji/SABLE split protocol and native endpoints; the A2C axis is displayed up to 5000 epochs and the SABLE update counter is divided by 100 for readability. Thick lines are seed means, translucent bands show one standard deviation, and faint traces show individual seeds. These curves are diagnostic W&B validation logs; final paper metrics are recomputed separately on held-out test rollouts.

L Additional results

Per-seed std for D2/D3-A routing quality. Table 20 reports mean±std for the evaluation that the main Table 3 summarizes with mean only, extended to D3-B and to the per-metric DRV/WL/Via breakdown.

Grid-action baseline (Jumanji). Jumanji [26] instantiates the simplest form of PCB routing under the grid-action abstraction: each net is an agent stepping on a neighbor grid in four directions from a source pad to a target pad, and any visited cell becomes unusable for all agents (single-occupancy). The episode reward is the number of connected nets minus λ times the total wirelength. With every net configured as a 2-pad connection of unit-cell width, we adopt this environment as our grid-action baseline and pit it against the PPO agent on identical instances.

KiCad-API treatment of grid scale. Under the grid-action abstraction, enlarging the grid preserves the local four-neighbor move set but expands both the state space and the routing horizon, so the same physical instance becomes a strictly harder MDP. The KiCad-API abstraction sidesteps this by treating track width and clearance as *environment parameters* rather than as state: the agent issues geometric net-segment commands whose action dimensionality is independent of the grid resolution, and grid scaling only changes the routing margin available to each net. The D1 grid-size scalability experiment isolates exactly this design choice on otherwise identical instances.

Table 18: OrthoRoute hyperparameters. PathFinder-style negotiation router [48]. The first block (*Exposed*) lists parameters that the user can change directly; the second block lists internal parameters that are fixed at the `PathFinderConfig` defaults. We use the upstream defaults for all values.

Group	Parameter	Value	Tunable
Exposed	`max_iterations`	40	✓
	`use_gpu`	`True`	✓
PathFinder defaults	`grid_pitch` (mm)	0.4	✗
	`pres_fac_init`	1.0	✗
	`pres_fac_mult`	1.1	✗
	`pres_fac_max`	64.0	✗
	`hist_gain`	0.2	✗
	`hist_cost_weight`	10.0	✗
	`base_cost_weight`	0.3	✗
	`portal_discount`	0.4	✗
	`span_alpha`	0.15	✗

Table 19: Freerouting 2.1.0 hyperparameters. We list only the CLI options that we explicitly set. Other options exposed by the JAR (e.g. `-mt` for the multi-threading level, `-pp` for the number of postroute passes) are left at their JAR defaults.

Group	Parameter	Value	Tunable
CLI	jar version	`freerouting-2.1.0.jar`	✓
	`-mp` (max passes)	10	✓
	`-Xmx` (JVM heap)	4 GiB	✓

Quantitative breakdown and horizon argument. Our KiCad-API agent's CP rises from 0.63 at grid= 10 to 1.00 for grid= 50–500 (selected-sample routability 0.90 to 1.00; the single-rollout Rout.@1 in Figure 6 is correspondingly lower, e.g. 0.77 at grid= 10), tracking the geometric easing of the instance as each unit-cell-width net occupies a smaller fraction of the board. The grid-action baselines display the opposite profile: A2C (Jumanji) and Sable are competitive at grid= 10–50, but A2C collapses by grid= 100 and runs out of memory at grid= 500, while Sable remains viable at grid= 100 (CP 0.73) but collapses at grid= 200–500. This is not a lack of wall-clock effort: the grid-action baselines consume far more environment steps through JAX/vectorized training. Rather, the experiment isolates the core abstraction gap: finer grids make the physical board easier, but they lengthen the local-move MDP horizon that grid-action agents must explore, whereas the KiCad-API action space keeps the decision horizon tied to routed segments rather than cells.

Per-seed std for D1 grid-size scalability. Companion to Fig. 6 in the main text: per-seed mean±std for the learned D1 methods on the exact 128 held-out test boards per grid and seed. The main figure reports CP; this appendix additionally keeps potential-gain, routability, and wirelength diagnostics as illustrated in Table 21. Each board is evaluated with five rollouts, then one selected rollout (largest potential gain) is retained before aggregation. CP requires full connectivity with zero DRC violations on this selected sample; routability divides routed nets by the five two-pin nets in each Connector-v2 board. WL is absolute routed wirelength in millimeters; Jumanji/Sable wirelength is converted from grid steps using the grid pitch, 100/grid mm per cell. WL values with Rout.< 0.5 are retained for auditability but are not comparable compactness estimates because incomplete episodes measure only produced traces.

Per-seed std for PPO/GRPO reward-training analysis. Companion to main Table 3: quality columns report mean±std for the three configurations across 4 seeds. Routability and quality metrics are recomputed from the saved selected PCBs using the KiCad evaluator; CP requires full connectivity and zero KiCad error-level DRVs. Time reports the recorded episode-latency mean used in main Table 3; Table 23 reports the board-level timing distribution.

Checkpoint provenance for reward analysis. The reward sweep in Fig. 9 uses the existing 9-cell × 4-seed checkpoint set from the benchmark experiments; it is not a new training sweep. The default dense PPO cell with wire penalty 0.002, via penalty 0.1, and seed 42 uses the best checkpoint from a completed same-configuration recovery run because the original symbolic-link target for that seed was unavailable. All reward-analysis metrics are still recomputed with the same held-out rollout and KiCad evaluator protocol used for the other seeds.

Training-time validation curves for PPO/GRPO analysis. Figure 23 reports the W&B diagnostics logged while training the three PPO/GRPO analysis policies. As with the D1 scalability curves, these curves are included to show optimization behavior, not to define the final paper numbers. Main Table 3 and Table 22 use selected held-out rollout artifacts, with quality metrics recomputed from saved PCBs using the

Table 20: Routing-quality std breakdown (D2 / D3-A / D3-B). Companion to the main Table 3. For the stochastic methods (Freerouting, Transformer-based RL) we report per-seed sample standard deviation ($n-1$) across 4 seeds alongside the seed-mean; † marks a single-seed value (std undefined). Metrics are clean pass (CP) and potential gain (Pot.). DRV/WL/Via are computed on the same potential-selected rollout as CP, Pot., and Rout.

