FARE: FINGERPRINT-AWARE REPRESENTATIVE EVALUATION FOR LLM-BASED HEURISTIC DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based heuristic discovery iteratively proposes, scores, and selects programs using feedback from an evaluation harness. The harness does not merely measure quality; it defines the objective that discovery optimizes. Evaluating every deployment scenario is infeasible because operating conditions vary continuously and interact combinatorially, so discovery must optimize against a finite scenario bank. Over successive iterations, the discovery loop can exploit gaps in this bank, so its high-scoring programs may fail under unseen conditions. Enlarging the bank raises the cost of evaluating every candidate program and may waste budget on behaviorally redundant scenarios. We introduce FARE (Fingerprint-Aware Representative Evaluation). FARE runs diverse probe programs offline on a scenario pool and fingerprints each scenario by its normalized probe scores. It then selects a compact training bank of K scenarios by k-medoids under program disagreement distance, so that every pool scenario has a nearby selected scenario as its representative. FARE reserves the unselected scenarios as a held-out test bank that discovery never sees, so the train–held-out gap measures generalization beyond the training bank. FARE also reports the representation error: the mean distance from each pool scenario to its representative. When a program's score is a smooth function of the fingerprint, the representation error bounds the average error of predicting a pool scenario's score from its representative's score. We evaluate FARE against four alternative bank-construction methods on two case studies, scheduling and congestion control, using 5G-Eval, our automated full-stack 5G platform built on real network software. Across three LLMs, FARE's top-ranked candidate program has the highest pool score in almost all model–pool combinations compared with the four alternatives.

1 INTRODUCTION

> *When a measure becomes a target, it ceases to be a good measure.*
> — Strathern (1997), after Goodhart (1984)

Large language model (LLM)-based heuristic discovery evolves a population of programs by mutating them, evaluating them, and retaining the high-scoring ones as parents of the next generation (Romera-Paredes et al., 2024; Novikov et al., 2025). An LLM serves as the mutation operator, and an evaluation harness scores every candidate program the LLM proposes. To produce that score, the harness runs the program on a bank of scenarios,[1] applies a scoring rule to each run's telemetry, and aggregates the results (Gao et al., 2021; Jimenez et al., 2024). The loop uses this score as its sole optimization signal to guide subsequent proposals, retain parent programs, and select the final program. Thus the evaluation harness does not merely measure the discovery loop's objective; it *defines* that objective and steers the discovery process.

Existing studies often rely on a small, hand-picked set of scenarios. For example, Glia searches against a single configured LLM-inference workload (Hamadanian et al., 2025), and Barbarians' transaction scheduler uses only five database traces (Cheng et al., 2025). However, evaluating on a

[1] A *scenario* is a trajectory of deployment conditions.

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

finite set of scenarios can create substantial generalization risk (Sim et al., 2025) (see Figure 1 for an example). Practical systems and networking tasks such as scheduling, congestion control, caching, and compiler optimization (Mao et al., 2019; Jay et al., 2019; Liu et al., 2020) involve continuously varying inputs (e.g., client arrivals and workload sizes) and operating configurations (e.g., resource budgets, buffer sizes, and latency targets), making exhaustive evaluation over the scenario space infeasible. This motivates a core question in LLM-based heuristic discovery: how to construct a finite representative scenario bank for harness evaluation?

To address this question, we propose Fingerprint-Aware Representative Evaluation (FARE). Given a scenario pool, FARE selects a compact training bank whose scenarios closely represent the pool. It first runs a diverse set of probe programs offline on every pool scenario; the normalized vector of a scenario's probe scores is its *fingerprint*. FARE then groups the pool scenarios into K clusters in fingerprint space and keeps each cluster's *medoid*, the member whose distances to the other members sum to the least, as that cluster's representative. FARE also reports the representation error δ^*, the mean distance from a pool scenario to its representative. This as an *a priori* measure of how faithfully the training bank represents the pool. FARE reserves the scenarios outside the bank as a held-out test bank to directly measure generalization beyond the discovery scenarios. Because the harness does not score candidate programs on these scenarios during discovery, the search cannot optimize for them directly. Any gap between a discovered program's training-bank score and its held-out score provides a complementary *empirical* measure of overfitting to the training bank.

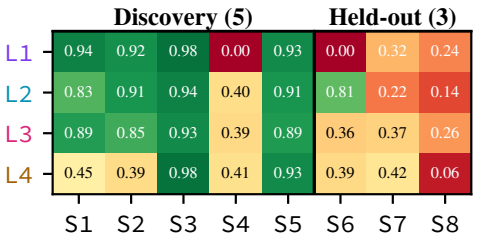

Figure 1: Programs that score highly during discovery may score poorly on held-out scenarios. The figure shows the top-ranked scheduler program from the discovery loop for each of four LLMs (L1–L4: gemini-3.1-pro, gemma4-31b, gpt-5.4, and qwen3.5-397b-a17b). Cell color shows the per-scenario score on S_1^{sch}–S_5^{sch} (used during discovery) and S_6^{sch}–S_8^{sch} (never used during discovery). Appendix C.2 gives experiment details.

We evaluate FARE in 5G radio control, where benchmark overfitting has measurable operational costs. A deployed heuristic must handle the number of active user devices; their traffic classes, mobility, and channel quality; and the available resources and latency targets. These factors interact combinatorially so that no feasible benchmark can test every combination. To study this setting, we built 5G-EVAL (Appendix C), a fully automated and reproducible platform in which a single declarative scenario file drives an unattended, full-stack cellular experiment on real network software. We evaluate two distinct harness instances with 5G-EVAL: one for the scheduler case study and one for the congestion-control case study. They share the platform and FARE's bank-construction method but use task-specific control interfaces, scenario banks, and scoring rules.

We compare FARE with four alternatives at the same bank size: random selection, clustering on scenario parameters, selection of the hardest scenarios, and an Order oracle that matches the full-pool probe ranking. Across three LLMs and both case studies, FARE's top-ranked candidate program has the highest pool score in five of six model–pool combinations and ties in the sixth. Its largest negative train–held-out gap, the overfitting direction, is -0.03, the smallest among the methods and tied with the Order oracle. The Order oracle baseline shows why faithful bank representation matters: its bank reproduces the pool's probe ranking almost exactly but leaves parts of the pool without a nearby scenario, and its programs score up to 0.12 lower than FARE's. Each scheduler evaluation on the FARE bank needs 8 scenario runs instead of 39, and the one-time fingerprinting cost is repaid after about 20 candidate programs.

2 Related Work

LLM-based algorithm and heuristic discovery. FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) established the paradigm of using LLMs as mutation operators inside an evolutionary loop over programs. Subsequent work has refined the search itself (Liu

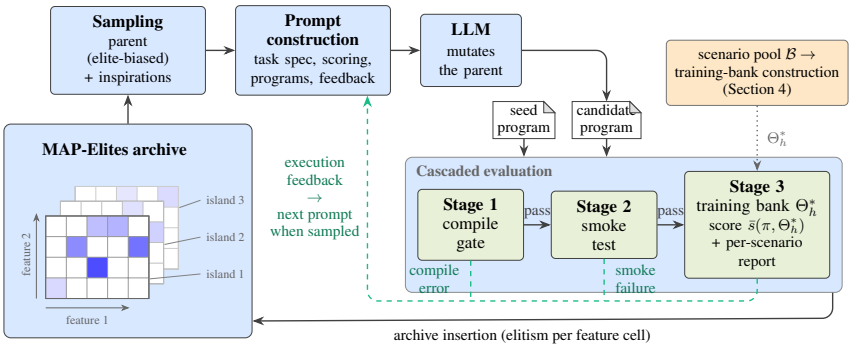

Figure 2: The discovery loop. The loop assembles a structured prompt containing a parent, inspirations, and prior execution feedback; the LLM proposes a candidate program (the evolve block); and the three-stage cascaded evaluation scores the candidate program on the FARE training bank.

et al., 2024; Ye et al., 2024; Sharma, 2025) and extended the paradigm to reward design (Ma et al., 2024). Recent work applies the paradigm to systems and networking heuristics (Cheng et al., 2025; Hamadanian et al., 2025; Dwivedula et al., 2025; Liao et al., 2025; Chen et al., 2026; Liu et al., 2026c). In these applications, discovered programs match or beat hand-engineered baselines on their discovery benchmarks. Most such systems treat the harness as a fixed, trusted input and give little attention to robustness beyond its author-chosen instances. We instead take the discovery loop as given and ask how to construct a training bank that makes the resulting harness scores predictive of performance under conditions unseen during search.

Benchmark subset selection and sample-efficient evaluation. Closest to our training-bank construction is work on compressing evaluation benchmarks: tinyBenchmarks fits item-response-theory models to a model-by-example score matrix and selects roughly a hundred examples that reproduce full-benchmark LLM scores (Maia Polo et al., 2024); Anchor Points exploits model-confidence correlations to rank models from a handful of instances (Vivek et al., 2024); and active testing adaptively selects test points to estimate loss (Kossen et al., 2021). Instance Space Analysis uses algorithm-performance profiles to diagnose benchmark diversity and bias (Smith-Miles & Muñoz, 2023); coresets select representative training examples (Sener & Savarese, 2018; Mirzasoleiman et al., 2020). These methods compress or diagnose fixed datasets for a static model pool. By contrast, our training-bank score becomes the objective of an adaptive search over a generated scenario pool, and that search may produce programs unlike the probe programs. We therefore evaluate both fidelity to the scenario pool and generalization to a search-unseen held-out test bank.

3 Discovery Loop

Our discovery loop (Figure 2) follows AlphaEvolve (Novikov et al., 2025). It stores candidate programs in a MAP-Elites archive (Mouret & Clune, 2015), which discretizes the feature space into cells and retains the highest-scoring candidate, the elite, in each cell. The archive is partitioned into islands that evolve independently and periodically exchange candidates. Within an island, the loop samples a parent from the elites with probability 0.7, uniformly from the full population with probability 0.2, and in proportion to score from the full population with probability 0.1. At each iteration, the LLM receives a prompt and proposes a candidate program. The prompt contains a fixed task specification and scoring rule, the parent, the highest-scoring and randomly selected programs (*inspirations*), and the parent's execution feedback, including compiler diagnostics and per-scenario evaluation reports. Appendix G gives the case-study templates (Listings 10–12).

Because each scenario run emulates the full system, evaluation cascades in three stages: a compile gate, a lightweight smoke test for crashes and invalid behavior, and evaluation on the full training bank. Failed stages return diagnostics, and successful evaluations return per-scenario reports. The next prompt includes this feedback; the final score and behavioral features decide whether and where the program enters the archive.

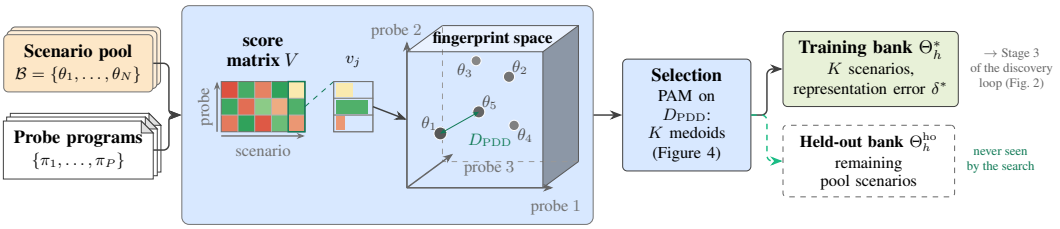

Figure 3: The FARE pipeline. Probe programs fingerprint the scenario pool $\mathcal{B}$; PAM selects K medoids under D_{PDD}, yielding the training bank Θ_h^* and representation error δ^*; and the unselected scenarios form the held-out test bank Θ_h^{ho}.

The training bank is therefore critical: it determines both the optimization signal that steers search and the cost of each candidate-program evaluation. Section 4 constructs a training bank compact enough for repeated evaluation yet representative of a much larger scenario pool.

4 FARE: Fingerprint-Aware Representative Evaluation

We consider heuristics deployed under a distribution $\mathcal{D}$ of operating conditions over a scenario parameter space Θ. Each scenario $\theta \in \Theta$ represents a trajectory of conditions. Exhaustive evaluation over Θ is infeasible, so we use a finite *scenario pool* $\mathcal{B} = \{\theta_1, \ldots, \theta_N\}$, sampled from $\mathcal{D}$ or assembled from real-world traces, as an empirical reference for $\mathcal{D}$.

Let $s(\pi, \theta) \in [0, 1]$ denote the score of program π on scenario θ. The mean score of program π over the scenario pool is $\bar{s}(\pi, \mathcal{B}) = \frac{1}{N} \sum_{\theta \in \mathcal{B}} s(\pi, \theta)$. Computing this mean for every candidate program is costly and wasteful when many scenarios are behaviorally redundant. We therefore construct a compact *training bank* $\Theta_h \subset \mathcal{B}$ of size $K \ll N$. The evaluation harness scores π on this bank by $\bar{s}(\pi, \Theta_h) = \frac{1}{K} \sum_{\theta \in \Theta_h} s(\pi, \theta)$. We choose Θ_h so that each of its K scenarios represents the pool scenarios it resembles, cutting each evaluation from N scenario runs to K. The bank serves as the objective that guides search, not merely as an estimator of the pool mean: we require every pool scenario to have a behaviorally similar scenario in Θ_h, so that no region of the pool is absent from the signal the discovery loop optimizes. Section 4.2 formalizes this requirement as coverage in fingerprint space, and Section 5.2 shows that banks built for coverage guide discovery better than banks that match the pool's probe ranking more closely.

4.1 Probe programs and fingerprints

To measure what each scenario in the pool reveals about program quality, FARE assembles P *probe programs* $\{\pi_1, \ldots, \pi_P\}$ that span a broad range of behaviors (Appendix D). It evaluates every probe program on every scenario and records the score matrix $V \in [0, 1]^{P \times N}$, where $V_{ij} = s(\pi_i, \theta_j)$. These PN runs are FARE's one expensive step; their cost is paid once and amortized over later searches.

To prevent probes with larger raw score ranges from dominating distances between scenarios, FARE normalizes each row of V to $[0, 1]$, yielding $\tilde{V}$. The *fingerprint* of scenario θ_j is the normalized column $v_j = \tilde{V}_{:,j} \in [0, 1]^P$, which captures scenario θ_j's difficulty profile as measured by the P probes. FARE defines *program disagreement distance* (PDD) between scenarios θ_j and $\theta_{j'}$ as the mean absolute difference between their fingerprints,

$$D_{\text{PDD}}(\theta_j, \theta_{j'}) \;=\; D_{\text{PDD}}(v_j, v_{j'}) \;:=\; \frac{1}{P} \sum_{i=1}^{P} \big| v_j(i) - v_{j'}(i) \big|. \tag{1}$$

Scenarios with PDD near zero provide essentially the same evidence about program quality, whereas scenarios with high PDD yield substantially different assessments.

4.2 CONSTRUCTING A REPRESENTATIVE TRAINING BANK

For a bank $\Theta_h \subset \mathcal{B}$, assign every pool scenario θ_j to its *representative* $r_{\Theta_h}(\theta_j) \in \Theta_h$, the selected scenario nearest to it under PDD:

$$r_{\Theta_h}(\theta_j) \;:=\; \arg\min_{\theta_m \in \Theta_h} D_{\mathrm{PDD}}(v_j, v_m).$$

FARE chooses the training bank of size K by minimizing the mean distance from each pool scenario to its representative,

$$\Theta_h^* \;=\; \arg\min_{\substack{\Theta_h \subset \mathcal{B} \\ |\Theta_h| = K}} \delta(\Theta_h), \qquad \text{where } \delta(\Theta_h) \;:=\; \frac{1}{N}\sum_{j=1}^{N} \min_{m \in \Theta_h} D_{\mathrm{PDD}}(v_j, v_m). \tag{2}$$

We call $\delta(\Theta_h)$ the *representation error* of Θ_h, and let $\delta^* = \delta^*(K) := \delta(\Theta_h^*)$ be the representation error of the training bank: the mean PDD between a pool scenario and its representative. A small δ^* means that pool scenarios have nearby representatives on average.

Assumption 1. *For any program π, there exist a* surrogate $g_\pi : [0,1]^P \to \mathbb{R}$ *and constants L_π, ϵ_π such that $|g_\pi(v) - g_\pi(v')| \le L_\pi D_{\mathrm{PDD}}(v, v')$, for all $v, v' \in [0,1]^P$, and $\big|s(\pi, \theta_j) - g_\pi(v_j)\big| \le \epsilon_\pi$ for all $j \in [N]$.*

For the probe programs, the assumption holds exactly ($\epsilon_{\pi_i} = 0$). For candidate programs produced by the discovery loop, the surrogate g_π captures the relationship between probe fingerprints and candidate scores, while ϵ_π quantifies the remaining approximation error. We use a fixed probe set in this work; periodically enriching it with newly discovered programs is a natural extension.

Proposition 1. *Under Assumption 1, for any bank Θ_h and any pool scenario θ_j with representative $\theta_{m(j)} = r_{\Theta_h}(\theta_j)$, the mean score error over the pool satisfies*

$$\frac{1}{N}\sum_{j=1}^{N} \big|s(\pi, \theta_j) - s(\pi, \theta_{m(j)})\big| \;\le\; L_\pi\, \delta(\Theta_h) + 2\epsilon_\pi.$$

Proposition 1 bounds the mean score error by the representation error. The mean score error is how far, on average, a pool scenario's score lies from its representative's score. It does not bound the gap between the bank's unweighted mean $\bar{s}(\pi, \Theta_h)$ and the pool mean $\bar{s}(\pi, \mathcal{B})$. That gap depends on how many pool scenarios each selected scenario represents, whereas the bank mean counts every selected scenario equally. We nonetheless minimize $\delta(\Theta_h)$. A small δ keeps every populated region of the pool close to some training scenario, so a candidate program cannot score well on the bank while failing an unrepresented part of the pool. Section 5.2 supports this choice empirically.

Selecting a size-K bank that minimizes $\delta(\Theta_h)$ is a discrete k-median problem under D_{PDD}, which is NP-hard (Kariv & Hakimi, 1979). We obtain an approximate solution using the Partitioning Around Medoids (PAM) heuristic (Kaufman & Rousseeuw, 1990) (Figure 4). PAM proceeds in two steps: BUILD constructs an initial bank by greedily selecting K medoids, and SWAP refines the bank by exchanging a selected medoid for an unselected scenario whenever the exchange reduces $\delta(\Theta_h)$.

Bank size selection. We choose K using scenario fingerprints. For each K, we rank the probes by their mean score on the training bank and compute the Spearman correlation $\rho(K)$ between this ranking and the full-pool ranking. We pick the elbow of this curve: the smallest K beyond which $\rho(K)$ improves little. This criterion fills the gap left by Proposition 1, which bounds per-scenario prediction error but not whether the mean used by the discovery loop preserves the ordering of programs; $\rho(K)$ checks that ordering directly.

Held-out validation. The representation error δ^* is static, whereas discovery adaptively selects candidate programs that perform best on Θ_h^* and may therefore overfit it (Dwork et al., 2015). We consequently reserve a disjoint held-out test bank Θ_h^{ho} and evaluate the top-ranked candidate programs on it only after discovery. The resulting train–held-out gap measures overfitting to the training bank, rather than generalization beyond the pool.

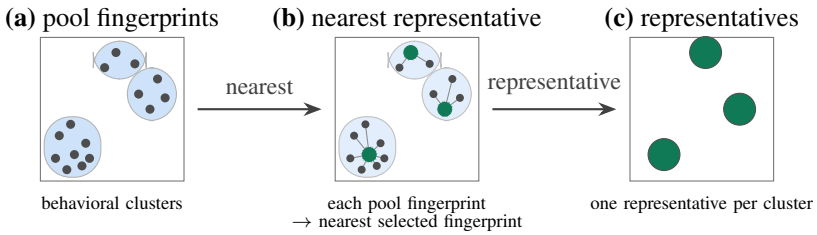

Figure 4: (a) The scenario pool's fingerprints form behavioral families. (b) Each pool fingerprint maps to its nearest selected fingerprint. (c) Each selected medoid is the representative of its cluster.