Split	Method	CP↑	Pot.↑	Rout.↑	DRV↓	WL↓	Via↓	Time↓
D2	Freerouting	1.00 ± 0.00	9.76 ± 0.07	1.00 ± 0.00	0.00 ± 0.00	392.3 ± 1.2	2.41 ± 0.05	2.69 ± 0.01
	OrthoRoute	0.01†	-4.58†	0.30†	8.55†	323.4†	11.30†	2.54†
	KiCadRoutingTools	1.00†	10.34†	1.00†	0.00†	373.9†	8.09†	0.82†
	GPT-5.4	0.96†	10.28†	1.00†	0.05†	390.3†	2.26†	94.04†
	GPT-5.4-mini	0.58†	6.66†	0.86†	1.40†	370.3†	2.09†	33.93†
	GPT-5.4-nano	0.55†	6.41†	0.85†	1.45†	362.5†	2.09†	63.39†
	Qwen3.5-397B	0.33†	3.85†	0.74†	2.48†	326.4†	1.57†	47.58†
	PPO	1.00 ± 0.00	10.52 ± 0.00	1.00 ± 0.00	0.00 ± 0.00	415.1 ± 3.3	2.18 ± 0.17	0.43 ± 0.04
	PPO (terminal)	0.00 ± 0.00	-5.24 ± 0.51	0.26 ± 0.04	6.40 ± 0.32	1476.3 ± 894.5	58.53 ± 35.90	5.53 ± 0.24
	GRPO	1.00 ± 0.00	9.16 ± 0.69	1.00 ± 0.00	0.00 ± 0.00	619.5 ± 75.2	7.24 ± 3.68	1.16 ± 0.13
D3-A	Reference	1.00†	15.33†	1.00†	0.00†	170.2†	1.18†	–
	Freerouting	0.80 ± 0.01	14.80 ± 0.10	0.91 ± 0.00	0.86 ± 0.05	154.9 ± 0.5	0.88 ± 0.03	7.09 ± 0.26
	OrthoRoute	0.02†	-11.50†	0.51†	99.52†	129.8†	26.97†	2.20†
	KiCadRoutingTools	0.74†	12.37†	0.94†	2.38†	142.8†	3.69†	0.65†
	GPT-5.4	0.65†	11.65†	0.91†	1.29†	151.1†	2.37†	231.15†
	GPT-5.4-mini	0.28†	6.22†	0.72†	3.88†	127.0†	0.41†	56.85†
	GPT-5.4-nano	0.30†	6.17†	0.73†	3.84†	129.1†	0.45†	100.76†
	Qwen3.5-397B	0.34†	6.88†	0.75†	3.62†	120.1†	0.66†	81.82†
	PPO	0.86 ± 0.03	13.59 ± 0.20	0.95 ± 0.01	0.88 ± 0.09	164.1 ± 1.6	1.16 ± 0.41	1.83 ± 0.26
	PPO (terminal)	0.01 ± 0.02	-5.81 ± 0.49	0.23 ± 0.03	11.44 ± 0.82	461.7 ± 236.4	42.55 ± 26.26	5.97 ± 0.76
	GRPO	0.85 ± 0.04	12.09 ± 0.99	0.98 ± 0.01	0.44 ± 0.13	293.1 ± 41.0	10.47 ± 4.62	3.54 ± 0.82
D3-B	Reference	1.00†	47.47†	1.00†	0.00†	570.3†	6.70†	–
	Freerouting	0.78 ± 0.05	44.18 ± 0.56	1.00 ± 0.00	10.60 ± 0.35	532.4 ± 3.5	4.00 ± 0.27	9.94 ± 2.17
	KiCadRoutingTools	0.20†	23.36†	0.86†	44.70†	472.3†	11.80†	3.27†
	GPT-5.4	0.00†	17.47†	0.62†	17.30†	347.1†	3.60†	865.83†
	GPT-5.4-mini	0.00†	13.76†	0.61†	17.60†	561.5†	0.90†	422.06†
	GPT-5.4-nano	0.00†	13.32†	0.59†	18.60†	460.0†	0.30†	510.25†
	Qwen3.5-397B	0.00†	8.82†	0.51†	21.20†	310.0†	10.70†	222.29†
	PPO	0.45 ± 0.10	33.85 ± 4.35	0.85 ± 0.06	8.28 ± 2.95	668.0 ± 63.7	6.22 ± 1.80	10.77 ± 3.51
	PPO (terminal)	0.00 ± 0.00	-14.98 ± 0.80	0.09 ± 0.02	40.72 ± 0.74	589.2 ± 270.1	41.50 ± 25.68	9.02 ± 2.53
	GRPO	0.10 ± 0.00	16.47 ± 6.10	0.69 ± 0.10	15.78 ± 4.71	1158.9 ± 209.7	24.77 ± 9.52	11.20 ± 5.12

KiCad evaluator. We do not plot a training-time CP curve here because the logged validation scalars are not consistently the CP quantity used in the tables; CP requires the joint condition of full connectivity and zero KiCad error-level DRV on the same selected route.

PPO is usually fast on D3-A (median 1.04 s, mean 1.83 s), with the tail driven by a few hard real boards that run to the 256-step horizon and trigger expensive KiCad geometry updates during rollout and serialization. On the larger D3-B boards all three configurations run an order of magnitude slower (means 9–11 s). GRPO and PPO (terminal) have more stable timing profiles because they more consistently consume the full horizon across boards.

Table 21: D1 grid-size sweep across action abstractions. Mean±std across seeds (per-board sample selected by Pot.) on the 128 held-out Connector-v2 boards per grid. Metrics are clean pass (CP), potential gain (Pot.), routability (Rout.), and routed wirelength (WL, in mm; grid-action baselines converted from grid steps via the grid pitch $100/\text{grid}$ mm/cell). A2C (Jumanji) at grid= 500 exceeded memory (OOM). WL at Rout.< 0.5 reflects only partial traces and is not a comparable compactness estimate.