5 EXPERIMENTS

We test whether FARE builds compact training banks whose scenarios closely represent the scenario pool and that steer discovery toward candidate programs that generalize beyond the training scenarios. We compare five bank-construction methods end-to-end and examine scenario geometry, bank size, and evaluation cost. All evaluations run on 5G-EVAL, which patches each candidate program into a real 5G software stack, executes the given scenario bank, and scores the run from its telemetry (Appendix C). Appendix F provides more details and auxiliary results.

5.1 EXPERIMENTAL SETUP

5G applications and their scenario pools. We study two problems in the context of 5G mobile networks: scheduling and congestion control. In scheduling, user devices share the base station's limited bandwidth. Each scenario sets a latency target, and the objective is to meet that target while conserving bandwidth. At each scheduling interval (0.5 ms), the base station decides which user devices to serve and how much bandwidth to give each. In the explicit congestion notification (ECN)-marking problem, a user device's packets wait in a queue at the base station, and a congestion mark tells the traffic source to send more slowly. The objective is to keep the queue short enough to meet the latency target without sacrificing throughput. The policy sets the probability of marking each outgoing packet. Appendix C states the inputs, the scoring rules, and the seed programs. In each case, $P = 15$ probe programs define the fingerprints (Appendix D). The scheduler scenario pool contains 39 scenarios; the ECN-marking scenario pool contains 37. We write scheduler scenarios as $\mathrm{S}_i^{\mathrm{sch}}$ and ECN-marking scenarios as $\mathrm{S}_i^{\mathrm{ecn}}$. Appendix C.4 maps every scenario label to its parameters.

Bank-construction methods. We compare five methods at the same K. *FARE* selects K scenarios by PAM under D_{PDD} and scores each candidate program by its mean over those scenarios. *Random* draws K scenarios uniformly at random. *Metadata* applies PAM to Gower distances between scenario parameters (Appendix F.3, Equation 5). *Difficult* and *Order oracle* select scenarios directly from the probe score matrix V rather than from fingerprint distances. Difficult tests whether training on the hardest scenarios yields programs that generalize: it keeps the K scenarios with the lowest mean score across the P probes. Order oracle targets the pool's *probe ranking*, the ordering of the P probes by their mean score over all N pool scenarios, which is the ranking that ρ compares against. Starting from an empty bank, it greedily adds the scenario that most increases the overlap between the three top-ranked probes on the bank and the three top-ranked probes on the pool, breaking ties by the Spearman correlation between the bank and pool rankings (Appendix F.2).

Discovery protocol. All reported runs use five islands, 50 iterations, LLM sampling temperature 0.8, and cascade thresholds of 0.4 for both the smoke test and the full bank. Every end-to-end comparison uses `gemma4-31b` (hereafter Gemma4), `qwen3.5-397b-a17b` (hereafter Qwen3.5), and `gemini-3.1-pro` (hereafter Gemini3.1). Every scheduler run starts from `ltti.c` (Listing 1); every ECN-marking run starts from `lin_ecn.c` (Listing 2). We retain the top-ranked candidate program from each run and evaluate it after discovery. Appendix F.4 plots best-so-far scores, which converge before iteration 50.

Choosing K. Figure 5 shows how faithfully each training bank's mean-score ranking of the probe programs matches the full-pool ranking as K grows. FARE reaches $\rho = 0.86$ at $K = 8$ on the

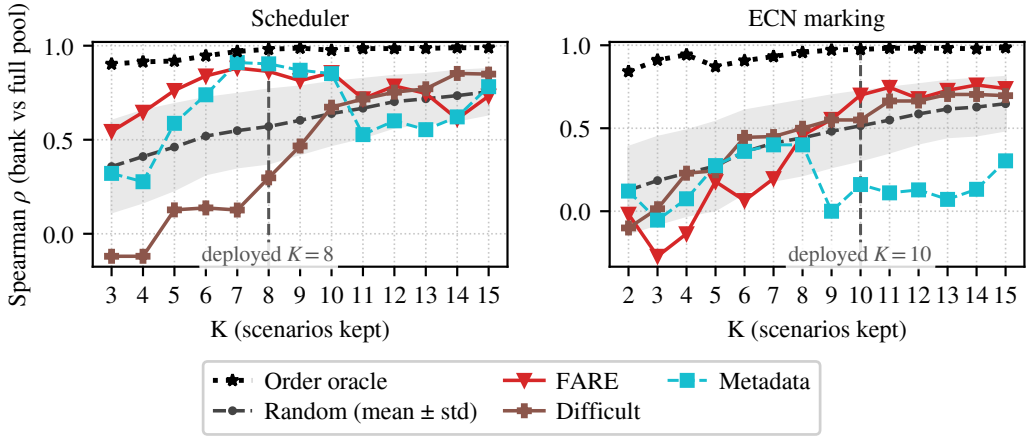

Figure 5: Spearman correlation ρ between each training bank's probe-program ranking and the full-pool ranking, against bank size K. The Random curve is the mean over 1,000 subsets, shaded to ± 1 standard deviation.

scheduler pool and $\rho = 0.70$ at $K = 10$ on the ECN-marking pool. We use these sizes; beyond them, extra scenarios add little rank fidelity for their cost. Figure 8 in Appendix F.1 plots $\delta^*(K)$. FARE attains the lowest representation error at every K, because that is the objective PAM minimizes. At the chosen K, the scheduler $\delta^*(8) = 0.152$ and the ECN-marking $\delta^*(10) = 0.050$.

5.2 End-to-end generalization

Table 1 compares the five methods on the scenario pool. The two *fidelity* columns describe the training bank alone. They are computed from the probe fingerprints before any discovery run, so all three models share them. δ is the bank's representation error (Equation 2), the mean D_{PDD} from a pool scenario to its nearest selected scenario; a lower δ means that every part of the pool has a behaviorally similar scenario in the bank. ρ is the Spearman correlation between the probe programs' ranking by mean score on the bank and their ranking on the full pool; $\rho = 1$ means the bank orders the probes exactly as the pool does. The *pool score* columns report the outcome of discovery: the mean score, over all pool scenarios, of each model's top-ranked candidate program at iteration 50. Higher is better, and the Seed program row is the reference point: a pool score below it means that discovery on that bank ended with a program worse than the one it started from. Reading across a row thus relates a bank's fidelity to the programs it produces, and the comparison below asks which of δ and ρ better predicts a high pool score. Figures 14 and 15 in Appendix F.5 give the per-scenario scores behind these means.

Behavioral selection. Against the three baselines that do not use the probe ranking, namely Random, Metadata, and Difficult, FARE's top-ranked candidate program has the highest pool score in five of the six model–pool combinations and is tied with Random for Gemma4 on ECN marking (Table 1). The margin over the next-best baseline is 0.04 (Gemma4), 0.03 (Qwen3.5), and 0.12 (Gemini3.1) on the scheduler. On ECN marking, FARE leads by 0.02 (Qwen3.5) and 0.01 (Gemini3.1). On the scheduler, FARE also beats the seed program for Gemma4, Qwen3.5, and Gemini3.1; Random, Metadata, and Difficult each fall below 0.71 for at least one model. Difficult, which fills the bank with the lowest-scoring scenarios, trails FARE in every combination by 0.03 to 0.12. On ECN marking, the Random subset used for discovery has $\rho = 0.69$, above the mean 0.51 in Figure 5 and close to FARE's 0.70, which may explain why Random nearly matches FARE there. Appendix E lists and describes the six top-ranked FARE programs.

Ordering alone is insufficient. The Order oracle greedily adds the scenario that most increases the overlap between the three top-ranked probes on the bank and the three top-ranked probes on the pool. The bank matches the full-pool probe ranking almost exactly ($\rho = 0.98$ on both pools), but its representation error is higher than FARE's (0.193 versus 0.152 on the scheduler, 0.083 versus 0.050

Table 1: Bank fidelity and end-to-end pool performance at the chosen K. Pool score is each model's top-ranked candidate program's mean over all pool scenarios, training and held-out alike. δ and ρ describe the training bank and are shared by Gemma4, Qwen3.5, and Gemini3.1. Random's δ and ρ are the fidelity of the subset used for those discovery runs. The Seed program row is the hand-written seed: LTTI on the scheduler pool and linear ECN on the ECN-marking pool.

	Scheduler (39 scenarios)					ECN marking (37 scenarios)				
	Fidelity		Pool score ↑			Fidelity		Pool score ↑		
Method	$\delta\downarrow$	$\rho\uparrow$	Gemma4	Qwen3.5	Gemini3.1	$\delta\downarrow$	$\rho\uparrow$	Gemma4	Qwen3.5	Gemini3.1
Seed program	—	—	0.71	0.71	0.71	—	—	0.61	0.61	0.61
FARE	0.152	0.86	0.72	0.73	0.80	0.050	0.70	0.71	0.69	0.72
Random	0.181	0.53	0.66	0.65	0.67	0.076	0.69	0.71	0.67	0.71
Metadata	0.183	0.90	0.63	0.70	0.66	0.094	0.16	0.68	0.67	0.70
Difficult	0.272	0.30	0.68	0.66	0.68	0.188	0.55	0.64	0.61	0.69
Order oracle	0.193	0.98	0.60	0.65	0.75	0.083	0.98	0.67	0.67	0.69

on ECN marking). On the scheduler, it places seven of its eight scenarios within four of FARE's eight clusters and leaves the other four, which hold 13 of the 39 pool scenarios, without a nearby representative. As a result, 31% of the pool lies farther than 0.3 in D_{PDD} from every selected scenario, versus 10% for FARE. Rank preservation constrains only the relative order of the P probe-program means, so a bank can reproduce that order while leaving part of the pool far from every selected scenario. Proposition 1's bound, which scales with $\delta(\Theta_h)$, is correspondingly looser. The pool scores follow the coverage gap. On the scheduler, FARE's top-ranked candidate program exceeds the Order oracle's by 0.12 (Gemma4), 0.08 (Qwen3.5), and 0.05 (Gemini3.1). On ECN marking, where the Order oracle leaves three smaller clusters (8 of 37 scenarios) unrepresented, FARE's top-ranked candidate program exceeds the Order oracle's by 0.04 (Gemma4), 0.02 (Qwen3.5), and 0.03 (Gemini3.1). A bank serves as a good training objective only if it covers the pool; ranking the probes correctly is not enough.

Cluster-mass weights. We also tried weighting each FARE scenario by its cluster mass (the share of the pool that scenario represents) and listing these weights in the prompt. On the probe programs, the rank correlation of this weighted bank mean with the pool ranking is 0.96 on the scheduler and 0.86 on ECN marking, versus 0.86 and 0.70 for the plain mean in Table 1. Discovery does not benefit from the weights. In one run per model on the scheduler, the top-ranked candidate program scores 0.62 (Gemma4) and 0.65 (Qwen3.5), versus 0.72 and 0.73 under the plain mean, and both fall below the seed program (0.71). On ECN marking the weighted programs are no better than FARE: 0.69 versus 0.71 for Gemma4, and 0.67 versus 0.69 for Qwen3.5.

Train–held-out gaps. For each method, we compare the mean score of its top-ranked candidate program on the method's training bank with its mean on the method's held-out test bank. Figure 16 in Appendix F.5 reports this gap. Each method selects its own training scenarios, so the held-out test banks differ across rows. Across both pools and all three models, FARE's training-bank mean lies within 0.09 of its held-out mean (held-out minus training, for Gemma4, Qwen3.5, and Gemini3.1: $+0.02$, $+0.05$, and $+0.09$ on the scheduler; -0.01, -0.03, and -0.01 on ECN marking). A negative gap is the overfitting direction: the training bank scores the program higher than the held-out test bank. FARE's largest negative gap is -0.03, the smallest among the methods and tied with the Order oracle. The remaining methods' largest negative gaps are -0.05 (Difficult), -0.06 (Random), and -0.08 (Metadata). This is consistent with a bank whose low δ^* lets its mean score stand in for the rest of the pool. Difficult's gaps are large and positive except for Qwen3.5 on the scheduler (-0.05): those training banks are harder than their held-out scenarios, so the scores that guide search understate held-out performance.

Transfer beyond the probes. We test Assumption 1 on candidate programs produced after the fingerprints were fixed. The scheduler case study supplies 15 such programs with a full-pool evaluation: the top-ranked candidate program of each method under each of the three models. Their pool means span 0.60 to 0.80, and each program's training-bank mean tracks its own pool mean to within

0.04 on average and 0.15 at worst. Aggregate scores therefore transfer; the Lipschitz constants do not. At the Lipschitz scale $L_\pi = 14.6$ that the probe programs themselves require, the tight residual is $\epsilon_\pi = 0.23$ for the median program, and only four of the 741 scenario pairs yield a nonzero residual for any of the 15 programs. The worst pair is $\mathrm{S}_{27}^{\mathrm{sch}}$ and $\mathrm{S}_{38}^{\mathrm{sch}}$, where the representative itself fails (Section 6). Ranking the 15 probe programs together with the five top-ranked Qwen3.5 candidate programs by their mean on the FARE bank reproduces the pool ranking at $\rho = 0.76$. Of the 190 pairs (none tied), 40 are reversed, and each reversed pair differs by at most 0.11 in pool mean. Table 9 in Appendix F.5 reports the same statistics for Random, Metadata, Difficult, and the Order oracle. On the ECN-marking pool, the corresponding 15 programs have residual $\epsilon_\pi = 0$ at the probes' Lipschitz scale $L_\pi = 14.8$, and each program's training-bank mean tracks its pool mean to within 0.05 on average and 0.22 at worst.

5.3 Scenario-bank fidelity

Metadata does not recover behavioral geometry. The Metadata baseline clusters pool scenarios by their parameters, avoiding probe runs, but similar parameters do not make two scenarios close under D_{PDD}. Gower distance D_{gower} correlates with D_{PDD} at Mantel $\rho = 0.23$[2] on the scheduler pool and $\rho = 0.22$ on the ECN-marking pool. Scenarios that match on those parameters remain nearly as distant under D_{PDD} as a random pair (mean 0.347 versus the pool mean 0.434 on the scheduler, and 0.270 versus 0.287 on ECN marking). A Gower-medoid training bank therefore cannot substitute for probe-program fingerprints. Appendix F.3 compares D_{PDD} for pairs that share a parameter value with pairs that do not, and plots D_{gower} against D_{PDD} for all pairs.

Evaluation cost. Each scenario run lasts two minutes. Scoring a candidate program on a $K = 8$ scheduler training bank requires eight scenario runs (16 minutes); scoring it on a $K = 10$ ECN-marking training bank requires ten (20 minutes). Full-pool evaluation requires 39 and 37 scenario runs (78 and 74 minutes). Building the $P = 15$ fingerprint matrix requires $15 \times 39 = 585$ scheduler scenario runs (19.5 hours) and $15 \times 37 = 555$ ECN-marking scenario runs (18.5 hours). Compared with evaluating every candidate program on the full pool, this one-time cost is repaid after about 19 scheduler candidate programs and 21 ECN-marking candidate programs.

6 Limitations

A pool scenario can lie very close to its representative, adding almost nothing to δ^*, even when later candidate programs separate the two. On the scheduler pool, the FARE medoid $\mathrm{S}_{38}^{\mathrm{sch}}$ represents $\mathrm{S}_{27}^{\mathrm{sch}}$ and 11 other scenarios, the largest cluster in the bank. Their fingerprints lie $D_{\mathrm{PDD}} = 0.007$ apart, yet the 15 top-ranked candidate programs separate them by up to 0.91; FARE's own programs score 0.38 on the medoid and 0.95 on $\mathrm{S}_{27}^{\mathrm{sch}}$, so the bank underestimates the easy scenario. Proposition 1 does not exclude this: at this distance its bound is $L_\pi \cdot 0.007 + 2\epsilon_\pi \approx 0.10 + 0.46 = 0.56$ at the median residual, still below the 0.91 gap.

7 Conclusion

In LLM-based heuristic discovery, the scenario bank does more than measure programs: it sets the objective the discovery loop optimizes. FARE fingerprints pool scenarios by probe-program scores, selects a compact training bank by k-medoids under program disagreement distance, and holds out the rest. Its representation error δ^* measures, *a priori*, how well the bank covers the pool, and the train–held-out gap measures overfitting empirically. On two 5G control tasks on 5G-Eval and across three LLMs, FARE's top-ranked candidate program has the highest pool score in almost all model–pool combinations. Although the Order oracle matches the pool's probe ranking almost exactly, it yields programs that score up to 0.12 lower, suggesting that bank quality depends on pool coverage, not just ranking known programs. That coverage, however, is only as complete as the probe set that measures it: because the fingerprints come from a fixed probe set, FARE cannot detect distinctions that only later programs expose. Section 6 shows one such case. Natural next steps are to add discovered programs to the probe set and rebuild the bank during search, and to apply FARE to other systems domains where the scenario space is too large to evaluate exhaustively.

[2]Mantel ρ is the Spearman correlation of the two pairwise-distance matrices.

ETHICS STATEMENT

Our experiments involve no human participants or personal data. They execute generated C programs only in a self-contained software emulation. Compile, runtime, timing, and starvation checks reject faulty programs, and failure recovery isolates runs. Automated generation of network-control code carries dual-use and deployment risks: an unvalidated program could degrade service or exhaust resources. We therefore present FARE as a controlled evaluation method, not as evidence that discovered programs are safe for direct deployment.

REPRODUCIBILITY STATEMENT

Appendix B states the method's assumptions and gives complete proofs. Section 5 and Appendix F specify the scenario pools, training banks, search settings, baselines, metrics, and evaluation protocol. Appendix C documents the hardware, scenario schema and seeds, automation, failure recovery, and telemetry; Appendix G reproduces the prompt templates. We will release the source code upon acceptance.

AI USE STATEMENT

Generative AI is intrinsic to this work: the discovery loop uses LLMs as mutation operators that rewrite candidate programs (Section 3). We also used generative AI to implement components of 5G-EVAL and plotting code, and to polish the writing. The authors reviewed and validated all AI-assisted code and prose; they take responsibility for the final content.

REFERENCES

Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010. doi: 10.1145/1851182.1851192.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety, 2016. URL https://arxiv.org/abs/1606.06565.

Bob Briscoe, Koen De Schepper, Marcelo Bagnulo, and Greg White. Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture. RFC 9330, 2023. URL https://www.rfc-editor.org/info/rfc9330.

Hongzheng Chen, Alexander Novikov, Ngân Vũ, Hanna Alam, Zhiru Zhang, Aiden Grossman, Mircea Trofin, and Amir Yazdanbakhsh. Magellan: Autonomous Discovery of Novel Compiler Optimization Heuristics with AlphaEvolve. 2026. URL https://arxiv.org/abs/2601.21096.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Shubham Agarwal, Mert Cemri, Bowen Wang, Alexander Krentsel, Tian Xia, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Ashwin Naren, Shulu Li, Ruiying Ma, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. How AI Can Accelerate Systems Performance Research, 2025. URL https://arxiv.org/abs/2512.14806.

Yuhang Chu, Changjun Wang, Mingyue Ma, Guangming Zhao, Zelin Miao, and Weiqing Wang. Urban High-Risk Scenarios for Automated Vehicle Safety Testing: A Generation and Generalization Method Based on Accident Data. *Accident Analysis & Prevention*, 231:108480, 2026. doi: 10.1016/j.aap.2026.108480.

Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning. *Journal of Machine Learning Research*, 23(226):1–61, 2022.

Arnaud Dethise, Marco Canini, and Srikanth Kandula. Cracking open the black box: What observations can tell us about reinforcement learning agents. In *Proceedings of the 2019 Workshop on Network Meets AI & ML (NetAI)*, pp. 29–36, 2019.

Rohit Dwivedula, Divyanshu Saxena, Sujay Yadalam, and Daehyeok Kim. Instance-Optimal Systems Heuristics Through LLM-Driven Search, 2025. URL https://arxiv.org/abs/2512.25065.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. Preserving statistical validity in adaptive data analysis. In *ACM Symposium on Theory of Computing (STOC)*, 2015.

Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 305–318, 2021.

Maxime Elkael, Michele Polese, Reshma Prasad, Stefano Maxenti, and Tommaso Melodia. ALLSTaR: Automated LLM-driven scheduler generation and testing for intent-based RAN, 2025. URL https://arxiv.org/abs/2505.18389.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation. Zenodo, version v0.0.1, September 2021. URL https://doi.org/10.5281/zenodo.5371628.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning (ICML)*, pp. 10835–10866, 2023.

Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, pp. 85–92, 2019.

Charles A E Goodhart. Problems of monetary management: The UK experience. In *Monetary Theory and Practice: The UK Experience*, pp. 91–121. Macmillan Education UK, 1984.

J. C. Gower. A General Coefficient of Similarity and Some of Its Properties. *Biometrics*, 27(4): 857–871, 1971. doi: 10.2307/2528823.

Joshua Groen, Mauro Belgiovine, Utku Demir, Brian Kim, and Kaushik Chowdhury. TRACTOR: Traffic analysis and classification tool for open RAN. In *IEEE International Conference on Communications (ICC)*, pp. 4894–4899, 2024.

Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. A Human-Inspired AI for Automated Systems Design and Optimization, 2025. URL https://arxiv.org/abs/2510.27176.

Christian Hennig. Cluster-wise assessment of cluster stability. *Computational Statistics & Data Analysis*, 52(1):258–271, 2007. doi: 10.1016/j.csda.2006.11.025.

Aleksandar Jalali, Roberto Padovani, and Rajesh Pankaj. Data throughput of CDMA-HDR a high efficiency-high data rate personal communication wireless system. In *Proceedings of the 51st IEEE Vehicular Technology Conference (VTC 2000-Spring)*, volume 3, pp. 1854–1858, 2000.

Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning (ICML)*, pp. 3050–3059, 2019.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Ingemar Johansson and Zaheduzzaman Sarker. Self-Clocked Rate Adaptation for Multimedia. RFC 8298, 2017. URL https://www.rfc-editor.org/info/rfc8298.

Florian Kaltenberger, Aloizio P. Silva, Abhimanyu Gosain, Luhan Wang, and Tien-Thinh Nguyen. OpenAirInterface: Democratizing innovation in the 5G Era. *Computer Networks*, 176:107284, 2020. doi: 10.1016/j.comnet.2020.107284.

Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. Robust heuristic algorithm design with LLMs. Technical Report MSR-TR-2025-44, Microsoft Research, 2025. arXiv:2510.08755.

Oded Kariv and S. L. Hakimi. An algorithmic approach to network location problems. II: The p-medians. *SIAM Journal on Applied Mathematics*, 37(3):539–560, 1979.

Leonard Kaufman and Peter J Rousseeuw. Partitioning around medoids (Program PAM). In *Finding Groups in Data: An Introduction to Cluster Analysis*, pp. 68–125. John Wiley & Sons, 1990.

Xinyi Ke, Kai Li, Junliang Xing, Yifan Zhang, and Jian Cheng. Game-theoretic co-evolution for LLM-based heuristic discovery. In *International Conference on Machine Learning (ICML)*, 2026.

Woo-Hyun Ko, Ushasi Ghosh, Ujwal Dinesha, Raini Wu, Srinivas Shakkottai, and Dinesh Bharadia. EdgeRIC: Empowering real-time intelligent optimization and control in NextG cellular networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1315–1330, 2024.

Jannik Kossen, Sebastian Farquhar, Yarin Gal, and Tom Rainforth. Active testing: Sample-efficient model evaluation. In *International Conference on Machine Learning (ICML)*, pp. 5753–5763, 2021.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. 2025. URL https://arxiv.org/abs/2512.23236.

Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning (ICML)*, pp. 6237–6247, 2020.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning (ICML)*, 2024.

Fei Liu, Alessio Figalli, Patrick Owen, and Nicola Serra. RAISE: LLM-based automated heuristic design with robust adversary instance search. *arXiv preprint arXiv:2606.31801*, 2026a.

Fei Liu, Yilu Liu, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. EoH-S: Evolution of heuristic set using LLMs for automated heuristic design. In *AAAI Conference on Artificial Intelligence*, volume 40, pp. 37090–37098, 2026b.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. 2026c. URL https://arxiv.org/abs/2602.23413.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinyBenchmarks: Evaluating LLMs with fewer examples. In *International Conference on Machine Learning (ICML)*, pp. 34303–34326, 2024.

Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning (ICML)*, pp. 6950–6960, 2020.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Kathleen Nichols and Van Jacobson. Controlling queue delay. *ACM Queue*, 10(5), 2012. URL https://queue.acm.org/detail.cfm?id=2209336.

Kathleen Nichols, Van Jacobson, Andrew McGregor, and Janardhan Iyengar. Controlled Delay Active Queue Management. RFC 8289, 2018. URL https://www.rfc-editor.org/info/rfc8289.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J R Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Open5GS Project. Open5GS, 2026. URL https://open5gs.org/.

Rong Pan, Preethi Natarajan, Fred Baker, and Greg White. Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem. RFC 8033, 2017. URL https://www.rfc-editor.org/info/rfc8033.

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In *International Conference on Machine Learning (ICML)*, pp. 5389–5400, 2019.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco J R Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Robert Schmidt, Mikel Irazabal, and Navid Nikaein. FlexRIC: An SDK for next-generation SD-RANs. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pp. 411–425, 2021. doi: 10.1145/3485983.3494870.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations (ICLR)*, 2018.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent, 2025. URL https://github.com/codelion/openevolve.

Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, Zhiguang Cao, and Jie Zhang. Generalizable heuristic generation through LLMs with meta-optimization. In *International Conference on Learning Representations (ICLR)*, 2026.

Kevin Sim, Quentin Renau, and Emma Hart. Beyond the hype: Benchmarking LLM-evolved heuristics for bin packing. In *Applications of Evolutionary Computation (EvoApplications)*. Springer, 2025.

Joar Skalse, Nikolaus H R Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9460–9471, 2022.

Kate Smith-Miles and Mario Andrés Muñoz. Instance space analysis for algorithm testing: Methodology and software tools. *ACM Computing Surveys*, 55(12):1–31, 2023.

Marilyn Strathern. 'Improving ratings': Audit in the British university system. *European Review*, 5 (3):305–321, 1997.

Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *ACM SIGMETRICS Performance Evaluation Review*, 49(2):43–46, 2022.

Rajan Vivek, Kawin Ethayarajh, Diyi Yang, and Douwe Kiela. Anchor Points: Benchmarking models with much fewer examples. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 1576–1601, 2024.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Mohammad Zangooei, Jannis Weil, Amr Rizk, Mina Tahmasbi Arashloo, and Raouf Boutaba. Analyzing symbolic properties for DRL agents in systems and networking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 10(2):1–27, 2026. doi: 10.1145/3805627.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Dylan Slack, Qin Lyu, et al. A careful examination of large language model performance on grade school arithmetic. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024.

A Additional Related Work

Section 2 positions FARE among LLM-based discovery and benchmark-subset selection. This appendix covers the remaining related work.

Robust heuristic discovery. Broad benchmarking shows that bin-packing heuristics produced by LLM-based discovery often specialize to the instance families used during discovery (Sim et al., 2025). Recent methods respond by evolving complementary heuristic portfolios (Liu et al., 2026b), using multi-task meta-optimization (Shi et al., 2026), co-evolving solvers and instance generators (Ke et al., 2026), searching for nearby adversarial instances (Liu et al., 2026a), or synthesizing region-specific heuristics from diagnosed failures (Karimi et al., 2025). These approaches alter the search, its scenarios, or its output. FARE is complementary: given a scenario pool sampled from a declared operating distribution, it constructs a compact training bank and a search-unseen held-out test bank.

Optimization-induced benchmark overfitting. Goodhart's law warns that a proxy can fail once it becomes an optimization target (Goodhart, 1984; Strathern, 1997); in machine learning this appears as reward hacking and specification gaming (Amodei et al., 2016; Skalse et al., 2022), and reward-model studies measure how true performance degrades as optimization pressure grows (Gao et al., 2023). Evaluations on fixed benchmarks may overstate capability when the benchmarks are contaminated, underspecified, or repeatedly reused (Recht et al., 2019; Zhang et al., 2024; D'Amour et al., 2022). The adaptive-data-analysis literature formalizes the resulting loss of statistical validity (Dwork et al., 2015). LLM-based discovery creates this failure mode inside a single run. Because the harness selects each generation using feedback from the same scenarios, it both evaluates and adaptively shapes the programs it later judges. FARE reduces the proxy–target discrepancy and measures residual adaptive overfitting on a search-unseen held-out test bank.

Learning-based network control. Prior work applies learning to congestion control and to radio-access scheduling and slicing (Jay et al., 2019; Tessler et al., 2022; Ko et al., 2024). The deployment literature emphasizes brittle generalization and the need to interpret or verify learned policies (Gilad et al., 2019; Dethise et al., 2019; Eliyahu et al., 2021; Zangooei et al., 2026). This literature motivates trustworthy network-control evaluation but mainly examines neural policies; we study the evaluation harness that steers LLM-based discovery of executable control programs.

B Proofs and Discussion

This appendix proves Proposition 1 and explains why the residual ϵ_π in Assumption 1 remains substantive for candidate programs.

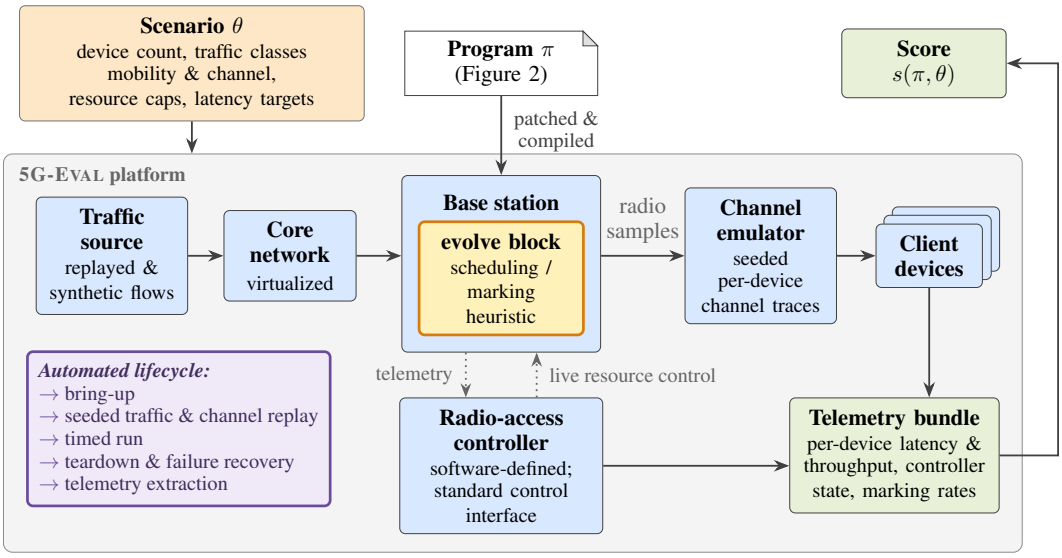

Figure 6: The 5G-EVAL platform. A scenario configuration θ (a single declarative file) instantiates a fully automated experiment on real cellular network software: a traffic source drives flows through a virtualized core network to a base station, which exchanges radio samples with user devices over a software channel emulator; a software-defined radio-access controller applies live resource control through a standard interface. 5G-EVAL patches the discovery loop's candidate program π into the base station, where the highlighted evolve block implements the heuristic under search. From each run's uniform telemetry bundle, 5G-EVAL computes the scenario score $s(\pi, \theta)$ and the execution feedback described in Section 3.

B.1 PROOF OF PROPOSITION 1

Fix θ_j and its representative $\theta_{m(j)}$. By Assumption 1, $|s(\pi, \theta_j) - g_\pi(v_j)| \le \epsilon_\pi$ and $|s(\pi, \theta_{m(j)}) - g_\pi(v_{m(j)})| \le \epsilon_\pi$. Because g_π is L_π-Lipschitz with respect to D_{PDD}, $|g_\pi(v_j) - g_\pi(v_{m(j)})| \le L_\pi D_{\mathrm{PDD}}(v_j, v_{m(j)})$. The triangle inequality

$$|s(\pi, \theta_j) - s(\pi, \theta_{m(j)})| \le |s(\pi, \theta_j) - g_\pi(v_j)| + |g_\pi(v_j) - g_\pi(v_{m(j)})| + |g_\pi(v_{m(j)}) - s(\pi, \theta_{m(j)})|$$

gives the pointwise bound $|s(\pi, \theta_j) - s(\pi, \theta_{m(j)})| \le L_\pi D_{\mathrm{PDD}}(v_j, v_{m(j)}) + 2\epsilon_\pi$. Averaging over $j = 1, \ldots, N$ and using $D_{\mathrm{PDD}}(v_j, v_{m(j)}) = \min_{m \in \Theta_h} D_{\mathrm{PDD}}(v_j, v_m)$ gives the pool-averaged bound $L_\pi \delta(\Theta_h) + 2\epsilon_\pi$ by the definition of δ.

B.2 WHY THE RESIDUAL ϵ_π IS SUBSTANTIVE

The score map $s(\pi, \cdot)$ is defined on scenarios, whereas the surrogate g_π sees only fingerprints. Because the fingerprint map $\phi : \theta \mapsto v_\theta$ can be many-to-one, no function of v alone can, in general, reproduce $s(\pi, \cdot)$ exactly. Assumption 1 instead requires the score map to factor through ϕ up to ϵ_π: $|s(\pi, \theta_j) - g_\pi(v_j)| \le \epsilon_\pi$ for every j. This requirement is substantive because ϕ discards information. Suppose θ and θ' have the same fingerprint but differ in a burst pattern that every probe program handles well. A future candidate program might score 0.9 on θ and 0.2 on θ'. Every surrogate g_π receives the same v for both scenarios, so a single value $g_\pi(v)$ cannot approximate both scores with error below 0.35. In general, the residual ϵ_π is small only when scenarios with the same fingerprint also receive similar scores from π.

C 5G-EVAL PLATFORM AND CASE STUDIES

We apply FARE to construct the scenario-bank component of two evaluation harnesses implemented on 5G-EVAL, an automated evaluation platform built on real cellular network software.

C.1 THE 5G-EVAL PLATFORM

5G-EVAL is the evaluation platform shared by both case studies. It combines (1) an open-source base-station and user-device stack (Kaltenberger et al., 2020) that exchanges radio samples through a software channel emulator, allowing the full physical and link layers to execute without radio hardware; (2) a virtualized core network (Open5GS Project, 2026); and (3) a software-defined radio-access controller (Schmidt et al., 2021) that applies live resource control through a standard interface. Traffic comprises replayed flows from three classes (bulk-data, latency-critical, and sparse machine-type), plus constant-bitrate flows and a real-time media flow governed by a delay-based congestion controller (Johansson & Sarker, 2017) with ECN support (Briscoe et al., 2023). A single declarative configuration file encodes each scenario $\theta \in \Theta$ (Section 4). The bank-construction pipeline and every evaluation stage consume this file unchanged.

5G-EVAL turns each scenario into a fully automated, reproducible experiment that runs unattended. A single invocation brings up the network, replays seeded per-user-device traffic (Groen et al., 2024) and channel traces (Ko et al., 2024), runs the timed experiment, tears everything down, and extracts a uniform telemetry bundle containing per-user-device latency and throughput, per-scheduling-interval controller state, and congestion-marking rates. The evaluator computes the scenario score and LLM feedback from this bundle. The evaluator also recovers from failures by resetting processes and shared state between runs and performing a hard reset after repeated failures.

C.2 CASE STUDY 1: SCHEDULING

The first case study evolves a scheduler. At each scheduling interval, the scheduler ranks the backlogged user devices and decides how many bandwidth units each may consume. The evolve block contains two co-evolved pure-C functions: a priority function and a bandwidth-sizing function. The priority function observes the user device's queueing delay (time since its oldest pending packet arrived), bytes waiting in its buffer, the transmission size it could currently achieve, an exponentially weighted average of its recent throughput, an indicator of its current channel quality, and its latency target. The bandwidth-sizing function additionally observes the remaining bandwidth budget and the number of competing user devices, allowing a candidate program to trade off deadline urgency, fairness, and resource efficiency.

The scheduler seed is Latency-Threshold Transmission Time Interval (LTTI) (Elkael et al., 2025), a latency-targeted scheduler whose exponential urgency term increases as queueing delay approaches the target. A user device near the target therefore outranks one whose only advantage is a better channel. LTTI then allocates bandwidth greedily: in rank order, it offers each scheduled user device the full remaining contiguous span, so the first-ranked user device can consume the entire remaining budget before the next device is considered.

We restrict candidate programs to short, deterministic code and prohibit dynamic memory allocation. The evaluator sanitizes invalid outputs, including negative values, NaNs, and infinities, and counts each as a violation; one violation disqualifies the candidate program. The evaluator also disqualifies a program whose mean per-interval scheduling time exceeds the fixed real-time budget or that starves a user device. A user device counts as starved if it stays backlogged but is almost never scheduled.

The per-scenario scheduler score is

$$s_{\text{sched}}(\pi, \theta) = \tfrac{1}{2}\,\mathbf{1}\{\text{p95 latency} < \Delta_\theta\} + \tfrac{1}{2}\max\left(0, 1 - \frac{\text{p95 bandwidth usage}}{\text{capacity}_\theta}\right), \qquad (3)$$

where Δ_θ is the scenario's class-dependent latency target and p95 denotes the 95th percentile. The two terms are additive rather than gated. Meeting the tail-latency target earns a fixed half of the score, and the other half rewards using fewer radio resources whether or not the target is met. A program that narrowly misses the target thus still earns credit for efficiency instead of falling off a score cliff. The archive indexes candidate programs by average latency slack and average bandwidth headroom. Appendix G.2 (Listing 11) reproduces the exact system message supplied to the LLM for this case study. Every scheduler discovery run starts from the LTTI evolve block in Listing 1.

Listing 1: Evolve block of `ltti.c`, the seed program for every scheduler discovery run. The priority function implements the LTTI baseline; the bandwidth-sizing function returns the full contigu-

ous span. The scaffolding above the block, omitted here with the file's header comment, defines `EVOLVED_EXP_OVERFLOW` as `1e10f`.

```
float evolved_dl_score(const evolved_score_ctx_t *ctx,
                       const evolved_score_input_t *input)
{
    const float R_inst = (float)input->tbs;
    const float pf_ratio = R_inst / input->R_avg;

    const double D_HOL_s = input->hol_delay_us / 1e6;
    const double denom = ctx->delta_s - D_HOL_s;

    float delay_mult;
    if (denom <= 1e-9) {
        delay_mult = EVOLVED_EXP_OVERFLOW;
    } else {
        const double exp_arg = ctx->delta_s / denom;
        delay_mult = (float)exp(exp_arg);
        if (delay_mult > EVOLVED_EXP_OVERFLOW || !isfinite(delay_mult))
            delay_mult = EVOLVED_EXP_OVERFLOW;
    }

    return (float)ctx->qos_weight * pf_ratio * delay_mult;
}

uint16_t evolved_dl_prb_size(const evolved_score_ctx_t *ctx,
                             const evolved_prb_size_input_t *input)
{
    (void)ctx;
    return input->max_contiguous_prbs;
}
```

Introduction figure (Figure 1). Figure 1 previews this case study's central generalization result. For each of four LLMs (L1–L4: `gemini-3.1-pro`, `gemma4-31b`, `gpt-5.4`, `qwen3.5-397b-a17b`), the figure shows the top-ranked candidate program from one 50-iteration, five-island discovery run on $\Theta_h^{\mathrm{sch},A}$, selected by its final training-bank score $\bar{s}_{\mathrm{sched}}$. Scenarios $\mathrm{S}_1^{\mathrm{sch}}$–$\mathrm{S}_5^{\mathrm{sch}}$ form $\Theta_h^{\mathrm{sch},A}$; $\mathrm{S}_6^{\mathrm{sch}}$–$\mathrm{S}_8^{\mathrm{sch}}$ form its held-out test bank. Table 3 maps these labels to their parameters. Every cell reports the per-scenario score $s_{\mathrm{sched}}(\pi, \theta)$ defined above.