Grid	Method	CP↑	Pot.↑	Rout.↑	WL↓
10	PPO	0.63 ± 0.01	4.75 ± 0.02	0.90 ± 0.00	344.2 ± 5.2
	A2C (Jumanji)	0.51 ± 0.02	4.22 ± 0.12	0.87 ± 0.01	407.1 ± 6.5
	Sable	0.46 ± 0.02	4.04 ± 0.11	0.86 ± 0.01	382.3 ± 2.7
50	PPO	1.00 ± 0.00	6.92 ± 0.03	1.00 ± 0.00	370.1 ± 2.6
	A2C (Jumanji)	0.91 ± 0.00	6.51 ± 0.03	0.98 ± 0.00	462.2 ± 6.3
	Sable	0.88 ± 0.02	6.38 ± 0.11	0.97 ± 0.01	370.9 ± 3.0
100	PPO	1.00 ± 0.00	6.95 ± 0.00	1.00 ± 0.00	363.0 ± 1.2
	A2C (Jumanji)	0.00 ± 0.00	-5.47 ± 0.15	0.03 ± 0.02	76.7 ± 6.6
	Sable	0.73 ± 0.13	5.59 ± 0.71	0.94 ± 0.04	379.1 ± 7.2
200	PPO	1.00 ± 0.00	6.92 ± 0.02	1.00 ± 0.00	368.7 ± 8.5
	A2C (Jumanji)	0.00 ± 0.00	-5.72 ± 0.02	0.01 ± 0.00	156.7 ± 1.8
	Sable	0.00 ± 0.00	-5.69 ± 0.09	0.01 ± 0.01	25.7 ± 1.6
500	PPO	1.00 ± 0.00	6.95 ± 0.00	1.00 ± 0.00	358.1 ± 1.5
	A2C (Jumanji)	OOM			
	Sable	0.00 ± 0.00	-5.79 ± 0.00	0.00 ± 0.00	28.4 ± 6.9

Table 22: PPO vs. GRPO std breakdown. Same quality setup as main Table 3; quality entries are mean±std across the four seed-level means. D3-A statistics use the 99 compatible boards per seed. Time(s) is the episode-latency mean over the full evaluation inputs.

Dataset	Configuration	Routability↑	CP↑	DRV↓	WL↓	Via↓	Time(s)↓
D2	PPO	1.00 ± 0.00	1.00 ± 0.00	0.00 ± 0.00	415.1 ± 3.3	2.18 ± 0.17	0.43
	GRPO	1.00 ± 0.00	1.00 ± 0.00	0.00 ± 0.00	619.5 ± 75.2	7.24 ± 3.68	1.16
	PPO (terminal)	0.26 ± 0.04	0.00 ± 0.00	6.40 ± 0.32	1476.3 ± 894.5	58.53 ± 35.90	5.53
D3-A	PPO	0.95 ± 0.01	0.86 ± 0.03	0.88 ± 0.09	164.1 ± 1.6	1.16 ± 0.41	1.83
	GRPO	0.98 ± 0.01	0.85 ± 0.04	0.44 ± 0.13	293.1 ± 41.0	10.47 ± 4.62	3.54
	PPO (terminal)	0.23 ± 0.03	0.01 ± 0.02	11.44 ± 0.82	461.7 ± 236.4	42.55 ± 26.26	5.97
D3-B	PPO	0.85 ± 0.06	0.45 ± 0.10	8.28 ± 2.95	668.0 ± 63.7	6.22 ± 1.80	10.77
	GRPO	0.69 ± 0.10	0.10 ± 0.00	15.78 ± 4.71	1158.9 ± 209.7	24.77 ± 9.52	11.20
	PPO (terminal)	0.09 ± 0.02	0.00 ± 0.00	40.72 ± 0.74	589.2 ± 270.1	41.50 ± 25.68	9.02

L.1 Throughput and Token Usage in LLM experiments

For the OpenAI LLM runs, we record wall-clock time and token consumption per episode. Tables 24, 25, and 26 report the per-episode means: latency in seconds, and three token streams broken out separately – *System* (system-prompt tokens accumulated across all turns), *User* (user-message tokens, dominated by the per-step state-of-board observation), and *Response* (model-generated tokens, including any hidden reasoning the API counts on the input/output side). *Total* is the sum of the three.

M LLM prompts

M.1 PCBWorld agent

The agent receives a single user message per step assembled from the template described in Figure 24. Placeholders with curly-braces are populated at runtime from the environment state.

Placeholders. The runtime values used throughout the experiments in this paper are:

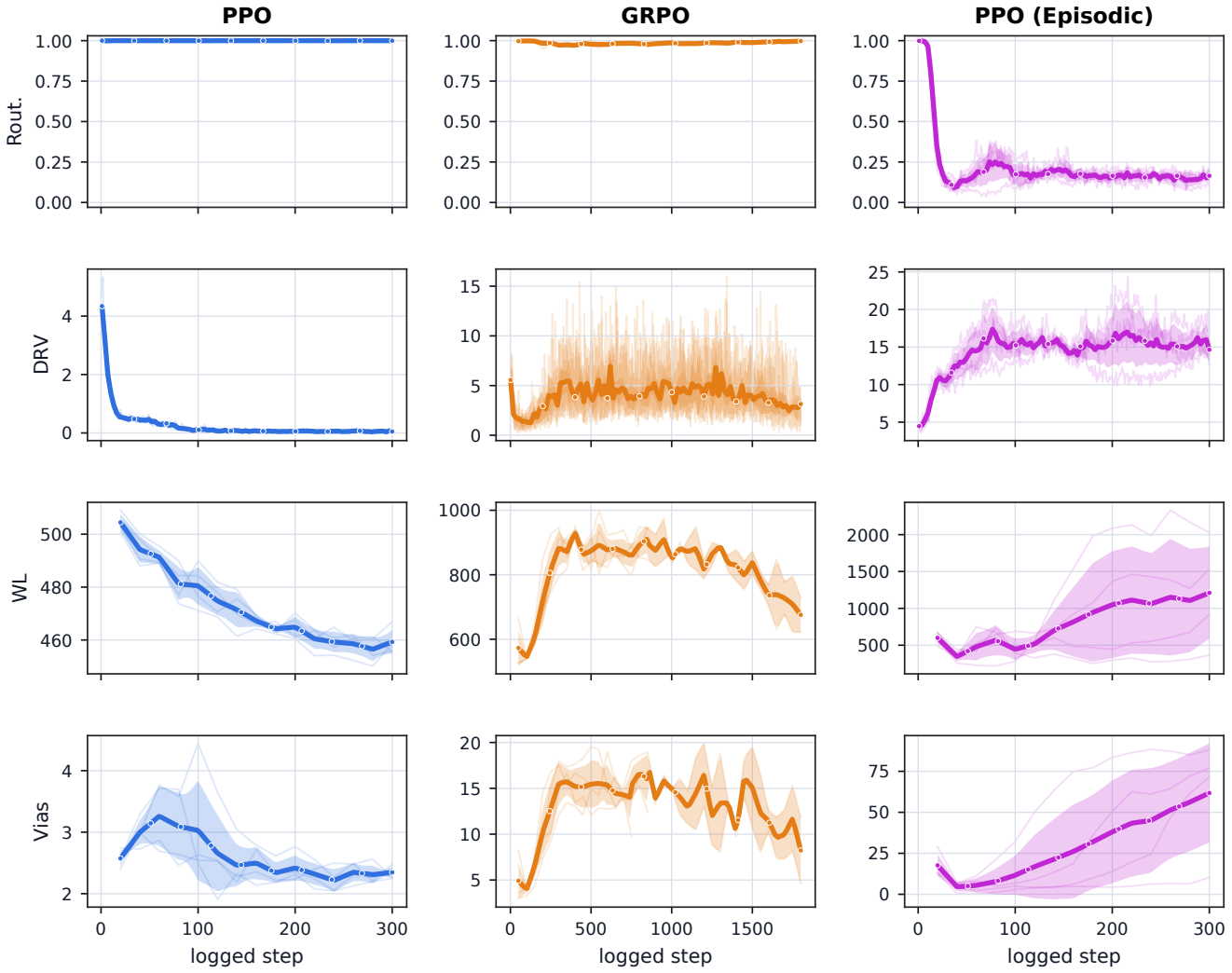


Figure 23: PPO/GRPO training-time validation diagnostics. Columns correspond to PPO, GRPO, and PPO (terminal); rows show routability, DRV, wirelength, and via diagnostics. Curves are W&B logged values over the logged training step or evaluation iteration; thick lines are seed means, translucent bands show one standard deviation, and faint traces show individual seeds. Final Table 3 CP and quality values are computed separately from selected held-out routed PCBs.

- `{StateFormatDesc}` – schema description of the board-state encoding. We use the S-expression encoding (`state_format=sexpr`); the alternative XML encoding is supported but not used in the reported runs.
- `{CurrentStep}` – 1-indexed step counter, ranging from 1 to the per-episode budget T. We set $T = 200$ for D2, D3-A, and D3-B.
- `{CurrentObservation}` – concatenation of the `routing_geometry` and `router_head` blocks, regenerated every step; all floating-point coordinates are rounded to 3 decimal places (mm).
- `{ValidStepCount}` – number of accepted (parsed *and* mask-valid) actions issued so far in the episode; bounded above by `{CurrentStep}`.
- `{ActionHistory}` – rolling window of the most recent `history_length`= 2 turns, each rendered as a `<think>`/`<action>` pair; entries that left the unrouted-pin count unchanged are prefixed with `[no effect]`.
- `{RejectedAttempts}` – conditional slot, empty string in the nominal case. When the previous action was rejected (parse failure or mask veto) it expands to a single line of the form `Step F-L: <body> (rejected ×N)`; consecutive identical bodies are collapsed and the streak is cleared by the next valid action.
- `{ValidActions}` – the action verbs allowed by the current router phase mask (NET_SELECT → START_ROUTE → ROUTING); the underlying action space has 7 indices (six routing verbs plus an IDLE fallback at index 6).

Table 23: Episode timing distribution for PPO/GRPO analysis. Board-level latency distribution for the Time(s) entries in main Table 3. Timing uses the recorded per-board rollout time and includes board reload/reset, policy rollout, and routed-PCB serialization. D3-A excludes the one incompatible 73-net board.

Dataset	Configuration	Boards	Mean	Median	P95	Max
D2	PPO	128	0.43	0.37	0.92	1.42
	GRPO	128	1.16	0.94	2.94	3.67
	PPO (terminal)	128	5.53	5.52	5.77	5.95
D3-A	PPO	99	1.83	1.04	5.39	15.00
	GRPO	99	3.54	3.09	8.55	12.98
	PPO (terminal)	99	5.97	5.80	6.99	12.90
D3-B	PPO	10	10.77	9.31	18.23	19.98
	GRPO	10	11.20	10.44	14.76	15.15
	PPO (terminal)	10	9.02	8.93	10.42	11.23

Table 24: Per-episode latency and token usage on D2 (128 boards × 5 rollouts = 640 episodes). Tokens are means over completed episodes.

Model	Sec/ep	System	User	Response	Total
gpt-5.4	94.0	62,899	39,909	3,823	106,631
gpt-5.4-nano	63.4	66,140	38,124	4,260	108,524
gpt-5.4-mini	33.9	51,538	26,438	2,382	80,358

Table 25: Per-episode latency and token usage on D3-A (99 boards × 5 rollouts = 495 episodes).

Model	Sec/ep	System	User	Response	Total
gpt-5.4	231.1	263,612	152,583	8,683	424,879
gpt-5.4-nano	100.8	199,424	92,200	6,667	298,291
gpt-5.4-mini	56.9	156,258	57,796	3,525	217,578

Table 26: Per-episode latency and token usage on D3-B (10 boards × 5 rollouts = 50 episodes).