The training scenarios and the held-out scenarios differ in channel dynamics and traffic class. $\mathrm{S}_1^{\mathrm{sch}}$–$\mathrm{S}_5^{\mathrm{sch}}$ use only static channel profiles, and each uses a 100% resource cap. $\mathrm{S}_1^{\mathrm{sch}}$ and $\mathrm{S}_3^{\mathrm{sch}}$ place two user devices on static line-of-sight channels and carry bulk-data traffic (300 ms latency target) and latency-critical traffic (50 ms), respectively. $\mathrm{S}_2^{\mathrm{sch}}$ and $\mathrm{S}_4^{\mathrm{sch}}$ raise the count to four user devices and mix static line-of-sight with static non-line-of-sight channels, again for those two traffic classes. $\mathrm{S}_5^{\mathrm{sch}}$ places six user devices on the same static profiles and carries sparse machine-type traffic (1000 ms). The held-out scenarios are mobile mixes at the same full resource cap. $\mathrm{S}_6^{\mathrm{sch}}$ places five bulk-data user devices on car, drone, robot, and rotational channels. $\mathrm{S}_7^{\mathrm{sch}}$ places six latency-critical user devices on that mobile mix together with one static non-line-of-sight channel. $\mathrm{S}_8^{\mathrm{sch}}$ places six constant-bitrate user devices, offered at heterogeneous rates, on a mobile mix, and sets a 200 ms latency target. Discovery therefore optimizes against static channels and three traffic classes (bulk-data, latency-critical, and sparse machine-type). The held-out columns test time-varying mobility, and $\mathrm{S}_8^{\mathrm{sch}}$ adds constant-bitrate traffic.

C.3 Case study 2: ECN marking

The ECN-marking case study evolves a network-side congestion-marking policy for L4S (Briscoe et al., 2023), the IETF-standardized architecture for low-latency, low-loss, scalable-throughput service. At each scheduling interval, the base station calls a pure-C function for each per-user-device queue; it returns an integer from 0 to 10,000; each unit represents 0.01 percentage points of marking probability. The base station applies this probability to eligible outgoing packets before transmission, and the receiver runs a delay-based congestion controller (Johansson & Sarker, 2017) that reacts to marks by reducing its sending rate. The policy observes queue occupancy in bytes and packets, queueing delay, an exponentially weighted average of packet sojourn time (time spent in the queue), bytes served in the previous interval, an indicator of channel quality, an exponentially weighted average of the user device's throughput, and the flow's latency target.

The marking score uses the same additive service-level structure as the scheduler score but replaces bandwidth efficiency with useful delivered throughput:

$$s_{\text{ecn}}(\pi, \theta) = \tfrac{1}{2}\,\mathbf{1}\{\text{p95 latency} < \Delta_\theta\} + \tfrac{1}{2}\min\left(1, \frac{\text{p50 throughput}}{\text{peak}_\theta}\right). \tag{4}$$

Here peak_θ is the flow's peak achievable rate implied by the scenario's bandwidth cap, and p50 denotes the 50th percentile. This score penalizes both under-marking, which misses the latency target, and over-marking, which suppresses the controller's throughput. The evaluator disqualifies a candidate program that silences the flow, as indicated by zero eligible packets or no usable latency samples after the warm-up period. The archive indexes candidate programs by average latency slack and average marking rate.

Listing 2: Evolve block of `lin_ecn.c`, the seed program for every ECN-marking discovery run. Marking probability rises linearly with queue occupancy between two byte thresholds that the seed derives from the latency target. The scaffolding above the block, omitted here with the file's header comment, defines `NOMINAL_RATE_KBPS` as 10,000 kbps, `LO_FRAC_PCT` as 10, `HI_FRAC_PCT` as 50, `FALLBACK_LO_BYTES` as 8,000, and `FALLBACK_HI_BYTES` as 20,000.

```
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    uint32_t lo, hi;
    if (ctx->target_delay_ms == 0u) {
        lo = FALLBACK_LO_BYTES;
        hi = FALLBACK_HI_BYTES;
    } else {
        /* bytes = (rate_kbps * 1000 / 8) * (frac_pct/100 * target_ms/1000)
         *       = rate_kbps * frac_pct * target_ms / 800
         * Compute in 64-bit to keep large targets safe, then narrow. */
        const uint64_t k = (uint64_t)NOMINAL_RATE_KBPS * ctx->target_delay_ms;
        lo = (uint32_t)((k * LO_FRAC_PCT) / 800u);
        hi = (uint32_t)((k * HI_FRAC_PCT) / 800u);
        if (hi <= lo) hi = lo + 1u; /* defensive: keep slope finite */
    }

    const uint32_t v = input->bytes_in_buffer;
    if (v <= lo)  return 0;
    if (v >= hi)  return 10000;
    return (uint16_t)(((uint64_t)(v - lo) * 10000u) / (hi - lo));
}
```

The two case studies are distinct harnesses. The scheduler task is a multi-user resource-allocation problem in which the score rewards latency compliance and bandwidth efficiency; the ECN-marking task is a single-flow congestion-signaling problem in which the score rewards latency compliance and throughput preservation. Because they share infrastructure and method but differ in interface, the two case studies show that FARE applies to more than one heuristic interface.

C.4 Implementation details

The remainder of this section details 5G-Eval's component stack, scenario schema, automation pipeline, and telemetry format.

Terminology and interfaces. Sections C.2 and C.3 describe the two case studies in general terms; Table 2 gives the corresponding cellular-networking terms. In the scheduler case study, the priority function is `evolved_dl_score` and the bandwidth-sizing function is `evolved_dl_prb_size`; in the ECN-marking case study, the marking-probability function is `evolved_drb_ecn_probability`.

Components. 5G-Eval combines an OpenAirInterface (OAI) 5G NR RAN (Kaltenberger et al., 2020), an Open5GS core (Open5GS Project, 2026), and a FlexRIC near-real-time RIC (Schmidt et al., 2021). The gNB and up to six OAI UE softmodems run in band n78 with 30 kHz subcarrier spacing and 273 physical resource blocks (PRBs). Instead of using radio hardware, the gNB and UEs exchange IQ samples through a shared-memory RF emulator. This setup keeps the full PHY/MAC stack in the loop and eliminates uncontrolled over-the-air effects. Open5GS runs its

Table 2: Mapping from the general terms used in this appendix to the underlying 5G New Radio terminology.

General term	5G New Radio term
base station	gNB
user device	user equipment (UE)
scheduling interval	transmission time interval (TTI)
bandwidth unit / resource block	physical resource block (PRB)
channel-quality indicator	channel quality indicator (CQI)
queueing delay	head-of-line (HOL) delay
resource partition	network slice
per-user-device data queue	data radio bearer (DRB)
packet	service data unit (SDU)
bulk-data traffic	enhanced mobile broadband (eMBB)
latency-critical traffic	ultra-reliable low-latency communication (URLLC)
sparse machine-type traffic	massive machine-type communication (mMTC)
radio-access controller	near-real-time RAN Intelligent Controller (RIC)

per-slice control-plane and user-plane functions on Kubernetes. FlexRIC xApps (controller applications) collect MAC, RLC, PDCP, and GTP telemetry (link, packet-data, and tunneling counters) and apply live slice controls, including PRB caps and scheduler selection, through the O-RAN E2 interface (the near-real-time control interface). A remote server generates traffic by replaying packet traces (Groen et al., 2024) for the eMBB, URLLC, and mMTC classes and by running constant-bitrate UDP and SCReAM RTP flows (Johansson & Sarker, 2017) with ECN/L4S support (Briscoe et al., 2023).

Hardware. Because no radio hardware is involved, the full platform (gNB, UEs, RF emulator, core, and RIC) runs self-contained on a single commodity server. Our experiments use a dual-socket Intel Xeon Silver 4314 machine (32 cores, 64 hardware threads, 2.40 GHz) running Ubuntu 22.04 LTS. The gNB and UE softmodems are the dominant CPU consumers because the PHY layer executes in real time; the remaining components (core, RIC, traffic endpoints) are lightweight.

Scenario specification. A declarative JSON scenario file specifies each experiment. Its parameters include carrier bandwidth, user-device count, per-user-device traffic type, per-user-device mobility or channel profile, slice minimum and maximum PRB ratios, scheduler type, per-slice latency target, duration, and random seed. Smoke tests, training-bank evaluation, and evaluation on the held-out test bank consume this file unchanged, eliminating separate hand-written test rigs.

Scenario names. The text and figures label scenarios with case-tagged integers. Scheduler labels $\mathrm{S}_1^{\mathrm{sch}}$–$\mathrm{S}_8^{\mathrm{sch}}$ denote the illustrative scenarios from Figure 1. Later scheduler labels follow the order in which the scenarios were generated, and ECN-marking labels follow lexicographic configuration-file order; unused labels are omitted from the tables below.

Scheduler parameter strings have the form `class-Nu-chan-rC`. The traffic class is `embb`, `urllc`, `mmtc`, or `cbr` (constant-bitrate) and sets the latency target (300, 50, 1000, and 200 ms, respectively). N is the user-device count. Each letter of `chan` is one user device's channel profile: `L` static line-of-sight, `N` static non-line-of-sight, `C` car, `D` drone, `R` robot, and `S` rotational. C is the slice PRB cap as a percentage of capacity. For example, `embb-4u-CCCD-r25` is an eMBB scenario with four user devices (three car, one drone) and a 25% PRB cap.

Tables 3–4 give the mapping to scenario parameters. The pool tables list the 39 scheduler and 37 ECN-marking scenarios.

Automation and reproducibility. 5G-EVAL automates each scenario from startup through post-processing and records the inputs required for exact replay. A single scenario invocation checks core health, stages the gNB configuration, launches the gNB, RIC, UEs, monitoring xApps, packet captures, traffic senders, and traffic receivers, runs the timed experiment, tears down all processes, and post-processes the logs. The scenario seed deterministically selects both packet traces and channel

Table 3: Scheduler labels and descriptive parameters for the eight illustrative scenarios of Figure 1 (S_1^{sch}–S_8^{sch}) and the 39-scenario pool. Unused labels are omitted. Each parameter string gives the traffic class, user-device count, per-user-device channel profiles, and slice PRB cap.

Label	Parameters	Label	Parameters	Label	Parameters
S_1^{sch}	`embb-2u-LL-r100`	S_{18}^{sch}	`embb-5u-LNNND-r75`	S_{41}^{sch}	`cbr-4u-LDRS-r75`
S_2^{sch}	`embb-4u-LLNN-r100`	S_{20}^{sch}	`urllc-4u-NCDD-r25`	S_{42}^{sch}	`cbr-5u-LLDDS-r50`
S_3^{sch}	`urllc-2u-LL-r100`	S_{22}^{sch}	`cbr-3u-RSS-r50`	S_{43}^{sch}	`cbr-5u-LNSSS-r100`
S_4^{sch}	`urllc-4u-LLNN-r100`	S_{25}^{sch}	`embb-5u-LNCCD-r100`	S_{44}^{sch}	`urllc-3u-NCR-r100`
S_5^{sch}	`mmtc-6u-LLLLNN-r100`	S_{27}^{sch}	`mmtc-6u-CRRSSS-r25`	S_{45}^{sch}	`urllc-4u-NNCR-r25`
S_6^{sch}	`embb-5u-CCDRS-r100`	S_{28}^{sch}	`embb-4u-CCCD-r25`	S_{46}^{sch}	`urllc-4u-NCDD-r100`
S_7^{sch}	`urllc-6u-NCCDRS-r100`	S_{29}^{sch}	`mmtc-6u-LNCRRR-r100`	S_{47}^{sch}	`urllc-4u-LNCD-r50`
S_8^{sch}	`cbr-6u-NCDRRS-r100`	S_{30}^{sch}	`urllc-3u-LDS-r50`	S_{48}^{sch}	`cbr-5u-NDDRS-r25`
S_9^{sch}	`mmtc-3u-CDR-r100`	S_{32}^{sch}	`urllc-4u-CDSS-r75`	S_{49}^{sch}	`cbr-4u-LCRR-r25`
S_{10}^{sch}	`mmtc-6u-LNNDRS-r100`	S_{33}^{sch}	`urllc-5u-LNCRS-r100`	S_{52}^{sch}	`embb-5u-LLCCS-r50`
S_{12}^{sch}	`embb-4u-NNCS-r75`	S_{34}^{sch}	`cbr-5u-LNNCD-r75`	S_{53}^{sch}	`embb-4u-LLSS-r100`
S_{13}^{sch}	`embb-5u-LLNCR-r25`	S_{36}^{sch}	`embb-5u-LCCDD-r25`	S_{54}^{sch}	`cbr-5u-LLCDR-r25`
S_{14}^{sch}	`embb-4u-NCDS-r100`	S_{37}^{sch}	`cbr-5u-LNCRS-r100`	S_{55}^{sch}	`embb-3u-DDR-r50`
S_{15}^{sch}	`urllc-5u-NNCDR-r25`	S_{38}^{sch}	`cbr-5u-LNRSS-r25`	S_{56}^{sch}	`mmtc-3u-NNC-r100`
S_{16}^{sch}	`embb-3u-DDR-r75`	S_{39}^{sch}	`embb-4u-LLRS-r75`	S_{57}^{sch}	`mmtc-5u-LLNRS-r100`
S_{17}^{sch}	`embb-4u-LNDR-r50`	S_{40}^{sch}	`mmtc-6u-NDRRSS-r100`		

Table 4: ECN-marking labels and descriptive parameters for the 37-scenario pool. Unused labels are omitted. Each entry gives the channel profile, latency target, and resource cap.

Label	Parameters	Label	Parameters	Label	Parameters
S_1^{ecn}	car, 60 ms, 5%	S_{16}^{ecn}	rotational, 100 ms, 10%	S_{31}^{ecn}	drone, 100 ms, 5%
S_2^{ecn}	car, 60 ms, 10%	S_{17}^{ecn}	rotational, 60 ms, 5%	S_{32}^{ecn}	drone, 100 ms, 10%
S_3^{ecn}	drone, 60 ms, 5%	S_{18}^{ecn}	rotational, 60 ms, 10%	S_{33}^{ecn}	drone, 20 ms, 5%
S_4^{ecn}	drone, 60 ms, 10%	S_{20}^{ecn}	line-of-sight, 20 ms, 10%	S_{34}^{ecn}	drone, 20 ms, 10%
S_5^{ecn}	random, 100 ms, 5%	S_{21}^{ecn}	line-of-sight, 60 ms, 5%	S_{35}^{ecn}	robot, 60 ms, 5%
S_6^{ecn}	random, 100 ms, 10%	S_{22}^{ecn}	line-of-sight, 60 ms, 10%	S_{36}^{ecn}	robot, 60 ms, 10%
S_9^{ecn}	random, 60 ms, 5%	S_{23}^{ecn}	non-line-of-sight, 100 ms, 5%	S_{37}^{ecn}	rotational, 20 ms, 5%
S_{10}^{ecn}	random, 60 ms, 10%	S_{24}^{ecn}	non-line-of-sight, 100 ms, 10%	S_{38}^{ecn}	rotational, 20 ms, 10%
S_{11}^{ecn}	robot, 100 ms, 5%	S_{25}^{ecn}	non-line-of-sight, 20 ms, 5%	S_{39}^{ecn}	line-of-sight, 100 ms, 5%
S_{12}^{ecn}	robot, 100 ms, 10%	S_{26}^{ecn}	non-line-of-sight, 20 ms, 10%	S_{40}^{ecn}	line-of-sight, 100 ms, 10%
S_{13}^{ecn}	robot, 20 ms, 5%	S_{27}^{ecn}	car, 100 ms, 5%	S_{41}^{ecn}	non-line-of-sight, 60 ms, 5%
S_{14}^{ecn}	robot, 20 ms, 10%	S_{28}^{ecn}	car, 100 ms, 10%	S_{42}^{ecn}	non-line-of-sight, 60 ms, 10%
S_{15}^{ecn}	rotational, 100 ms, 5%				

traces. The platform stores the resolved traces and final configuration snapshot with the experiment artifacts. It replays per-UE CQI traces (Ko et al., 2024) representing static line-of-sight, static non-line-of-sight, car, robot, drone, rotational, and random channel profiles. Consequently, two candidate programs see the same traffic arrivals and the same time-varying channel when evaluated on the same scenario.

Failure recovery. Between runs, 5G-EVAL resets OAI processes, FlexRIC processes, remote traffic generators, shared-memory segments, packet captures, and UE routing state; repeated smoke-test failures trigger a hard environment reset. This isolation is important because the discovery loop evaluates many untrusted candidate programs, some of which crash, starve user devices, exceed timing budgets, or leave the radio stack in an inconsistent state.

Telemetry. Every run produces a uniform telemetry bundle. Per-UE packet captures yield one-way latency and throughput traces; gNB logs yield a per-TTI scheduler trace with HOL delay, RLC buffer state, CQI, ranking, PRB allocation, and slot utilization; ECN logs yield per-DRB marking probabilities; and the platform exports RIC telemetry from its SQLite database. The same post-processing code computes the scenario score, checks guardrails, and builds per-scenario feedback tables for the LLM. The evaluator compiles the evolved functions into the OAI gNB with wrappers that clamp invalid outputs, count violations, and measure per-call execution time.

D PROBE PROGRAMS

This appendix describes the $P = 15$ probe programs that define the fingerprints in each case study. Each set combines three kinds of programs: the discovery seed (`ltti.c` or `lin_ecn.c`), classical probe programs that make design choices different from the seed's, and LLM-generated probe programs. Earlier LLM-based searches from a classical seed produced the LLM-generated probes; all probes were fixed before any reported discovery run, so they are disjoint from the candidate programs those runs produce. The classical probes anchor distinct design points, and the LLM-generated probes fill in between them with combinations the classical ones do not make. Together they span a range of behaviors, which makes the fingerprints informative. Each LLM-generated probe is labeled with its LLM family: Gemini, Gemma, GPT, or Qwen.

D.1 SCHEDULING PROBES

The scheduler probes are the seed LTTI (Elkael et al., 2025), proportional fairness (PF) (Jalali et al., 2000), and 13 LLM-generated probes. PF ranks user devices by the ratio of the transmission size they could currently achieve to their average throughput, so it favors devices on a good channel relative to their own history and ignores queueing delay. LTTI (Listing 1) multiplies that ratio by an exponential urgency term. Listing 3 shows the evolve block of PF. Both classical probes offer the first-ranked user device the full remaining contiguous span of bandwidth.

Listing 3: Evolve block of the PF probe program. The priority is the ratio of the achievable transmission size to the average throughput, and bandwidth sizing returns the full remaining contiguous span. The scaffolding above the block, omitted here with the file's header comment, defines `EVOLVED_EXP_OVERFLOW` as `1e10f`, which this block does not use.

```
/* EVOLVE-BLOCK-START */
float evolved_dl_score(const evolved_score_ctx_t *ctx,
                       const evolved_score_input_t *input)
{
    const float R_inst = (float)input->tbs;
    const float pf_ratio = R_inst / input->R_avg;

    return (float) pf_ratio;
}

uint16_t evolved_dl_prb_size(const evolved_score_ctx_t *ctx,
                             const evolved_prb_size_input_t *input)
{
    (void)ctx;
    return input->max_contiguous_prbs;
}
/* EVOLVE-BLOCK-END */
```

The 13 LLM-generated probes fall into two groups: four rewrites of PF and nine rewrites of LTTI. The PF-derived probes all add sensitivity to queueing delay, which PF lacks, each through a different urgency curve. The LTTI-derived probes mostly keep LTTI's exponential urgency, and two keep it unchanged, so their main departure is in bandwidth sizing. We write d for a user device's queueing delay (the time since its oldest pending packet arrived) and $u = d/\Delta_\theta$ for the delay ratio. Fair share is the remaining bandwidth budget divided among the competing user devices, and estimated need is the number of bandwidth units needed to clear the queue at the current channel quality. All 13 size each grant from some mix of three quantities: fair share, estimated need, and a factor that grows with how close the queueing delay is to the latency target. Only one still returns the full span outright, and only near the latency target. The probes differ in how strongly they weight each factor, which spreads them along the latency-versus-bandwidth trade-off that the scheduler score rewards. Table 5 summarizes each probe.