Model	Sec/ep	System	User	Response	Total
gpt-5.4	865.83	1,119,161	775,289	26,717	1,921,167
gpt-5.4-nano	510.25	1,117,479	861,073	29,945	2,008,496
gpt-5.4-mini	422.06	1,147,573	918,588	20,134	2,086,296

```
You are a PCB routing agent for KiCad. Connect unconnected pins (points in routing_geometry) within each net. Never connect pins from
      different nets.

# Priority
1. Complete all connections (reduce unconnected points to zero).
2. Avoid DRC violations (no crossing other nets, respect clearance).
3. Minimize total wirelength.

# Routing Guidelines
- Targets are the (point ...) entries in routing_geometry; the final segment must land on the target pad's (x, y) while the cursor is on
      that pad's layer (points carry no layer -- look up the pad in board_static or the nearby tracks).
- Use exact coordinates from the observation (pad positions, point targets, track endpoints). Only invent coordinates for detour waypoints.
- Avoid static obstacles, board edges, and existing tracks on the same layer (any net, including the current one).
- When a path is blocked, detour: switch layers with `make_via` (offset >=1mm from any pad -- never place a via on a pad), route around on
      the opposite layer, then switch back.
```

```
- When the cursor is already near the remaining target with a clear path, prefer `finish <mode>` over a manual `make_line`.

## Layer-change detour pattern
Route from (0, 0) on layer 1 to (10, 10) on layer 1, bypassing an obstacle via layer 2:
  1. start_route 0.0 0.0 1
  2. make_via   0.0 0.5 w        # offset off the source pad, then cross to layer 2
  3. start_route 0.0 0.5 2       # re-enter routing on the new layer
  4. make_via   10.0 9.5 w       # STOP before the target's (x, y) and cross back
  5. start_route 10.0 9.5 1      # re-enter routing on the target pad's layer
  6. make_line  10.0 10.0 w      # final segment lands on the pad, on its own layer

## Do NOT draw or end on a pad from the wrong layer
Never let `make_line` or `make_via` reach the target pad's exact (x, y) while the cursor is on a different layer than that pad. If you do,
      you are trapped:
  - placing a via at the pad violates the "no via on pad" rule, and
  - the pad is unreachable from the wrong layer, so `finish` will fail repeatedly.
Always make the layer-return `make_via` at an offset waypoint (>=1mm away from the target pad) *before* the track arrives at the target's
      (x, y).

# Board State
{StateFormatDesc}

## Step {CurrentStep}
{CurrentObservation}

# History (step {CurrentStep}, {ValidStepCount} valid actions taken)
{ActionHistory}{RejectedAttempts}

# Valid Actions
Choose exactly one from:
{ValidActions}

# Response Format
Output exactly one <think>...</think> followed by one <action>...</action> with a valid action. Never repeat think-action pairs.
"[no effect]" in history = action made no progress; "Last rejected attempt" below = action was refused. Do not repeat either.
```

Figure 24: Full prompt template for the PCBWorld agent.

M.2 Baselines

Engine-free (open-loop) generation. An LLM directly generates a complete KiCad board file (`.kicad_pcb`) from the initial board state.

```
You are an expert PCB routing engineer using KiCad PCB format.

Your task is to generate valid PCB routing (tracks and vias) for a given KiCad PCB board.

## Instructions
- Analyze the given PCB layout, including components, pads, and nets.
- Generate routing (segments and vias) that correctly connects pads belonging to the same net.
- Ensure:
  - No short circuits between different nets
  - Minimal via usage unless necessary
  - Reasonable routing paths (avoid unnecessary detours)
  - Respect layer usage (F.Cu, B.Cu)

## Routing geometry: OCTILINEAR ROUTING with the 45-degree-ONLY constraint (mandatory)
Use **octilinear routing** for every `(segment ...)`. Octilinear routing is a wire/edge routing style where connections are restricted to
      **eight directions**: the four cardinal (horizontal, vertical) plus the four diagonals at 45 degrees. It sits between rectilinear
      routing (4 directions, Manhattan-style) and fully arbitrary Euclidean routing.

### The 45-degree-only constraint (sub-rule of octilinear)
Whenever a segment is *not* horizontal or vertical, it MUST be a **strict 45 (or 135) degree diagonal** -- i.e. exactly |dx| == |dy|. No
      other diagonal angle is permitted. The following are all VIOLATIONS:
    30, 60        (e.g. (0,0) -> (10, 5.77) or (0,0) -> (5, 8.66))
    22.5, 67.5    ("half-octilinear" angles)
    13.16, 26.6   (or any other arbitrary slope)
```

```
Even tiny rounding deviations break the rule: `(start 0 0) (end 10 9.9)` is NOT octilinear. If you intend a 45-degree diagonal, the rise
      must equal the run exactly.

### Allowed segment shapes (and only these)
Concretely, a segment from `(start sx sy)` to `(end ex ey)` is octilinear iff one of the following holds, with `dx = ex - sx` and `dy = ey
      - sy`:
    1. dy == 0                          (East / West   -- 0,   horizontal)
    2. dx == 0                          (North / South -- 90,  vertical)
    3. dx ==  dy  (and both nonzero)    (NE  / SW      -- 45,  diagonal)
    4. dx == -dy  (and both nonzero)    (NW  / SE      -- 135, diagonal)
Equivalently, every segment is horizontal, vertical, or a 45-degree diagonal where |dx| == |dy|. Anything else is forbidden.

### How to handle non-octilinear paths
If the natural route between two pads is at an angle that is not in {0, 45, 90, 135} degrees, decompose it into multiple octilinear
      segments joined at corners. For example, to go from (0,0) to (10,3) you might use:
    (segment ... (start 0 0) (end 3 3) ...)   ; 45-degree diagonal
    (segment ... (start 3 3) (end 10 3) ...)  ; horizontal
Use as many segments as needed; each one must individually satisfy the rule above.

This is exactly the constraint KiCad's interactive PNS router enforces with `corner_mode = MITERED_45`.

## Output Format
- Return ONLY the routing additions in valid KiCad PCB format.
- Do NOT repeat the full board.
- Only include `(segment ...)` and `(via ...)` entries.
```

Figure 25: Full prompt template for the engine-free (open-loop) generation.

Plan-only (open-loop) generation. An LLM produces subsequent routing API calls at once, without interacting with the environment during generation.

```
You are an expert PCB routing engineer driving a KiCad routing API.

You will be given the *initial* state of a PCB board (footprints, pads, nets) and must output the **complete sequence of routing API
      calls** that would route every net.

## API
The router exposes 6 high-level actions. Emit one per line.
    net_select <net_id>                     Select a net to start routing.
    start_route <x_mm> <y_mm> <layer>       Begin routing at a pad position.
                                            layer: 1 = F.Cu (top), 2 = B.Cu (bottom).
    make_line <x_mm> <y_mm> <mode>          Extend a track to (x,y) on the current layer.
    make_via <x_mm> <y_mm> <mode>           Extend track to (x,y), drop a via, switch layer.
                                            AFTER make_via, the current layer flips.
    finish <mode>                           Auto-complete the active route on the *CURRENT*
                                            layer to the nearest pending pad of the
                                            current net. CANNOT cross layers — if the
                                            remaining pad is on a different layer, finish
                                            will NOT reach it.
    net_end                                 Mark the active net as fully routed.
    mode is one of: m (MarkObstacles), p (PushAndShove), w (Walkaround). Use w by default.

## Reading <BOARD>
The input is KiCad-style sexpr. Inside `(nets ...)` each net lists its pads:
    (pad <id> <x> <y> <layer_tag>)
Layer tags:
    `1`  -> the pad lives on layer 1 (F.Cu, top).
    `2`  -> the pad lives on layer 2 (B.Cu, bottom).
    `th` -> through-hole, electrically present on BOTH layers; you
           may treat it as either 1 or 2 when choosing start_route's
           layer parameter or as the via-side of a make_via.

## Coordinates and layers (MUST follow exactly)
- Copy each pad's `<x>` and `<y>` token **verbatim** — same digits, same decimal places, no rounding. The router only accepts a pad's
      *exact* (x_mm, y_mm). Example: if the board has
    (pad D0 67.500 44.700 th)
  then output
```

```
    start_route 67.500 44.700 1
  not `start_route 67.5 44.7 1`.
- start_route MUST be issued at a pad position of the currently selected net.

## Layer-choice rule for the FIRST pad
You pick a `<layer>` argument for `start_route`. The choice determines which layer the router head sits on, which constrains every
      subsequent action.

Pick the start layer using THIS priority:
  1. If the first pad has tag `1` -> use 1.
  2. If the first pad has tag `2` -> use 2.
  3. If the first pad has tag `th` and at least one other pad of
     this net has tag `1` or `2` -> use that other pad's layer.
     (This avoids an unnecessary via.)
  4. If the first pad has tag `th` and all other pads also `th`
     -> use 1.

## Routing protocol (state machine — violations cause the entire net to fail)
After `net_select`, the router is in {has_net=True, is_routing=False}.
After a SUCCESSFUL `start_route`, it moves to {has_net=True, is_routing=True}.
`make_line`, `make_via`, `finish` are ONLY valid while is_routing=True.
`net_end` closes the net and returns to {has_net=False}.
Emitting any of make_line / make_via / finish / net_end before a
successful start_route causes the entire net to fail.

## Per-topology emission rules (CASE BY CASE — match the net's shape)
Resolve each pad's *effective* layer using the layer-choice rule (`th` resolves to either 1 or 2 by neighbour). Then dispatch:

### Case A — 0 or 1 pads : skip the net entirely.

### Case B — 2 pads on the SAME effective layer L
  net_select <id>
  start_route <P1.x> <P1.y> L
  make_line  <P2.x> <P2.y> <mode>   ; recommended over finish
  net_end

  (alternative: `finish <mode>` also works — auto-completes head to P2 on L)

### Case C — 2 pads on DIFFERENT layers (cross-layer)
  `finish` cannot cross layers. Pick an intermediate point P1' and go
  P1 -> P1' -> P2, using make_via at P1' to switch layers:
  net_select <id>
  start_route <P1.x> <P1.y> L1        ; L1 = P1's layer
  make_via   <P1'.x> <P1'.y> <mode>   ; switch L1 -> L2 at P1'
  start_route <P1'.x> <P1'.y> L2      ; L2 = P2's layer, restart at P1'
  make_line  <P2.x> <P2.y> <mode>     ; (or `finish <mode>`)
  net_end

### Case D — k pads (k >= 3) all on the SAME effective layer L
  net_select <id>
  start_route <P1.x> <P1.y> L
  make_line  <P2.x> <P2.y> <mode>
  start_route <P2.x> <P2.y> L
  make_line  <P3.x> <P3.y> <mode>
  ... up to P(k-1) ...
  make_line  <P(k).x> <P(k).y> <mode>   ; (or `finish <mode>`)
  net_end

## Anti-patterns (do NOT do these)
  ✗ `finish` immediately after start_route when the second pad is
     on a different layer. finish will route within the current
     layer and silently miss the cross-layer pad.
     ✓ Use make_via to land on the cross-layer pad (Case C).
  ✗ Picking a `<layer>` for start_route that no pad of this net
     lives on (e.g. layer 2 when both pads are on 1).
     ✓ Apply the layer-choice rule.
  ✗ Reformatting coordinates (5.7 instead of 5.700, dropping
     trailing zeros). The board's pad table has the exact form;
```

```
    copy it.
  ✗ Emitting `make_line <Pk.x> <Pk.y>` THEN `finish` for the
    final pad of a same-layer chain. `finish` already handles Pk
    — the explicit make_line creates a duplicate / overlapping
    segment and may break DRC.
  ✗ Forgetting `net_end` between nets. The next `net_select`
    fails if the previous net wasn't closed.
  ✗ Calling start_route at coordinates that are not a pad of the
    currently selected net (e.g. picking a pad of a different
    net by accident). The route fails silently.

## Output Format
Wrap the entire sequence in <actions>...</actions>. One action per line. No commentary inside the block. Do NOT repeat the board.
```

Figure 26: Full prompt template for the plan-only (open-loop) generation.

N Visualization gallery and LLM failure traces

Importance of net selecting order. Across the studied rollouts, `net_select`—nominally a simple routing-target selection action—emerges as a critical strategic decision whose first invocation often determines whether a board can be successfully routed (see Figure 27). In the 0018 `hy_adapter` case, successful episodes consistently begin with selecting a routing order that minimizes early routing difficulty, whereas unsuccessful episodes choose a more challenging net at the outset and subsequently fail to recover within the limited step budget. Since the agent rarely performs corrective recovery behaviors after an unfavorable decision, the initial `net_select` effectively constrains the future routing space and strongly influences the overall routing outcome. These observations suggest that routing-target ordering is not merely a procedural choice, but a primary factor governing downstream solve success.

Importance of Route Editing. In the action space of our Gym environment, the agent is allowed to push existing wires, but direct wire editing operations were intentionally excluded. This design choice was made to reduce the action space and focus the problem scope on routing itself. However, as a consequence, the agent tends to struggle when severe congestion occurs, since it has limited capability to resolve routing deadlocks after they emerge. Nevertheless, we observe that successful routing remains achievable when the agent proactively avoids congestion through early-stage planning and anticipatory reasoning.
Figure 28 analyzes successful and failed cases on (`0100_smt-zvs-driver_IH10-mc`). Since the action space does not include direct editing of pre-existing tracks, once congestion is detected, the agent tends to repeatedly attempt rerouting through alternative paths. In contrast, when the agent proactively anticipates potential congestion and initiates routing from a different region in advance, it can achieve more effective routing with fewer steps.