Table 5: Scheduler probe programs. Each row gives the logic of the priority function and of the bandwidth-sizing function.

Probe	Family	Priority function	Bandwidth sizing
Classical			

Probe	Family	Priority function	Bandwidth sizing
PF		Rate ratio only; ignores queueing delay.	Full remaining contiguous span.
LTTI (seed)		Rate ratio times $\exp(\Delta_\theta/(\Delta_\theta - d))$.	Full remaining contiguous span.
LLM-generated, from PF			
PF with exponential urgency	Gemini	PF times $\exp(6u)$ (u capped at 4) and up to $2\times$ for short queues.	Estimated need plus a 12.5% margin. With competitors, half a fair share at low u, rising stepwise toward the full span above $u = 0.8$, never above the need.
PF that recovers LTTI	Gemma	LTTI's exponential urgency with the exponent capped at 20.	Queued bytes over average throughput, shrunk by competition, raised on poor channels, $0.7\times$ when $u < 0.3$ and growing with u above that.
PF with polynomial urgency	GPT	PF times a quadratic in u (no exponential), with small channel-quality and queue-size factors.	Fair share times a factor that rises from 0.3 to about 2.5 with u.
PF with steep urgency	Qwen	PF times $\exp(2\Delta_\theta/(\Delta_\theta - d))$ and $1 + \text{bytes}/10^4$; urgency falls back to 1 once the target is exceeded.	Fair share, enlarged when $u > 0.7$, then trimmed to the estimated need unless $u > 0.8$, where it takes the midpoint.
LLM-generated, from LTTI			
LTTI with delay numerator	Gemini	Exponent $d/(\Delta_\theta - d)$, so urgency is 1 at zero delay and PF dominates while user devices are far from their target.	A fraction $1.5u$ of the span, divided by competition and scaled by channel quality, with a 2% floor.
LTTI with late quadratic boost	Gemma	LTTI times $1 + 3u^2$ once $u > 0.6$.	Queue demand at a channel-dependent rate. Adds units in proportion to u above 0.7; scaled down when more than four user devices compete and $u \leq 0.7$.
LTTI with multiplicative adjustments	GPT	LTTI (exponent capped) times small factors for channel quality, high u, long queues, and poor channels under delay.	Remaining budget over competitors, scaled stepwise by u from $0.4\times$ up to the whole budget above $u = 0.78$, then adjusted for queue length and channel quality.
Unchanged LTTI, need-based sizing	Qwen	LTTI unchanged.	Queued bytes over the achievable transmission size, times $15/\text{CQI}$ and $1 + wu$ ($w = 6$ for latency targets of 100 ms or less, 2 otherwise); divided by competition unless the target is 100 ms or less and $u > 0.2$.
LTTI with channel floor and violation ranking	Gemini	Rate ratio floored at 0.05 and raised to 0.5 when $u > 0.75$. Once the target is exceeded, channel quality is ignored and user devices are ranked by the violation.	Fair share times $0.25 + 4u^3$ (u capped at 1.5), trimmed to the estimated need while $u < 0.85$.
Unchanged LTTI, late full span	Gemma	LTTI unchanged.	Full span once $u \geq 0.8$; otherwise the smaller of the estimated need and the fair share.
LTTI with mild multipliers	GPT	LTTI (exponent capped) times small factors for u, channel quality (a bonus on poor channels), and queue length.	Fair share times $0.30 + 0.58u$, adjusted for channel quality and queue length, with floors of half the budget above $u = 0.94$ and a third on poor channels above $u = 0.78$.

Probe	Family	Priority function	Bandwidth sizing
LTTI favoring long queues on good channels	Qwen	LTTI times $(1 + \text{bytes}/2000)(1 + \text{CQI}/10)$.	Estimated need, doubled above $u = 0.7$ and tripled above $u = 0.9$, cut 20% with more than three competitors.
LTTI with capped rate ratio	Qwen	LTTI with the rate ratio capped at 100.	Fair share times $0.5 + 0.5u$; never exceeds the fair share.

D.2 ECN MARKING PROBES

The ECN-marking probes are the seed, four classical markers, and ten LLM-generated markers. The seed (`lin_ecn.c`, Listing 2) ramps the marking probability linearly with queued bytes, between 10% and 50% of the bytes that the flow's bandwidth would drain within the latency target Δ_θ at a nominal 10 Mbps. It reads queue occupancy alone. Each classical marker isolates one other design choice. DCTCP-step (Listing 4) uses the seed's signal but replaces the ramp with a step (Alizadeh et al., 2010). PIE-like (Listing 5) steers delay toward the middle of the latency budget, where the seed applies no pressure (Pan et al., 2017). Sojourn-only (Listing 6) reacts to delay rather than bytes (Nichols & Jacobson, 2012; Nichols et al., 2018). Channel-throttle (Listing 7) reacts to channel quality alone.

Listing 4: Evolve block of the DCTCP-step probe program. It marks every packet when the queued bytes exceed a single threshold, 30% of the bytes drained within the latency target at the nominal rate, and marks nothing below it. The scaffolding above the block, omitted here with the file's header comment, defines `NOMINAL_RATE_KBPS` as 10,000 kbps, `K_FRAC_PCT` as 30, and `FALLBACK_K_BYTES` as 8,000.

```
/* EVOLVE-BLOCK-START */
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    uint32_t k_bytes;
    if (ctx->target_delay_ms == 0u) {
        k_bytes = FALLBACK_K_BYTES;
    } else {
        const uint64_t k = (uint64_t)NOMINAL_RATE_KBPS * ctx->target_delay_ms;
        k_bytes = (uint32_t)((k * K_FRAC_PCT) / 800u);
    }
    return (input->bytes_in_buffer > k_bytes) ? 10000 : 0;
}
/* EVOLVE-BLOCK-END */
```

Listing 5: Evolve block of the PIE-like probe program. It is a stateless proportional marker: it averages a delay term (the larger of the queueing delay and the average sojourn time, measured above a setpoint at half the latency target) with a queue term (the queued bytes, measured in scheduling intervals of channel-implied drain and saturating at four). It never marks an empty queue. The scaffolding above the block, omitted here with the file's header comment, defines `FALLBACK_TARGET_US` as 60,000.

```
/* EVOLVE-BLOCK-START */
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    if (input->bytes_in_buffer == 0u) {
        return 0;
    }

    uint32_t target_us = ctx->target_delay_ms ?
                         ctx->target_delay_ms * 1000u :
                         FALLBACK_TARGET_US;
    uint32_t setpoint_us = target_us / 2u;

    uint32_t delay_us = input->hol_delay_us;
    if (input->avg_sojourn_us > delay_us) {
        delay_us = input->avg_sojourn_us;
    }
```

```
    uint32_t p_term = 0u;
    if (delay_us > setpoint_us) {
        uint64_t err = (uint64_t)(delay_us - setpoint_us) * 10000u;
        p_term = (uint32_t)(err / target_us);
        if (p_term > 10000u) p_term = 10000u;
    }

    /* Drain capacity in bytes over one 0.5 ms TTI at the CQI-implied rate.
     * rate_kbps -> bytes_per_tti = rate_kbps * 1000 / 8 / 2000 = rate_kbps / 16. */
    uint8_t cqi = input->wb_cqi;
    if (cqi > 15u) cqi = 15u;
    uint32_t rate_kbps = 500u + (uint32_t)cqi * 600u;
    uint32_t drain_per_tti = rate_kbps / 16u;
    if (drain_per_tti == 0u) drain_per_tti = 1u;

    /* d_term in [0, 10000]: queue depth measured in TTIs of drain.
     * Saturate at 4 TTIs so we don't dominate the proportional term. */
    uint32_t ttis_queued = input->bytes_in_buffer / drain_per_tti;
    uint32_t d_term = (ttis_queued >= 4u) ? 10000u : (ttis_queued * 2500u);

    uint32_t prob = (p_term + d_term) / 2u;
    if (prob > 10000u) prob = 10000u;
    return (uint16_t)prob;
}
/* EVOLVE-BLOCK-END */
```

Listing 6: Evolve block of the Sojourn-only probe program. The marking probability ramps linearly with the larger of the queueing delay and the average sojourn time, between 30% and 80% of the latency target. Queued bytes only rule out an empty queue. The scaffolding above the block, omitted here with the file's header comment, defines `LO_FRAC_PCT` as 30, `HI_FRAC_PCT` as 80, and `FALLBACK_TARGET_US` as 60,000.

```
/* EVOLVE-BLOCK-START */
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    uint32_t target_us = ctx->target_delay_ms ?
                         ctx->target_delay_ms * 1000u :
                         FALLBACK_TARGET_US;
    uint32_t delay_us = input->hol_delay_us;
    if (input->avg_sojourn_us > delay_us) {
        delay_us = input->avg_sojourn_us;
    }
    if (input->bytes_in_buffer == 0u) {
        return 0;
    }

    uint32_t lo_us = (target_us * LO_FRAC_PCT) / 100u;
    uint32_t hi_us = (target_us * HI_FRAC_PCT) / 100u;
    if (hi_us <= lo_us) hi_us = lo_us + 1u;

    if (delay_us <= lo_us) return 0;
    if (delay_us >= hi_us) return 10000;
    return (uint16_t)(((uint64_t)(delay_us - lo_us) * 10000u) / (hi_us - lo_us));
}
/* EVOLVE-BLOCK-END */
```

Listing 7: Evolve block of the Channel-throttle probe program. It marks only when the channel is poor and the queue is nonempty: never at CQI 11 or above, with probability 6% to 30% for CQI 6 to 10, and 50% to 100% for CQI 5 or below. The scaffolding above the block, omitted here with the file's header comment, defines no constants.

```
/* EVOLVE-BLOCK-START */
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    (void)ctx;
    if (input->bytes_in_buffer == 0u) {
        return 0;
    }
    uint8_t cqi = input->wb_cqi;
    if (cqi > 15u) cqi = 15u;

    if (cqi >= 11u) {
```

```
        return 0;
    }
    if (cqi >= 6u) {
        return (uint16_t)((11u - cqi) * 600u);  /* 600..3000 (= 6%..30%) */
    }
    /* cqi in [0..5]: 5000..10000 (= 50%..100%) */
    uint32_t p = 5000u + (uint32_t)(5u - cqi) * 1000u;
    if (p > 10000u) p = 10000u;
    return (uint16_t)p;
}
/* EVOLVE-BLOCK-END */
```

The ten LLM-generated markers all add queueing delay and channel quality to the seed's byte ramp, inputs the seed ignores. They disagree on three choices. The first is whether queueing delay or queued bytes is the primary signal. The second is whether a poor channel should trigger marking earlier, because a slowly draining queue builds delay from few bytes, or later, because marking on a slow channel sacrifices throughput. The third is whether to use the bytes served in the previous interval to detect whether the queue is draining. These disagreements separate their fingerprints. Table 6 summarizes each marker. A *byte ramp* is a linear ramp of marking probability in queued bytes, and a *delay ramp* is one in queueing delay d, with thresholds given as fractions of the latency target Δ_θ. Probabilities are in percent, and "points" means percentage points of marking probability.

Table 6: ECN-marking probe programs. Each row gives the logic of the marking function.

Probe	Family	Marking logic
Seed and classical		
Linear byte ramp (seed)		Byte ramp between 10% and 50% of the byte budget; reads no other input.
DCTCP-step		Marks every packet above one byte threshold (30% of the budget) and none below it.
PIE-like		Average of a delay term and a queue-drain term, aimed at the middle of the latency budget; never marks an empty queue.
Sojourn-only		Delay ramp on the larger of d and the average sojourn time, from 30% to 80% of Δ_θ; bytes only rule out an empty queue.
Channel-throttle		Marks only on a poor channel with a nonempty queue: 6–30% for CQI 6–10, 50–100% for CQI ≤ 5, never at CQI ≥ 11.
LLM-generated		
Predicted-delay ramp	Gemini	No byte ramp. Converts queued bytes to a predicted drain time at a channel-derived rate, then ramps from 30% to 70% of Δ_θ on the largest of that prediction, d, and the average sojourn time.
Channel-scaled dual ramp	Gemini	Larger of a byte ramp and a delay ramp; both start later as channel quality improves (the start moves from 20% to 65% of the budget). Marking is canceled when the queue fits in two of the previous interval's transmissions and quartered when it fits in four.
Half-probability byte ramp	Gemma	The seed's byte ramp with probability capped at 50%. A delay ramp scaled by channel quality is present, but its thresholds lie tens of times Δ_θ out, so it is effectively inactive. Never marks when no latency target is set.
Measured-throughput thresholds with fade detection	Gemma	Larger of a delay ramp (30–80% of Δ_θ, upper end 60% when CQI < 6) and a byte ramp whose thresholds use the user device's measured average throughput instead of the nominal rate and tighten on poor channels. Adds 20 points when the previous interval served less than half the average throughput (a fade), and 10 more at CQI < 4 when the probability is below 50%.
Latency-target and channel lookup	GPT	Larger of a byte ramp and a delay ramp, with both sets of thresholds taken from a small table indexed by latency target (at most 25 ms or not) and channel-quality band. A tight target on a poor channel starts marking at about 1% of the byte budget; a loose target on a good channel relaxes to 18–85%. Never marks an empty queue.

Probe	Family	Marking logic
Lookup with slow-drain floors	GPT	A similar channel-by-target table for both ramps, with three additions: an 80% floor for a tight target on a very poor channel with more than one packet queued; floors of 20–95% when the previous interval served less than an eighth of the queue; and marking quartered on a good channel when one previous-interval transmission would clear the queue and $d < 0.35\Delta_\theta$.
Seed ramp with poor-channel override	GPT	Identical to the seed for CQI > 4 and latency targets above 25 ms, so it is the probe closest to the seed. On poor channels it starts the byte ramp earlier, adds a steep delay ramp, and floors the probability at 19%. For tight targets on other channels it adds a delay ramp capped at 32%.
Tightened bytes with stepped delay bonus	Qwen	The seed's byte ramp with thresholds shrunk to 35% (CQI < 5) or 55% (CQI 5–8), plus 15, 30, or 60 points as d passes 30%, 50%, and 70% of Δ_θ. The bonus is additive.
Delay ramp that recedes on poor channels	Qwen	On good channels it ignores queued bytes and uses a delay ramp from 40% to 70% of Δ_θ at CQI 15. The ramp shifts later as channel quality falls, so for CQI ≤ 11 it starts at or beyond Δ_θ. For CQI ≤ 7 a separate byte ramp at 5–20% of the budget takes over.
Relaxed bytes with delay ramp and stall boost	Qwen	The seed's byte ramp with thresholds widened by 25% (CQI 8–11) or 50% (CQI ≥ 12), the reverse of the probes that tighten on poor channels. Takes the larger of that and a delay ramp from 40% to 100% of Δ_θ, and multiplies by 1.5 when the previous interval served less than a quarter of the lower byte threshold.

E Discovered Programs

This appendix provides the six top-ranked FARE candidate programs behind Table 1, one from each model on each pool. Listings 8 and 9 print the two Gemini3.1 programs; the text describes the Gemma4 and Qwen3.5 programs. These candidate programs come from the reported discovery runs, so they are disjoint from the probe programs of Appendix D.

E.1 Scheduling

The seed program scores a user device as the product of its service weight, its instantaneous rate over its average rate, and an exponential in its queueing delay. Its bandwidth function returns the full contiguous span on every call.

Each FARE candidate program withholds part of that span while head-of-line delay is low and returns the full span as delay nears the latency target. Meeting the latency target earns half the score; the other half rewards a smaller bandwidth footprint, whether or not the target is met. The seed always returns the full span, and so earns little of that half.

Gemini3.1 keeps the seed priority, with a floor of 0.1 on average rate, and caps the exponential once its argument exceeds 23. While at least 15% of the latency budget and at least 4 ms remain, it sizes the grant to empty the buffer in 60% of the time left, at $5 \cdot \mathrm{CQI} + 4$ bytes per resource block per millisecond, and it grants at least five blocks. Otherwise, it returns the full span.

Listing 8: Evolve block of the top-ranked candidate program from the Gemini3.1 FARE scheduler run. The priority is the LTTI seed's, with a floor on average rate. The bandwidth function sizes the grant to drain the buffer in 60% of the remaining time and grants the full span only near the latency target.

```
float evolved_dl_score(const evolved_score_ctx_t *ctx,
                       const evolved_score_input_t *input)
{
    float r_avg = input->R_avg < 0.1f ? 0.1f : input->R_avg;
    float pf_ratio = (float)input->tbs / r_avg;

    double denom = ctx->delta_s - (input->hol_delay_us / 1e6);
    float delay_mult = EVOLVED_EXP_OVERFLOW;
```

```
    if (denom > 1e-9) {
        double exp_arg = ctx->delta_s / denom;
        if (exp_arg <= 23.0) {
            delay_mult = (float)exp(exp_arg);
        }
    }

    return (float)ctx->qos_weight * pf_ratio * delay_mult;
}

uint16_t evolved_dl_prb_size(const evolved_score_ctx_t *ctx,
                             const evolved_prb_size_input_t *input)
{
    float time_left_ms = ctx->delta_ms - (input->hol_delay_us / 1000.0f);
    uint16_t max_prb = input->max_contiguous_prbs;

    if (time_left_ms < ctx->delta_ms * 0.15f || time_left_ms < 4.0f) {
        return max_prb;
    }

    float cqi = input->wb_cqi > 0 ? (float)input->wb_cqi : 1.0f;
    float bytes_per_prb = cqi * 5.0f + 4.0f;

    float target_time_ms = time_left_ms * 0.6f;
    if (target_time_ms < 1.0f) target_time_ms = 1.0f;

    uint16_t desired_prbs = (uint16_t)(input->bytes_in_buffer / (target_time_ms *
     bytes_per_prb)) + 1;

    if (desired_prbs < 5) desired_prbs = 5;
    if (desired_prbs > max_prb) desired_prbs = max_prb;

    return desired_prbs;
}
```

With $u = d/\Delta_\theta$ as in Appendix D.1, Gemma4 multiplies the seed priority by $1 + 5u$ and sets the grant to the span times $0.2 + 0.8u$, capped by the blocks still available, with a floor of one block. Qwen3.5 multiplies the seed priority by $1 + 5u/(1.1 - u)$, and by a buffer factor near one. Its grant is the larger of a quarter-step share of the span and $\text{bytes}/150$, again capped by the blocks still available, with a floor of one block.

E.2 ECN MARKING

The seed marks from queued bytes alone. The probability rises linearly between 10% and 50% of the bytes a 10 Mbps flow drains within the latency target.

Each FARE candidate program adds a delay ramp and marks sooner on a poor channel. Gemini3.1 and Gemma4 take the larger of head-of-line delay and average sojourn time. Qwen3.5 uses head-of-line delay alone.

In the Gemini3.1 program (Listing 9), channel quality, capped at 15, sets both ends: the ramp starts at $\Delta \cdot (10 + 2 \cdot \text{CQI})/100$ and reaches 100% at $\Delta \cdot (40 + 3 \cdot \text{CQI})/100$, so a poor channel marks sooner. A second ramp, from half to all of $(50 + 100 \cdot \text{CQI}) \cdot \Delta$ bytes, with Δ in milliseconds, can only raise that probability. With no latency target, Gemini3.1 uses the seed's byte ramp.

Listing 9: Evolve block of the top-ranked candidate program from the Gemini3.1 FARE ECN-marking run. Marking follows the larger of queueing delay and sojourn time, and both the delay and byte thresholds move earlier as channel quality falls. The scaffolding above the block, omitted here with the file's header comment, defines `FALLBACK_LO_BYTES` as 8,000 and `FALLBACK_HI_BYTES` as 20,000.

```
uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                     const evolved_drb_ecn_input_t *input)
{
    if (ctx->target_delay_ms == 0u) {
        const uint32_t v = input->bytes_in_buffer;
        if (v <= FALLBACK_LO_BYTES) return 0;
        if (v >= FALLBACK_HI_BYTES) return 10000;
        return (uint16_t)(((uint64_t)(v - FALLBACK_LO_BYTES) * 10000u) / (FALLBACK_HI_BYTES
      - FALLBACK_LO_BYTES));
    }
```

```
    uint32_t cqi = input->wb_cqi;
    if (cqi > 15) cqi = 15;

    uint32_t target_us = ctx->target_delay_ms * 1000u;
    uint32_t delay_us = input->hol_delay_us > input->avg_sojourn_us ?
                        input->hol_delay_us : input->avg_sojourn_us;

    uint32_t lo_time = target_us * (10u + cqi * 2u) / 100u;
    uint32_t hi_time = target_us * (40u + cqi * 3u) / 100u;

    uint32_t prob = 0;
    if (delay_us >= hi_time) {
        prob = 10000;
    } else if (delay_us > lo_time) {
        prob = (uint32_t)(((uint64_t)(delay_us - lo_time) * 10000u) / (hi_time - lo_time));
    }

    uint32_t max_bytes = (50u + cqi * 100u) * ctx->target_delay_ms;
    uint32_t lo_bytes = max_bytes / 2u;

    if (input->bytes_in_buffer >= max_bytes) {
        return 10000;
    } else if (input->bytes_in_buffer > lo_bytes) {
        uint32_t p_bytes = (uint32_t)(((uint64_t)(input->bytes_in_buffer - lo_bytes) *
     10000u) / (max_bytes - lo_bytes));
        if (p_bytes > prob) prob = p_bytes;
    }

    return (uint16_t)prob;
}
```

Gemma4 uses the same delay signal. A target of at most 20 ms sets the ramp from 25% to 60% of the target; any longer target sets it from 40% to 80%. Both ends move earlier by 10 percentage points of the target when channel quality is below 7. Below the lower end, Gemma4 marks at 5% when more than 12 packets are queued and channel quality is below 10. Inside the ramp, it adds 20 percentage points when the buffer exceeds three times the last transport block, and another 20 percentage points when queued bytes over average service cross a threshold. The code comments call that threshold 80% of the target, but they assume a 1 ms scheduling interval, so the threshold is a shorter drain time than 80% of the target. With no latency target, Gemma4 returns zero.

Qwen3.5 keeps the seed's byte ramp and takes the larger of that ramp and a delay ramp from 40% to 75% of the target. It multiplies the result by 1.3 when channel quality is at most 3. It halves the probability when all three hold: the last interval served more than half the queue, average service exceeds $5{,}000$ bytes per interval, and channel quality is at least 12.

F Additional Experimental Details and Results

This appendix records the scenario banks, search configuration, auxiliary diagnostics, and complete evaluation protocol for Section 5. It uses the scenario labels defined in Appendix C.4.

F.1 Score variability and the training-bank representation error

To measure score variability, we re-evaluate each seed program twice on its pool.

F.2 Scenario pools and compared banks

The scheduler pool has 39 scenarios and the ECN-marking pool 37; from each, every method builds a training bank of $K = 8$ and $K = 10$ scenarios, respectively. Table 7 lists every training bank used in the five-method comparison.

Order oracle. This method reads the full-pool ranking, so it is a reference rather than a training bank one could build without that ranking. Greedy addition does not maximize the overlap over every K-subset, so the result does not bound what any K-subset can achieve. Let $\bar{s}_S(\pi_i)$ be probe i's uniform mean over a scenario subset S, and let T_S be the three probes with the largest such means; $T_{\mathcal{B}}$ is that set over the full pool. Starting from the empty set, the oracle adds at each step the

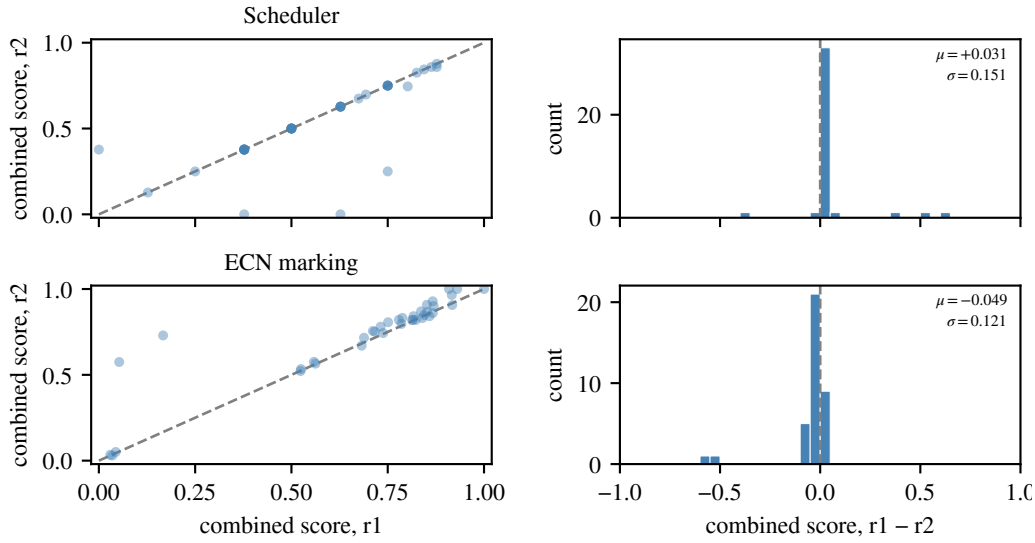

Figure 7: Independent re-evaluations of `ltti.c` (top) and `lin_ecn.c` (bottom) on the full scheduler and ECN-marking scenario pools.

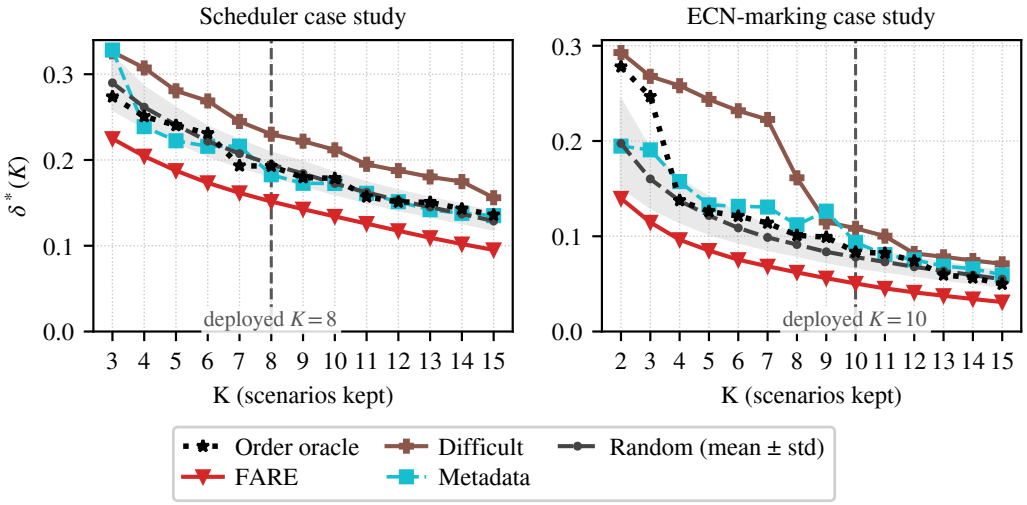

Figure 8: Representation error $\delta^*(K)$, the mean PDD from each pool scenario to its representative, against training-bank size K for the scheduler (left) and ECN-marking (right) pools. Vertical lines mark the chosen K.

scenario that maximizes $|T_{\mathcal{B}} \cap T_S|$, breaking ties by the Spearman correlation between $\bar{s}_S$ and $\bar{s}_{\mathcal{B}}$, and any remaining tie by the earliest scenario in the pool listing.

PDD geometry. Figure 9 shows the pairwise PDD matrix reordered by PAM cluster; the block structure it reveals is what PAM exploits to place medoids. Figure 10 places the same scenarios by classical multidimensional scaling (MDS) of those distances. MDS recovers Euclidean coordinates whose pairwise distances best match D_{PDD}; the plotted axes are the first two such coordinates and have no interpretation as scenario features. The two coordinates capture 48% of the positive eigenvalue mass on the scheduler pool and 83% on the ECN-marking pool. The clusters and medoids in both figures produce the FARE training banks listed in Table 7.

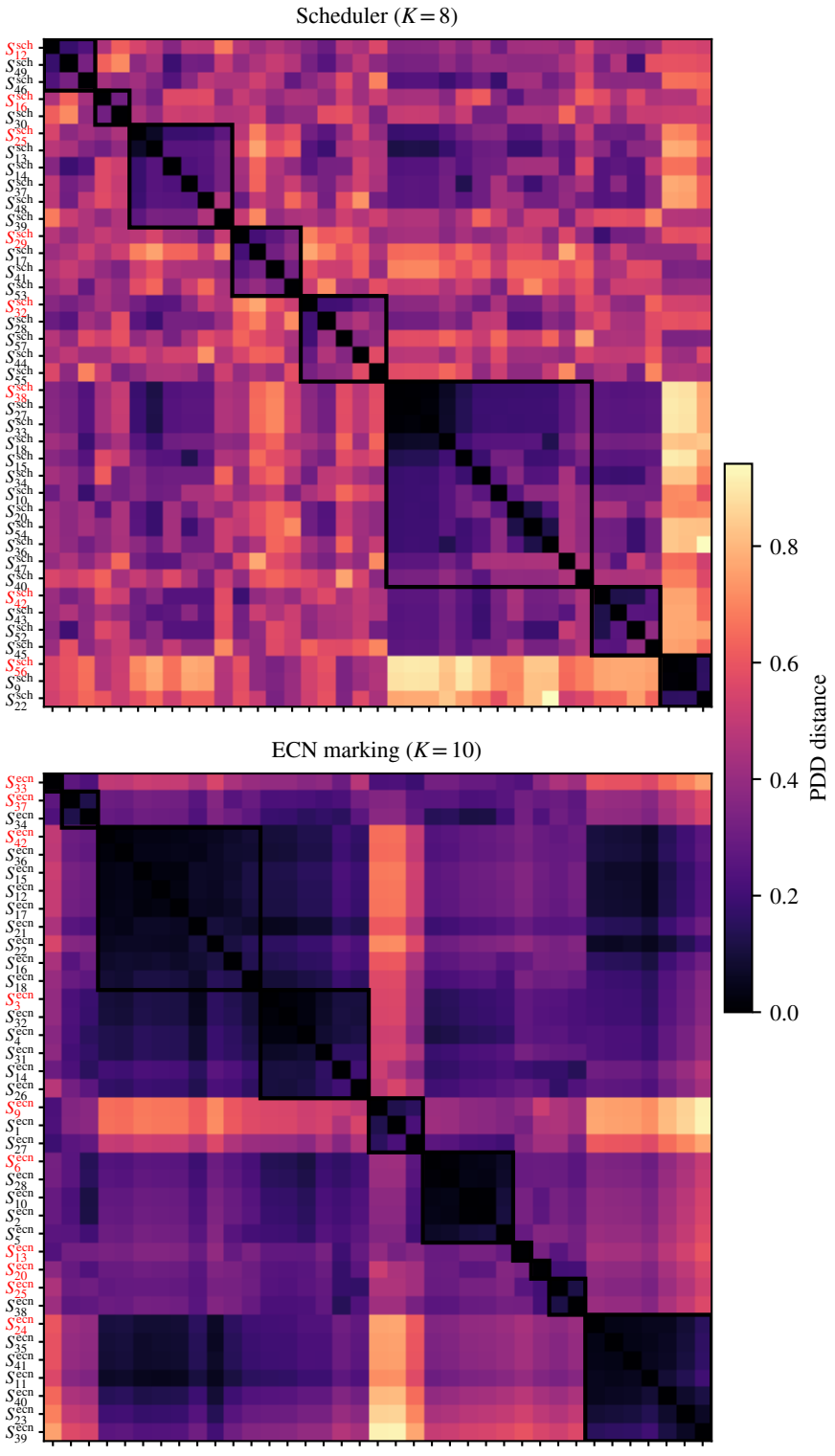

Figure 9: PDD matrices of the scheduler (top) and ECN-marking (bottom) scenario pools, ordered by PAM cluster. The matrices are symmetric, so scenario labels appear on the rows; medoids are in red. Stars in the companion classical-MDS embedding (Figure 10) mark the same medoids.

Table 7: Training-bank scenarios. Scheduler cells contain eight scenarios; ECN-marking cells contain ten.

Method	Scheduler	ECN marking
FARE	$\{S^{sch}_{12}, S^{sch}_{16}, S^{sch}_{25}, S^{sch}_{29}, S^{sch}_{32}, S^{sch}_{38}, S^{sch}_{42}, S^{sch}_{56}\}$	$\{S^{ecn}_{3}, S^{ecn}_{6}, S^{ecn}_{9}, S^{ecn}_{13}, S^{ecn}_{20}, S^{ecn}_{24}, S^{ecn}_{25}, S^{ecn}_{33}, S^{ecn}_{37}, S^{ecn}_{42}\}$
Random	$\{S^{sch}_{9}, S^{sch}_{10}, S^{sch}_{12}, S^{sch}_{20}, S^{sch}_{25}, S^{sch}_{33}, S^{sch}_{38}, S^{sch}_{44}\}$	$\{S^{ecn}_{2}, S^{ecn}_{9}, S^{ecn}_{14}, S^{ecn}_{20}, S^{ecn}_{27}, S^{ecn}_{28}, S^{ecn}_{33}, S^{ecn}_{35}, S^{ecn}_{37}, S^{ecn}_{38}\}$
Metadata	$\{S^{sch}_{9}, S^{sch}_{10}, S^{sch}_{14}, S^{sch}_{36}, S^{sch}_{37}, S^{sch}_{38}, S^{sch}_{47}, S^{sch}_{55}\}$	$\{S^{ecn}_{3}, S^{ecn}_{4}, S^{ecn}_{17}, S^{ecn}_{18}, S^{ecn}_{27}, S^{ecn}_{35}, S^{ecn}_{36}, S^{ecn}_{40}, S^{ecn}_{41}, S^{ecn}_{42}\}$
Difficult	$\{S^{sch}_{15}, S^{sch}_{18}, S^{sch}_{27}, S^{sch}_{33}, S^{sch}_{34}, S^{sch}_{36}, S^{sch}_{38}, S^{sch}_{54}\}$	$\{S^{ecn}_{1}, S^{ecn}_{2}, S^{ecn}_{5}, S^{ecn}_{9}, S^{ecn}_{13}, S^{ecn}_{27}, S^{ecn}_{33}, S^{ecn}_{34}, S^{ecn}_{37}, S^{ecn}_{38}\}$
Order oracle	$\{S^{sch}_{13}, S^{sch}_{14}, S^{sch}_{20}, S^{sch}_{27}, S^{sch}_{33}, S^{sch}_{41}, S^{sch}_{42}, S^{sch}_{52}\}$	$\{S^{ecn}_{1}, S^{ecn}_{5}, S^{ecn}_{16}, S^{ecn}_{20}, S^{ecn}_{23}, S^{ecn}_{24}, S^{ecn}_{25}, S^{ecn}_{34}, S^{ecn}_{35}, S^{ecn}_{37}\}$

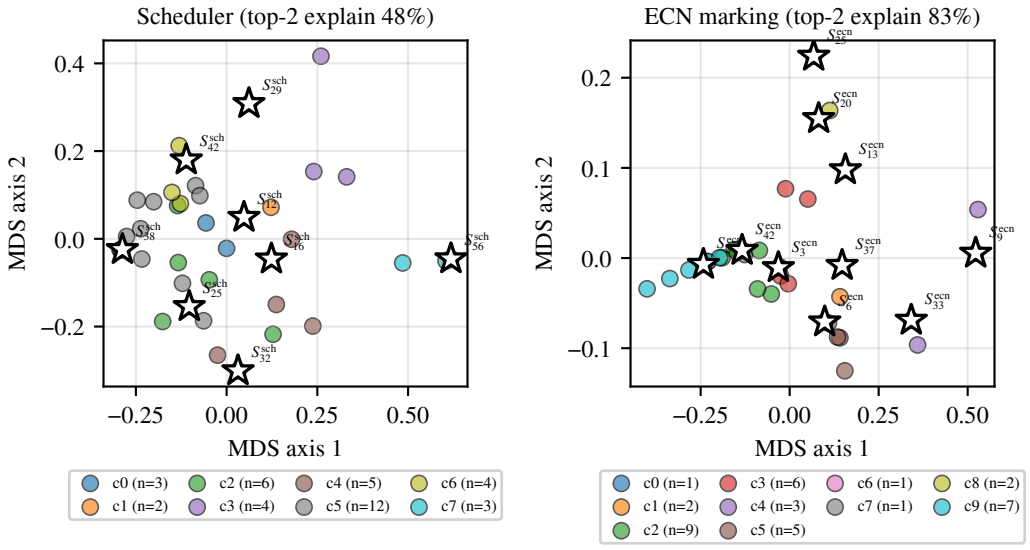

Figure 10: Classical MDS embeddings of the scheduler (left) and ECN-marking (right) scenario pools. Each point is a scenario; Euclidean distance in the plane approximates D_{PDD}. The axes are the leading principal coordinates of that embedding (top two capture 48% and 83% of the positive eigenvalue mass). Stars mark selected medoids.

Drop-one-probe stability. Dropping any one probe program changes the FARE training bank but leaves its representation error nearly unchanged. Table 8 reports a Jaccard jackknife (Hennig, 2007) over the $P = 15$ probe programs: each fold deletes one row of V, rebuilds the fingerprints from the remaining $P-1$ probe programs, and re-runs PAM at the chosen K. Mean Jaccard overlap between the reselected medoids and the chosen training bank of Table 7 is 0.49 on the scheduler pool (range 0.14–0.78) and 0.54 on the ECN-marking pool (range 0.25–0.82). On each fold we then recompute δ^* under the reduced PDD for both banks; $\Delta\delta^*$ is the extra mean PDD to the representative of keeping the chosen training bank instead of the reselected one. That penalty averages 0.002 (at most 0.006) on the scheduler pool, compared with the full-probe $\delta^*(8) = 0.152$, and averages 0.000 (at most 0.001) on the ECN-marking pool, compared with $\delta^*(10) = 0.050$. Distinct medoid sets can therefore realize nearly the same representation error.

Table 8: Leave-one-probe-out stability of the FARE training bank. Jaccard is the overlap of the reselected medoids with the chosen training bank, averaged over the $P = 15$ folds; $\Delta\delta^*$ is the mean extra PDD to the representative incurred by keeping the chosen training bank under that fold's reduced fingerprints.

Case study	Jaccard overlap	$\Delta\delta^*$
Scheduler	0.49	$+0.002$
ECN marking	0.54	$+0.000$

F.3 METADATA AND BEHAVIORAL GEOMETRY

We test whether scenario parameters recover neighborhoods under D_{PDD}. The scheduler comparison uses traffic class (`embb`, `urllc`, `mmtc`, `cbr`), user-device count (3–6), resource cap (25, 50, 75, or 100%), latency target (50, 200, 300, or 1000 ms), the set of mobility profiles (static line-of-sight, static non-line-of-sight, car, drone, robot, and rotational), and the number of distinct mobile-channel types (1–3). The ECN-marking comparison uses channel profile (line-of-sight, non-line-of-sight, car, drone, robot, rotational, or random), resource cap (5 or 10%), and latency target (20, 60, or 100 ms); traffic class (`scream`) and user-device count (1) are constant on that scenario pool.

The two channel-related rows of Figure 11 match how each pool assigns channels. Every ECN-marking scenario has one user device, so *channel profile* is that user device's channel. Every scheduler scenario has several user devices, each with its own channel, and every scenario in the 39-scenario pool includes at least one mobile-channel user device. The scheduler row is therefore *mobile-channel types*, the count of distinct mobile profiles among those user devices (values 1–3).