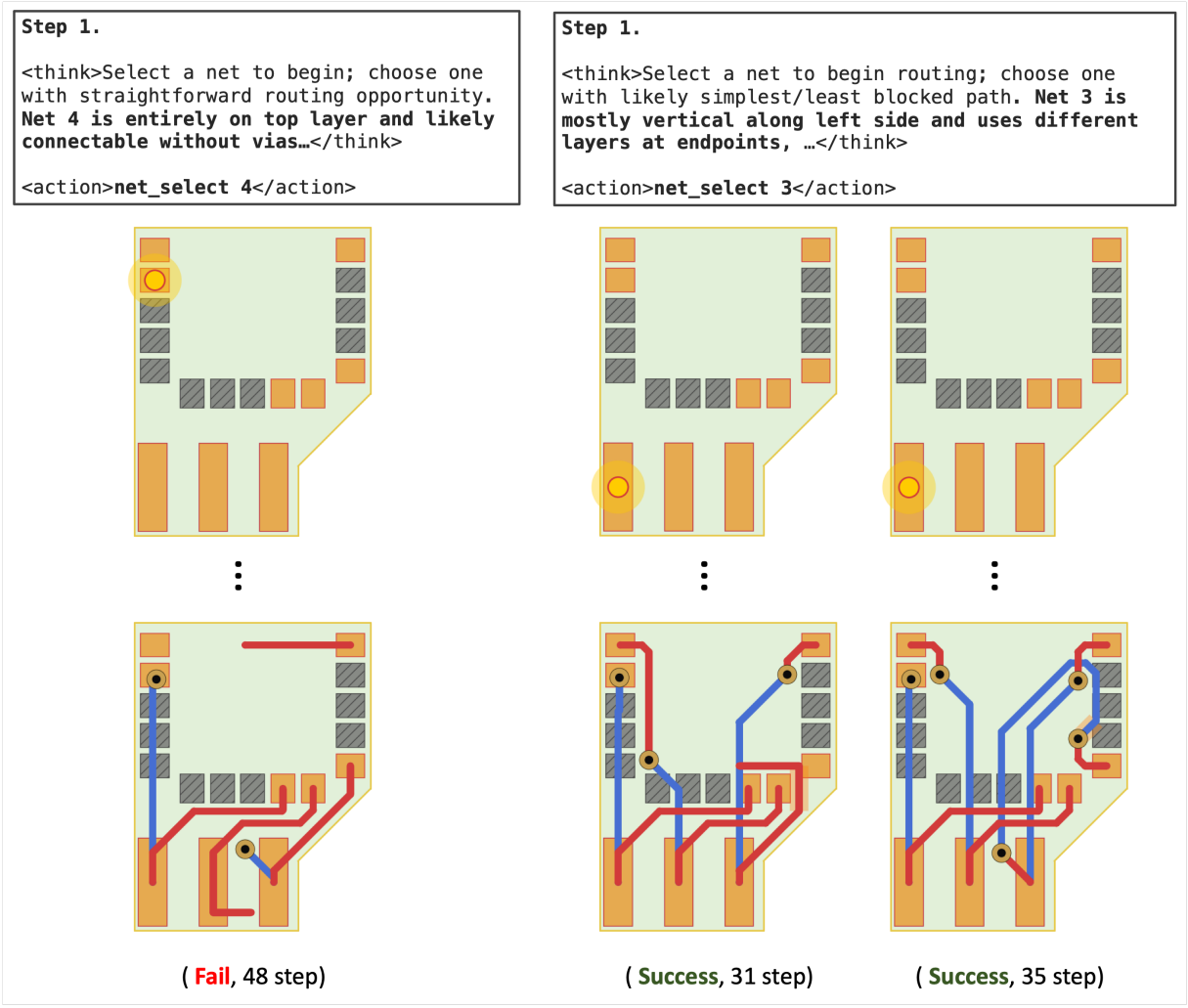


Figure 27: Case Study 1 (`0018_hy_adapter`). In our current action design, determining the routing priority of nets is critically important. Although different nets were selected for different reasons, we observe that routing succeeds when a particular net (net 3) is selected first, whereas it fails when another net (net 4) is selected first.