Gower metadata distance. For scenarios θ_a and θ_b, we use a Gower-style mixed-type dissimilarity (Gower, 1971). Let $\mathcal{F}_{ab}$ be the scenario parameters populated for both scenarios. Their distance is

$$D_{\mathrm{gower}}(\theta_a, \theta_b) = \frac{1}{\max\{1, |\mathcal{F}_{ab}|\}} \sum_{f \in \mathcal{F}_{ab}} d_f(\theta_a, \theta_b). \tag{5}$$

The per-parameter dissimilarity is

$$d_f(\theta_a, \theta_b) = \begin{cases} \dfrac{|x_{af} - x_{bf}|}{\max\{R_f, 10^{-12}\}}, & f \text{ numeric}, \\ \mathbb{1}[x_{af} \neq x_{bf}], & f \text{ categorical}, \\ 1 - \dfrac{|M_{af} \cap M_{bf}|}{|M_{af} \cup M_{bf}|}, & f \text{ set-valued}. \end{cases}$$

The numeric parameters are user-device count, resource cap, and latency target; traffic class is categorical. Gower's original coefficient compares only those two types. The mobility parameter is a set of profiles, which that coefficient does not cover, so we add a third type: Jaccard dissimilarity of the two sets. The mix then counts as a single parameter, equal in weight to user-device count or traffic class, and two overlapping mixes are only partly distant. A parameter missing from either scenario is omitted, two empty mobility sets are omitted, and a pair with no shared parameters is assigned distance zero. For ECN marking, user-device count and traffic class are constant and therefore contribute zero; each mobility set is a singleton, so Jaccard reduces to ordinary $0/1$ mismatch on the channel profile. Its varying parameters are thus channel profile, resource cap, and latency target. Our *Metadata* baseline applies PAM to D_{gower} to construct a simulator-free training bank (Gower distance and k-medoids have also been paired for an analogous test-construction task (Chu et al., 2026)). We then test whether those parameter-defined representatives preserve the program-induced behavioral geometry measured by PDD. Unlike PDD, D_{gower} requires no probe-program executions.

Appendix F.2 (Figures 9 and 10) visualizes the PDD distance matrix and its PAM clusters and medoids for both pools. Figure 11 tests whether individual parameters recover that behavioral geometry. Sharing a parameter value changes mean PDD little relative to the pairwise spread: on the scheduler pool, four of six absolute intra–inter gaps are below 0.03, and only user-device count passes the $p < 0.01$ permutation-test threshold (gap 0.113 against an all-pair mean of 0.434). On the ECN-marking pool, channel profile has the largest gap (0.078, $p < 0.001$), but that gap is likewise small relative to the spread.

Combining the parameters into Gower distance (Figure 12) leads to the same conclusion. Its correlation with PDD is weak in both pools (Mantel $\rho = 0.23$ for scheduling and 0.22 for ECN marking), and Gower-nearest neighbors rarely coincide with PDD-nearest neighbors. Even pairs with identical scenario parameters remain nearly as far apart behaviorally as arbitrary pairs (mean PDD 0.347 versus 0.434 for scheduling and 0.270 versus 0.287 for ECN marking). Thus, the declared scenario parameters do not recover the behavioral geometry that the probe-program fingerprints expose.

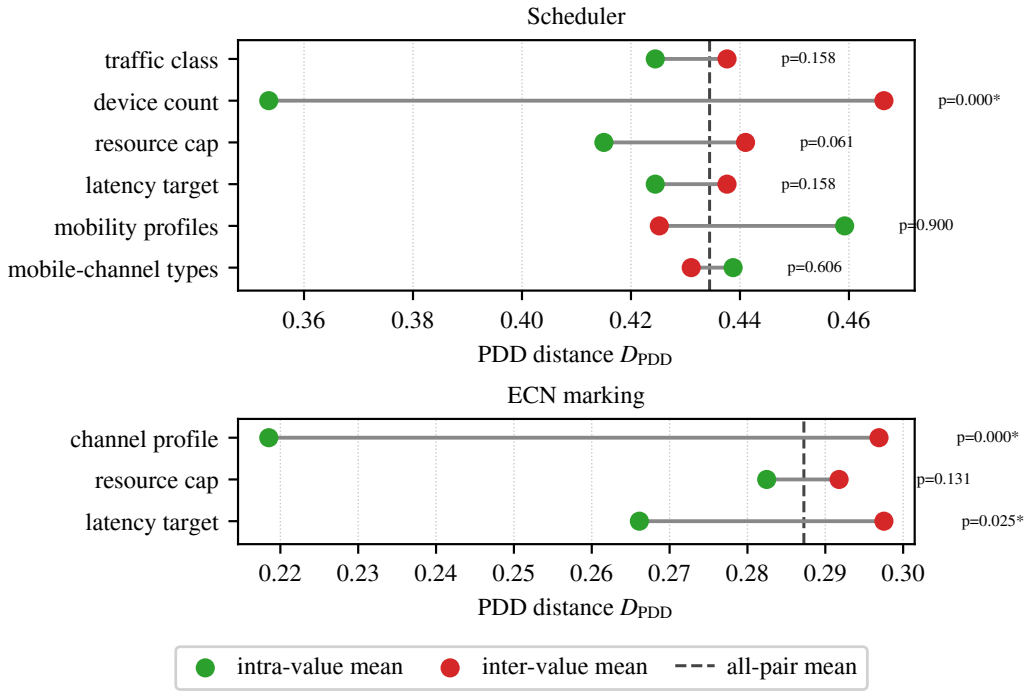

Figure 11: Mean PDD between scenario pairs with the same metadata value (intra) and pairs with different values (inter). Error bars show one standard deviation; dashed lines show each scenario pool's all-pair mean. Scheduler results appear at top and ECN-marking results at bottom. The scheduler's mobile-channel types row counts distinct mobile profiles among a multi-user-device mix; the ECN-marking channel profile row is the single user device's channel.

F.4 Discovery convergence

Figure 13 shows best-so-far training-bank score through iteration 50 on the chosen FARE training banks.

F.5 Pool-wide and held-out evaluations

Figures 14 and 15 resolve the pool-score columns of Table 1 into per-scenario scores.

Section 4.2 reserves each method's unselected scenarios as a search-unseen held-out test bank and treats the difference between held-out and training-bank means as a measure of overfitting to that training bank. Figure 16 reports this within-method gap for each method's top-ranked candidate program.

On the scheduler pool, Qwen3.5 keeps every method's training-bank mean within 0.05 of its own held-out mean. Three of the five gaps are at most 0.02 in absolute value (Random $+0.01$, Metadata $+0.02$, Order oracle -0.01). FARE is $+0.05$ (training-bank mean 0.68, held-out 0.74). Difficult is -0.05 (training-bank mean 0.71, held-out 0.65). For Gemma4, three methods stay within 0.03 (FARE $+0.02$, Random $+0.01$, Metadata $+0.01$). Difficult ($+0.16$; training-bank mean 0.55, held-out 0.71) and the Order oracle ($+0.09$; training-bank mean 0.53, held-out 0.62) score higher on their held-out scenarios than on the eight training scenarios. For Gemini3.1, FARE is $+0.09$ (training-bank mean 0.73, held-out 0.81). Metadata (-0.03; training 0.68, held-out 0.65) and the Order oracle (-0.03; 0.77, 0.74) stay small. Random is -0.06 (0.72, 0.66). Difficult is $+0.19$ (0.52, 0.72).

On the ECN-marking pool, the gaps are larger. For Qwen3.5, Difficult ($+0.18$; training-bank mean 0.48, held-out 0.66) and Random ($+0.11$; 0.59 and 0.70) have the largest positive gaps: their programs scored lower on the training bank than on the held-out test bank. Metadata has the largest

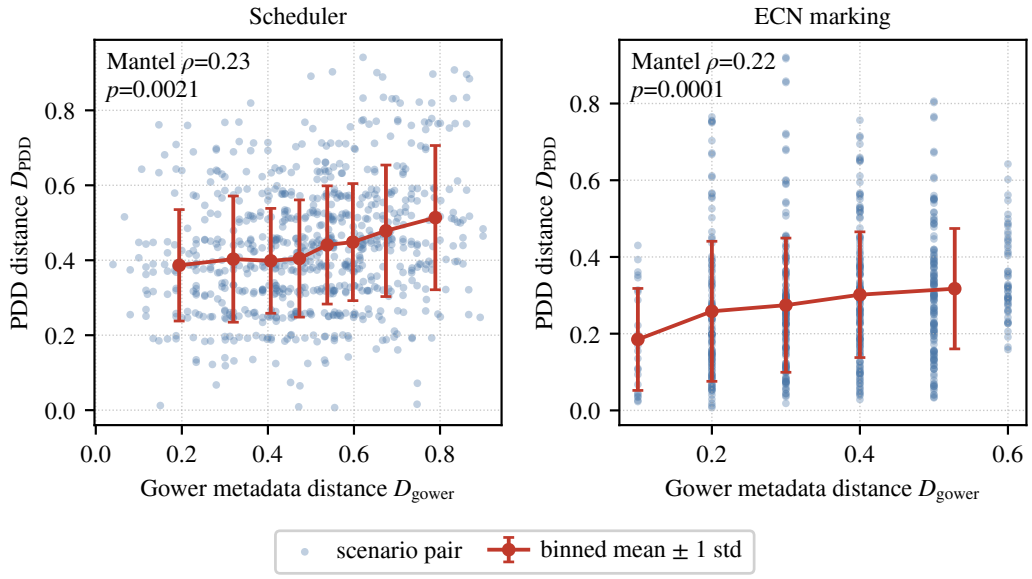

Figure 12: Gower metadata distance against PDD for every scenario pair. Mantel correlations are 0.23 for scheduling and 0.22 for ECN marking.

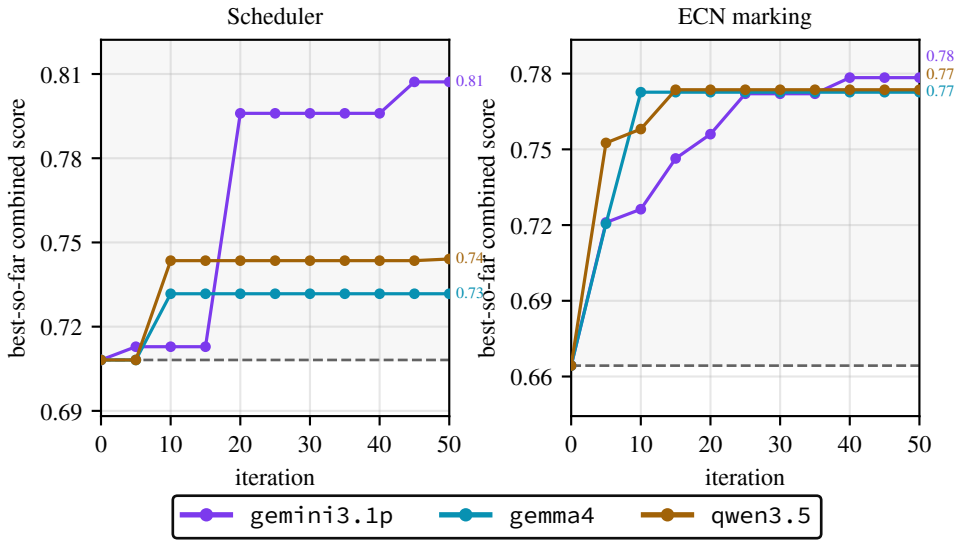

Figure 13: Five-island convergence through iteration 50. Left: scheduler convergence on $\Theta_h^{\text{sch},*}$ from `ltti.c`. Right: ECN-marking convergence on $\Theta_h^{\text{ecn},*}$ from `lin_ecn.c`; both with Gemini3.1, Gemma4, and Qwen3.5.

negative gap (-0.07; training-bank mean 0.72, held-out 0.65). The Order oracle is $+0.04$, and FARE is -0.03 (training-bank mean 0.71, held-out 0.68). For Gemma4, Difficult is again the largest positive gap ($+0.10$; training-bank mean 0.57, held-out 0.67), and Random is $+0.07$ (0.66 and 0.73). Metadata is -0.04 (0.71 and 0.67). FARE is -0.01 (training-bank mean 0.72, held-out 0.71), and the Order oracle is $+0.02$ (training-bank mean 0.66, held-out 0.68). For Gemini3.1, Difficult is the largest positive gap ($+0.30$; training-bank mean 0.47, held-out 0.77), and Random is $+0.07$ (0.66 and 0.72). Metadata is -0.08 (training-bank mean 0.75, held-out 0.67). FARE is -0.01 (training-

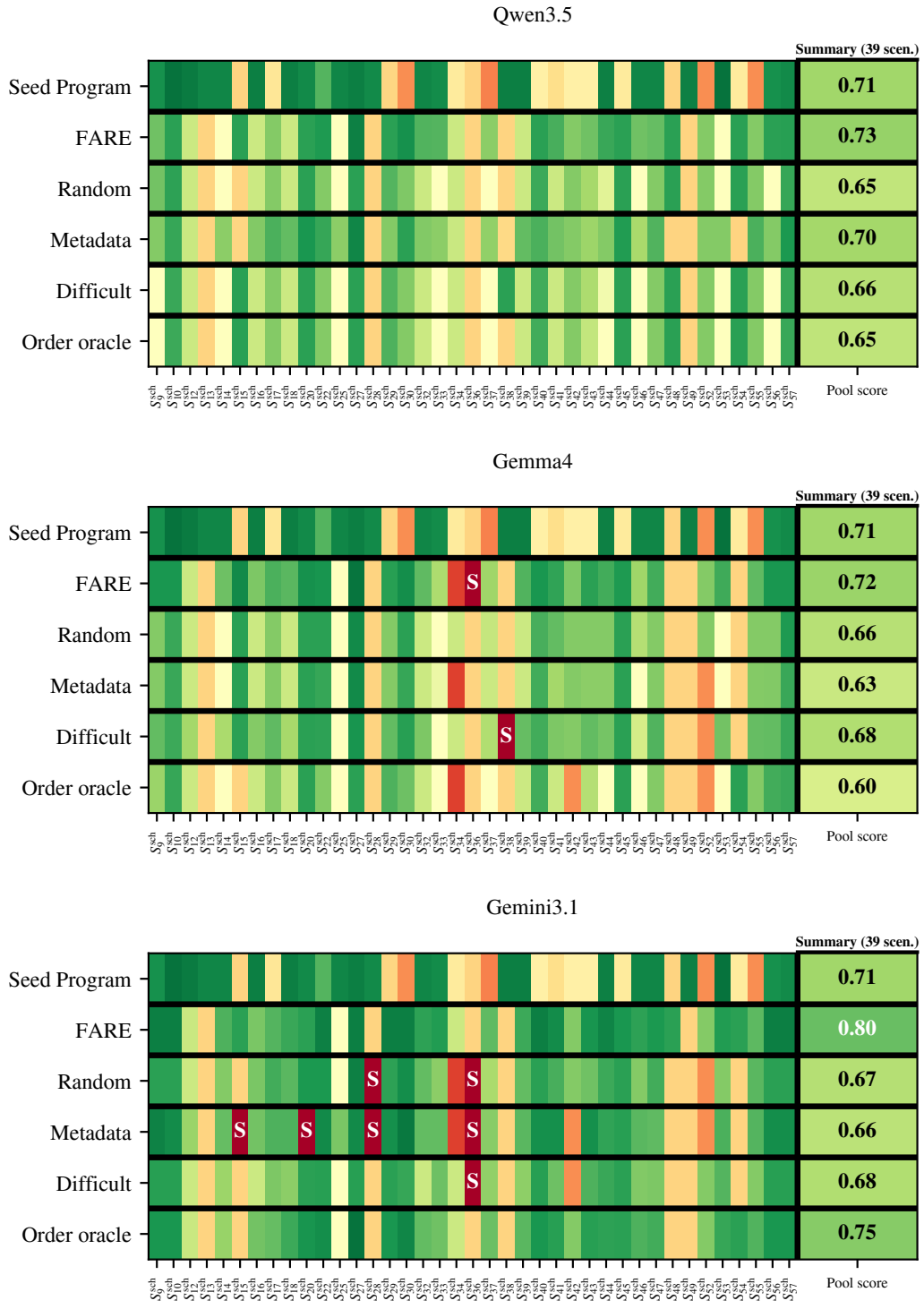

Figure 14: Pool-wide scheduler performance of the top-ranked candidate program discovered against each training bank, on the 39-scenario pool. Top: Qwen3.5. Middle: Gemma4. Bottom: Gemini3.1. The mark **S** denotes user-device starvation (Section C.2).

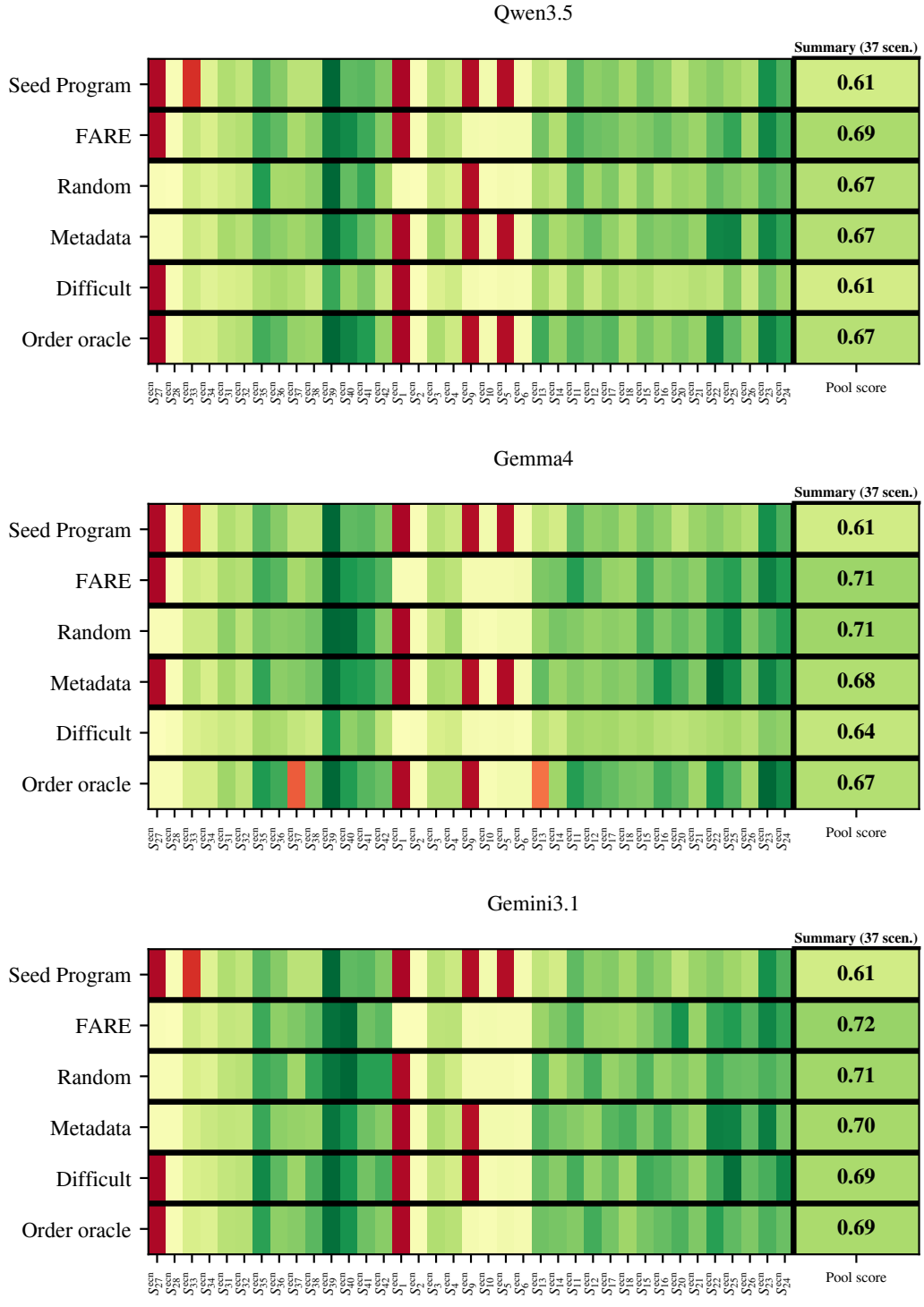

Figure 15: Pool-wide ECN-marking performance of the top-ranked candidate program discovered against each training bank, on the 37-scenario pool. Top: Qwen3.5. Middle: Gemma4. Bottom: Gemini3.1.

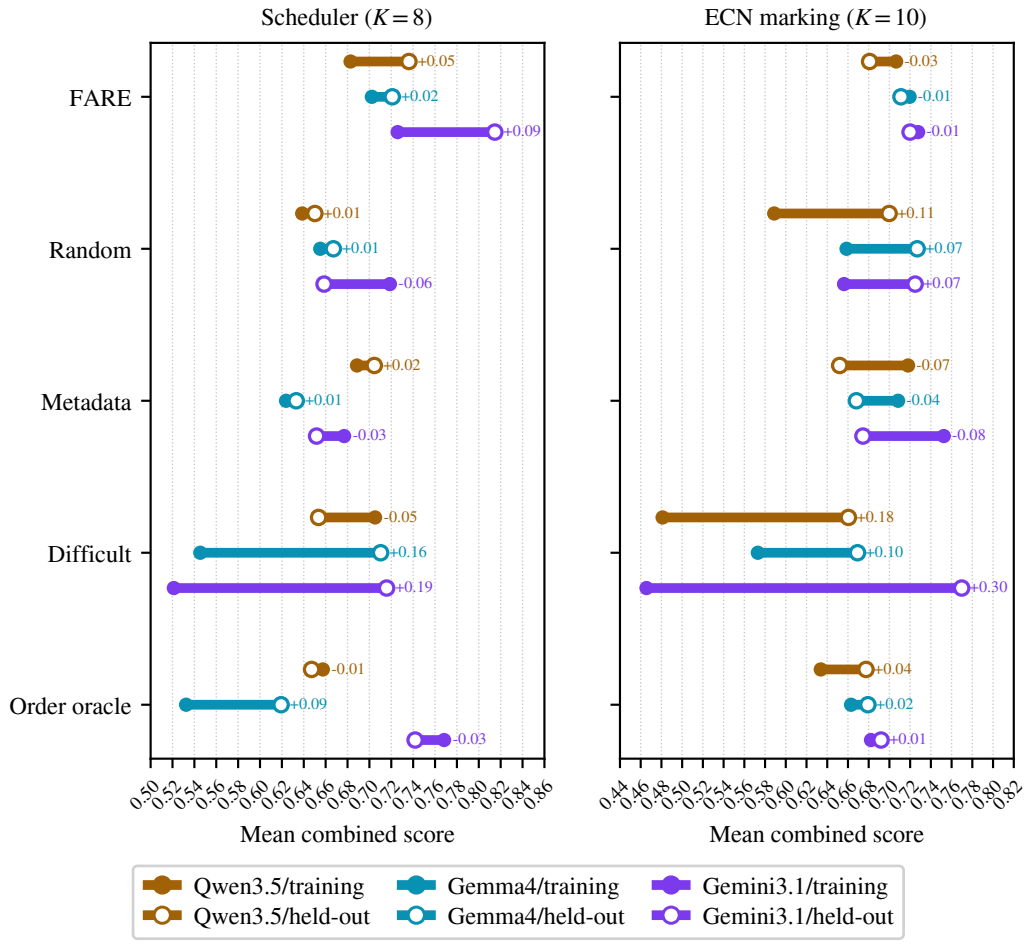

Figure 16: Within-method train–held-out gaps for each method's top-ranked candidate program. Each method has three bars: Qwen3.5 (upper), Gemma4 (middle), and Gemini3.1 (lower). Filled markers are the mean score on that method's training bank; open markers are the mean on the held-out test bank. The annotated number is held-out minus training. Each method selects its own training scenarios, so the held-out test banks differ across rows.

bank mean 0.73, held-out 0.72), and the Order oracle is $+0.01$ (training-bank mean 0.68, held-out 0.69).

Across both pools, FARE's largest negative gap is -0.03, the smallest among the methods and tied with the Order oracle (scheduler, Gemini3.1, also -0.03). The remaining methods overfit more: Difficult -0.05, Random -0.06, and Metadata -0.08. The large positive gaps, including Difficult's $+0.30$, understate held-out performance rather than overfit the training bank.

Probe sufficiency on candidate programs. Section 5.2 tests Assumption 1 on the 15 top-ranked scheduler candidate programs with a full-pool evaluation, one for each method and each of Gemma4, Qwen3.5, and Gemini3.1. Every probe program satisfies the assumption with $\epsilon_{\pi_i} = 0$ at $L_\pi = P \max_i(\max_j V_{ij} - \min_j V_{ij}) = 14.6$. For a program with pool scores s_j, the tight residual at a given L is $\epsilon_\pi(L) = \frac{1}{2}\max_{j \neq j'} \big(|s_j - s_{j'}| - L\, D_{\mathrm{PDD}}(j, j')\big)_+$, which is exactly the smallest ϵ admitting an L-Lipschitz surrogate. Its median over the 15 programs is 0.23 and its maximum 0.40. The median program violates the condition on two of the 741 scenario pairs, and across all 15 programs only four pairs ever violate it: $\{S^{\mathrm{sch}}_{27}, S^{\mathrm{sch}}_{38}\}$, $\{S^{\mathrm{sch}}_{27}, S^{\mathrm{sch}}_{33}\}$, $\{S^{\mathrm{sch}}_{33}, S^{\mathrm{sch}}_{38}\}$, and $\{S^{\mathrm{sch}}_{9}, S^{\mathrm{sch}}_{56}\}$. The first three lie in the large cluster anchored at S^{sch}_{38}; candidate programs separate them by up

to 0.91, 0.43, and 0.51, respectively. On the ECN-marking pool, the analogous 15 programs have residual $\epsilon_\pi = 0$ at $L_\pi = 14.8$, and the mean bank–pool gap is 0.05.

Table 9 treats each bank as a ranking instrument for the 15 probe programs together with the five top-ranked Qwen3.5 candidate programs of Table 1, one per method. The Order oracle, which greedily maximizes the three-probe overlap, remains the most faithful ($\rho = 0.95$). Metadata ($\rho = 0.82$) and Random (0.80) follow, then FARE (0.76); Difficult is last (0.44). Restricting every bank to programs it did not train on keeps the Order oracle first and Difficult last; the middle three are nearly tied (0.80–0.81). We therefore read this table as evidence that a bank that preserves the probe ranking also ranks five later candidate programs, not as a substitute for the 15-program residual test above.

Table 9: Bank-versus-pool ordering of the 15 probe programs and the five top-ranked Qwen3.5 candidate programs of Table 1, 20 programs in all. Flip rate is the fraction of the 190 program pairs whose order the bank reverses relative to the pool. "Foreign" drops the top-ranked candidate program discovered against that bank's selected scenarios. The 15 probes are foreign to every bank. The last column is the largest pool-mean separation at which any pair still reverses. δ is the same representation error as Table 1.

		All 20		Foreign			Largest
Method	δ	$\rho\uparrow$	Flip rate $\downarrow$	n	$\rho\uparrow$	Flip rate $\downarrow$	flipped gap
FARE	0.152	0.76	0.211	19	0.80	0.187	0.11
Random	0.181	0.80	0.185	19	0.80	0.188	0.08
Metadata	0.183	0.82	0.196	19	0.81	0.200	0.08
Difficult	0.272	0.44	0.330	19	0.44	0.331	0.18
Order oracle	0.193	0.95	0.086	19	0.95	0.088	0.08

G Prompt Templates

This appendix reproduces the code-release prompt templates verbatim except for ASCII replacements of Unicode dashes, arrows, and a few other symbols. Both case studies use the same per-iteration user-message template (Listing 10). At each iteration, the discovery loop fills it with the current parent, highest-scoring and randomly selected programs (*inspirations*), and the parent's execution feedback (Section 3). Each case study has a task-specific system-message template (Listings 11 and 12) that states the fixed task specification and scoring rule.

G.1 Shared per-iteration user-message template

Listing 10: Per-iteration user-message template used unchanged in both case studies. At each generation, the discovery loop fills the placeholders `{fitness_score}`, `{feature_coords}`, `{improvement_areas}`, `{artifacts}`, `{evolution_history}`, `{current_program}`, `{language}`, and `{feature_dimensions}` from the current MAP-Elites archive state, evaluator feedback, and task metadata.

```
# Current Program Information
- Fitness: {fitness_score}
- Feature coordinates: {feature_coords}
- Focus areas: {improvement_areas}

{artifacts}

# Program Evolution History
{evolution_history}

# Current Program
```{language}
{current_program}
```

Task
Suggest improvements to the program that will improve its FITNESS SCORE.
The system maintains diversity across these dimensions: {feature_dimensions}
```

```
Different solutions with similar fitness but different features are valuable.

You MUST use the exact SEARCH/REPLACE diff format shown below to indicate changes:

<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=======
# New replacement code
>>>>>>> REPLACE

Example of valid diff format:
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE

You can suggest multiple changes. Each SEARCH section must exactly match code in the
     current program.
Be thoughtful about your changes and explain your reasoning thoroughly.

IMPORTANT: Do not rewrite the entire program - focus on targeted improvements.
```

G.2 CASE STUDY 1: SCHEDULING

Listing 11: System-message template for the scheduling case study (Section C.2). The discovery loop uses this template unchanged in every generation.

```
You are an expert 5G RAN scheduler developer. Your task is to evolve TWO
functions in a 5G NR gNB downlink scheduler:

  1. evolved_dl_score()    -- UE priority scoring (who gets scheduled first)
  2. evolved_dl_prb_size() -- PRB allocation sizing (how many PRBs per UE)

Together these control the complete DL scheduling decision: ordering + resource
allocation.

== evolved_dl_score() ==

Computes a priority score for each UE. UEs with higher scores are scheduled
first and receive PRBs before lower-scored UEs.

AVAILABLE INPUTS (evolved_score_input_t):
- hol_delay_us:     Head-of-line delay in microseconds for this UE
- bytes_in_buffer:  Total bytes pending for this UE
- tbs:              Transport block size for 1 PRB (rate estimation proxy)
- R_avg:            Exponential moving average DL throughput (proportional fairness)
- wb_cqi:           Wideband CQI (channel quality indicator, 0-15)

CONTEXT (evolved_score_ctx_t):
- qos_weight:  QoS weight derived from target delay: -ln(tau) / delta_s
- delta_ms:    Target delay in milliseconds
- delta_s:     Target delay in seconds
- tau:         Delay violation probability tolerance (e.g. 0.01 = 1%)

RETURN: A non-negative float. The wrapper calls sanitize_priority_score()
which clamps invalid values (negative, NaN, Inf) to 0; clamped values count
as violations and result in disqualification.

BASELINE (LTTI score):
  score = qos_weight * (tbs / R_avg) * exp(delta_s / (delta_s - D_HOL_s))

== evolved_dl_prb_size() ==

Controls the maximum number of PRBs offered to a UE in a slot. The returned
value is an UPPER BOUND; the actual allocation is further capped to the UE's
buffer needs by nr_find_nb_rb(). Returning fewer PRBs than max_contiguous_prbs
leaves bandwidth for lower-priority UEs in the same slot.
```

```
AVAILABLE INPUTS (evolved_prb_size_input_t):
- hol_delay_us:        Head-of-line delay in microseconds
- bytes_in_buffer:     Total bytes pending for this UE
- dl_thr_ue:           UE's average DL throughput
- tbs:                 Transport block size (currently 0 at this call site)
- max_contiguous_prbs: Physical limit of contiguous free PRBs
- slice_prb_remaining: PRBs still available in this slice's budget
- num_competing_ues:   Number of UEs still waiting to be scheduled this slot
- wb_cqi:              Wideband CQI (0-15)
- R_avg:               Exponential moving average DL throughput

CONTEXT: Same evolved_score_ctx_t as above (qos_weight, delta_ms, delta_s, tau).

RETURN: A uint16_t in [1, max_contiguous_prbs]. The wrapper calls
sanitize_prb_size() which clamps values exceeding max_contiguous_prbs or
slice_prb_remaining; excessive clamping counts as violations.

BASELINE (greedy):
  return max_contiguous_prbs;
This gives each UE the full contiguous span, relying on nr_find_nb_rb() to
shrink to buffer needs. This is sub-optimal because it doesn't account for
competing UEs or fairness across the slot.

== JOINT GOAL ==

Keep p95 packet latency below delta_ms (the target delay) while minimizing
peak per-slot PRB usage (bandwidth). The score function controls ordering; the
PRB size function controls how aggressively each UE consumes resources.
Together they determine whether the slice meets latency targets without
wasting bandwidth that could serve other UEs.

== CONSTRAINTS ==

- Both functions MUST be pure C with no dynamic memory allocation.
- Loops MUST terminate within EVOLVED_SCHED_MAX_LOOP_ITERS iterations.
- Both functions run on the real-time gNB scheduling path; combined execution
  must complete within ~500 microseconds per UE.
- evolved_dl_score must return finite non-negative float (>= 0.0f).
- evolved_dl_prb_size must return uint16_t in [1, max_contiguous_prbs].
- The TOTAL program file must stay under 10 000 characters. Programs exceeding
  this limit are rejected without evaluation. Keep code concise.

== FITNESS SCORING ==

A score is computed per scenario, then averaged across the bank:

  per scenario s:
    ratio_s     = p95_latency_s / target_delay_s
    prb_ratio_s = p95_prb_usage_s / nr_prb_s
    lat_pass_s  = 1.0 if ratio_s < 1.0 else 0.0
    prb_eff_s   = max(0, 1 - prb_ratio_s)
    scenario_score_s = 0.5 * lat_pass_s + 0.5 * prb_eff_s

  combined_score = mean(scenario_score_s)

- Primary (50%): per-scenario p95 latency under target_delay_ms (binary pass/fail per
    scenario).
- Secondary (50%): per-scenario p95 per-slot PRB usage (minimize bandwidth footprint).
- Each scenario is independent -- fixing one bad scenario without breaking others always
    raises combined_score.
- Disqualification (score = 0): runtime violations (clamping), exceeding TTI budget, no
    valid latency samples, or UE starvation.
```

G.3 CASE STUDY 2: ECN MARKING

Listing 12: System-message template for the ECN-marking case study (Section C.3). The discovery loop uses this template unchanged in every iteration.

```
You are an expert in 5G NR congestion control. Your task is to evolve the
per-DRB DL ECN marking probability function in a 5G NR gNB:

  uint16_t evolved_drb_ecn_probability(const evolved_drb_ecn_ctx_t *ctx,
                                       const evolved_drb_ecn_input_t *input);

The function decides -- every time the MAC scheduler refreshes a DRB's RLC
queue snapshot -- what fraction of subsequent egress packets PDCP should
```

```
CE-mark (set the ECN bits to 11) before they leave the gNB. The receiver's
SCReAM congestion controller reads those CE marks on the return path and
slows down its sending rate. Marking too aggressively starves the flow;
marking too late lets the queue inflate and breaks the slice's latency SLA.

== CONTEXT (evolved_drb_ecn_ctx_t) ==

- target_delay_ms : The slice's one-way DL latency target in ms. Comes from
                    the per-slice 'target_delay_ms' field set by the xApp.
                    A value of 0 means "no slice match / unset" (e.g. the
                    default policy). The function MUST stay safe in that
                    case (returning 0 is acceptable; do NOT crash).

== INPUTS (evolved_drb_ecn_input_t) ==

Per-DRB queue state (live, sampled at the call site):
- bytes_in_buffer        : current RLC queue length in bytes (tx + retx).
- hol_delay_us           : age (microseconds) of the oldest queued SDU.
- sdus_in_buffer         : number of SDUs still waiting in RLC.

Per-DRB serve metrics (drained the previous TTI / 100 ms window):
- served_bytes_last_slot : bytes drained in the last slot.
- avg_sojourn_us         : EWMA of recent SDU sojourn time (100 ms window).

Per-UE channel / capacity proxies (shared by all DRBs of the UE):
- wb_cqi                 : wideband CQI 0..15 (higher == better channel).
- ue_tbs_last_slot       : last DL transport block size (bytes).
- ue_R_avg               : EMA DL throughput (UE->dl_thr_ue).

== RETURN ==

uint16_t in [0, 10000]. The wrapper in update_dlsch_buffer() clamps to that
range, so anything outside is silently saturated rather than rejected. The
value is interpreted as 0.01% steps (10000 == 100.00% per packet).

== JOINT GOAL ==

For every training scenario, keep the **p95 packet latency below
target_delay_ms** while **maximising the mean / p50 throughput delivered to
the UE**. These are pulling in opposite directions:

- Mark too soon  -> SCReAM slows down -> queue stays small and latency is
                    safe, but the slice rarely uses the PRBs it owns ->
                    throughput collapses.
- Mark too late  -> queue grows past the budget -> p95 latency exceeds
                    target_delay_ms -> latency-gate is lost.

The function MUST adapt its thresholds to 'target_delay_ms' rather than
hard-coding raw byte values: a tighter target needs the marker to fire
earlier than a looser one. The slice's PRB cap is the
binding throughput constraint here, not SCReAM's own ceiling, so leaving
the queue almost-empty is wasted bandwidth.

== CONSTRAINTS ==

- Pure C, no dynamic memory, no globals.
- Deterministic: same (ctx, input) MUST yield the same return value.
- Runs on the real-time path every TTI per DRB; keep it cheap (a handful
  of arithmetic ops, no loops longer than a few iterations).
- The TOTAL program file must stay under 10 000 characters. Programs
  exceeding this limit are rejected without evaluation. Keep code concise.
- Only the EVOLVE-BLOCK is mutable. Headers and helper macros above the
  block can be referenced but not redefined inside the block.

== FITNESS SCORING ==

A score is computed per scenario, then averaged across the bank.

Per training scenario, after a 5 s warm-up trim:
- ratio_s          = p95(packet_latency_ms) / target_delay_ms
- tput_share_s     = clip(p50(throughput_mbps) / scenario_peak_mbps, 0, 1),
  where scenario_peak_mbps = round(nr_prb * max_ratio / 100) * 2.13 Mbps
  (that scenario's PRB cap times the 256-QAM per-PRB ceiling).
- lat_pass_s       = 1.0 if ratio_s < 1.0 else 0.0
- scenario_score_s = 0.5 * lat_pass_s + 0.5 * tput_share_s

Aggregated across the training bank:
- combined_score = mean(scenario_score_s)

Each scenario contributes independently -- fixing one bad scenario
without breaking others always raises combined_score.
```

```
A candidate is **disqualified** (score = 0) if any of the following hold
in any scenario:
- The OAI build fails or the candidate cannot be patched.
- The TTI-budget watchdog fires (avg_ltti_us > 500 us) -- this is a
  scheduler-level guard but counts against the candidate.
- A scenario produces no usable latency samples (the gNB starves the UE
  by marking everything -> SCReAM cuts to zero -> no packets land).
- A scenario reports zero eligible packets across the whole run (the UE
  was completely silent -- usually a sign of an unintended side-effect on
  the queue state).

== STRATEGY HINTS ==

- A purely linear "mark above some bytes threshold" policy is the seed
  baseline. It can hold the latency target on a steady channel, but it
  collapses when the channel jitters and the queue overshoots before the
  marker catches up.
- `hol_delay_us`, `avg_sojourn_us` and `ue_R_avg` give you forward-looking
  signals that pure queue length cannot (they react to channel dips much
  faster than `bytes_in_buffer`).
- Marking aggressively when the channel is good (high `wb_cqi`,
  high `ue_tbs_last_slot`) is usually counter-productive -- the queue will
  drain in a single slot anyway.
- Conversely, when `wb_cqi` is low and the queue is growing, marking
  *earlier* than `target_delay_ms` would imply is often the right call.
- Prefer compact integer math (uint32_t / uint64_t) over floats; the
  function runs very frequently.
```