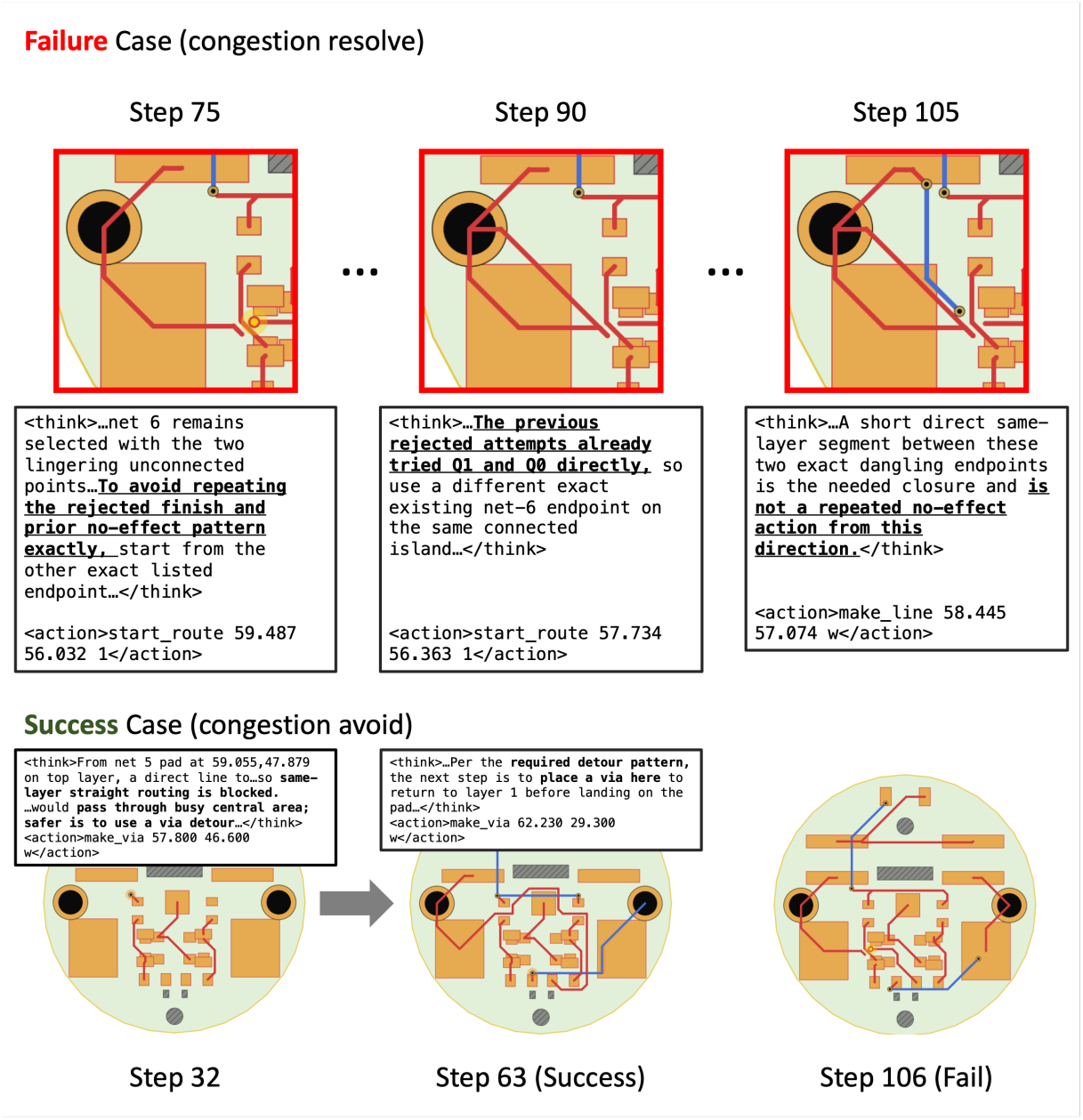


Figure 28: Case Study 2 (`0100_smt-zvs-driver_IH10-mc`). Failure case (top) and success case (bottom). Since the action space does not include direct editing of pre-existing tracks, once congestion is encountered, the agent repeatedly attempts rerouting through alternative paths. In contrast, proactively anticipating congestion and initiating routing from a different region enables more effective routing with fewer steps.

O Datasheet for Datasets

This appendix summarizes PCBWorld-Bench using the main categories of the Datasheets for Datasets template [16]. The benchmark is a collection of KiCad-native PCB routing instances used with the engine-grounded evaluation protocol described in Section 4.

O.1 Motivation

PCBWorld-Bench was created to evaluate PCB routing agents under the same engine-checked conditions: DRC status, connectivity, wirelength, and via count are all computed by KiCad. The dataset targets comparison among LLM tool-call agents, RL policies, grid-action baselines, and rule-based routers, rather than supervised imitation of a single reference layout. It contains no human-subject records or personal data.

O.2 Composition

Each instance is stored as a `.kicad_pcb` file containing the board geometry, placed pads/components, nets, design rules, and the initial routing state. The benchmark has three task families: D1 is a 1-layer grid-routing family converted from Jumanji Connector-style instances [5, 26]; D2 is the Synthetic Gridless Boards family, a 2-layer multi-pin family with configurable net and pad counts; and D3 is a 679-board curated set derived from PCBench's open-source PCB corpus [22], normalized to the KiCad 9 format, populated with design rules taken from each board's own wire values, and retained only when DRV-free. D1/D2 use 10,000/128/128 train/validation/test splits, while D3 is held out for zero-shot evaluation and stratified by pad count into **D3-A** (the 100 boards with the fewest pads, all with ≤ 31 pads), **D3-B** (boards with up to 100 pads not assigned to D3-A), and **D3-C** (> 100 pads). Table 2 summarizes the per-family statistics.

O.3 Collection Process

D1 is produced by converting grid instances into executable KiCad boards. D2 is generated from a parametric synthetic board generator. D3 starts from PCBench, a public PCB routing dataset with per-board `raw.kicad_pcb` and `processed.kicad_pcb` files, PCB-RDL `final.json` routing descriptions, metadata, data augmentation scripts, and an RL environment [22]. We use the KiCad-9-converted `PCBs_version_9` pool as the raw D3 source and apply the compatibility filtering below. No crowdworkers or paid participants were involved.

O.4 Preprocessing

The D3 preprocessing step addresses KiCad-version DRC compatibility rather than changing PCB geometry. The `PCBs_version_9` source pool contains 1,182 boards converted to KiCad 9; many were originally authored in KiCad 4–7, so conversion assigns newer KiCad 9 default DRC constraints that can flag violations even when the physical board file is unchanged. We therefore apply a project-file compatibility pass to `.kicad_pro` files only: non-routing manufacturing/courtyard checks such as `solder_mask_bridge`, `drill_out_of_range`, `malformed_courtyard`, and `courtyards_overlap` are ignored, and KiCad-9 floor constraints for annular width, copper-edge clearance, and hole clearance are relaxed to match earlier-version board compatibility. Core routing constraints used for evaluation, including clearance, shorts/crossings, track width, via diameter, and the source-board netclass settings, are preserved.
After this compatibility pass, 679 boards remain in the curated D3 set. The pass does not edit the physical `.kicad_pcb` content: a byte-level comparison confirmed that all 679 retained board files are unchanged before and after preprocessing. The changes are limited to DRC rule interpretation in the project file, and the retained boards are revalidated through the same KiCad engine interface used for evaluation.

O.5 Uses

The intended use is benchmarking PCB routing policies under a shared engine-grounded protocol, including studies of closed-loop tool use, action masking, synthetic-to-real transfer, and constrained sequential decision-making. The dataset should not be used as fabrication signoff: it does not replace professional checks for signal integrity, thermal behavior, EMI, or product-specific manufacturing requirements.

O.6 Distribution

The synthetic datasets are released under the CC-BY 4.0 license together with their generator and evaluation code. Curated open-source D3 boards retain the licenses of their source repositories and are distributed with attribution metadata where redistribution is permitted. Each instance is versioned together with the corresponding PCBWorld environment release.

O.7 Maintenance

Dataset and environment versions are maintained together so that results can be reproduced against the exact board files and evaluator used. Maintainers will preserve attribution metadata, keep prior versions available when practical, and accept issue reports for corrupted files, license metadata problems, incompatible boards, and evaluator discrepancies.

P License, hosting, maintenance

Splits, Licensing, and Versioning. The synthetic datasets are released under the CC-BY 4.0 license together with their generator code. Open-source boards retain the license of their source repository and ship with an attribution table for redistribution. Every instance carries a dataset version tag in 1:1 correspondence with the environment version. The datasheet is provided in Appendix O.