US DOT National
University Transportation Center for Safety

Carnegie Mellon University

Safe Driving Across Domains

Peter Zhang (PI)
ORCID: https://orcid.org/0000-0002-0422-834X

Pingbo Tang (Co-PI)
ORCID: https://orcid.org/0000-0002-4910-1326

Final Report – August 31, 2026

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Safe Driving Across Domains		**5. Report Date** August 31, 2026
		6. Performing Organization Code
7. Author(s) Peter Zhang, Ph.D. https://orcid.org/0000-0002-0422-834X; Pingbo Tang, Ph.D. https://orcid.org/0000-0002-4910-1326		**8. Performing Organization Report No.**
9. Performing Organization Name and Address Carnegie Mellon University 5000 Forbes Avenue Pittsburgh, PA 15213		**10. Work Unit No.**
		11. Contract or Grant No. Federal Grant No. 69A3552344811
12. Sponsoring Agency Name and Address U.S. Department of Transportation Office of the Assistant Secretary for Research and Technology University Transportation Centers Program 1200 New Jersey Avenue SE, Washington, DC 20590		**13. Type of Report and Period Covered** Final Report (July 1, 2025 – July 31, 2026)
		14. Sponsoring Agency Code USDOT

15. Supplementary Notes
Conducted through the Safety21 National University Transportation Center, sponsored by the U.S. Department of Transportation, Office of the Assistant Secretary for Research and Technology. Project record: https://ppms.cit.cmu.edu/projects/detail/593
Project software: https://github.com/Fake-fate11/camp-core

16. Abstract
Self-driving systems are designed and tested for specific operating conditions, called operational design domains. Moving a system to new roads, new cities, or new traffic rules normally requires retraining it on large amounts of local data, which is slow, expensive, and hard to audit. This project developed and tested a different approach. Modern self-driving planners generate several candidate paths at every moment, and the vehicle executes one of them. The project's method, called CAMP, leaves the planner untouched and improves only the final choice among the candidate paths. It scores each candidate on explicit, readable measures of safety, comfort, and progress, learns how to weigh those measures by studying recorded human driving, and adjusts the weighting to the situation at hand. In tests on 66,843 driving scenes from a public benchmark, CAMP reduced the share of scenes with a safety violation from 82.0 percent to 60.2 percent, the best result possible with the given candidates. In tests across Boston, Pittsburgh, and Singapore, most safety benefits carried over to a region the system had never seen. In 480 full driving simulations in the open-source Autoware platform, vehicles using CAMP completed every run without a collision. The project's software is publicly available as open-source code at https://github.com/Fake-fate11/camp-core. The results show that improving the final choice is a practical, low-cost, and auditable way to extend automated driving safely to new places.

17. Key Words	18. Distribution Statement
Autonomous vehicles; Motion planning; Trajectory selection; Operational design domain; Risk analysis; Machine learning; Open source software; Traffic safety	No restrictions. This document is available through the National Technical Information Service, Springfield, VA 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	29	
Form DOT F 1700.7 (8-72)		Reproduction of completed page authorized	

Contents

Executive Summary

Self-driving technology has reached the point where a vehicle can handle many ordinary driving situations on its own. But every self-driving system is designed, trained, and tested for particular conditions: certain kinds of roads, traffic, signs, and rules. Engineers call this set of conditions the system's operational design domain. Move the system somewhere new, from downtown intersections to hilly rural roads, or from an American city to one where traffic drives on the left, and its safety can no longer be taken for granted. The standard remedy is to retrain the system on large amounts of new local data. That is slow, expensive, and out of reach for most communities, and the retrained system is a black box whose behavior must be revalidated from the beginning.

This project developed and tested a different approach. A modern self-driving planner does not produce a single plan. Several times per second it generates a handful of reasonable candidate paths, and the vehicle executes one of them. We leave the planner exactly as it is and improve only the final step: the rule that decides which candidate path the vehicle follows. That rule is small, inexpensive to train, and easy to inspect, because it scores each candidate on explicit measures that people can read and debate, such as the distance kept from other road users, staying on the road, obeying speed limits, ride smoothness, and progress toward the destination. The rule learns how to weigh these measures by studying recorded human driving, and it adjusts the weighting to the situation at hand. We call the method CAMP.

We tested CAMP at three levels of realism, using only open-source driving software and public research datasets. First, on 66,843 driving scenes from a standard research benchmark, CAMP cut the share of scenes in which the chosen path violated a safety check from 82.0 percent to 60.2 percent, which is the best any choosing rule could possibly do with the candidates it was given. Second, on more than 56,000 test scenes from Boston, Pittsburgh, and Singapore, CAMP chose paths that stayed on the road more, kept more distance from other road users, and tracked human driving more closely, and most of these benefits held in Singapore, a region the system had never seen during training. Third, in 480 full closed-loop driving simulations in Autoware, the leading open-source self-driving platform, vehicles using CAMP completed every run without a collision.

The project's code is publicly available as open-source software that works with the Autoware platform, so any company, researcher, or public agency can use and inspect it. For transportation agencies, the central lesson is that the safety of an automated vehicle in a new environment depends on two separate things: whether the system can propose a safe path at all, and whether it picks that path when it exists. These two questions can and should be evaluated separately.

This report is written to be self-contained. It explains every concept from the ground up and assumes no technical background.

1 Introduction

About 40,000 people die on American roads every year, and an estimated 1.35 million die worldwide. Automated driving is one of the most promising tools for reducing those numbers, because most crashes involve human error. But automated driving only helps the places that have it, and today the technology is concentrated in a small number of carefully mapped, data-rich cities. The places with some of the greatest safety needs, including rural areas and lower-income regions, are the least likely to have the data and money that current methods require.

The reason is that a self-driving system is not a general driver. It is engineered for a specific envelope of conditions: which kinds of roads, what weather, what traffic, which side of the road, what signage. Within that envelope it can be very good. Outside it, nobody can promise how it behaves. Extending the envelope today usually means collecting millions of miles of new local driving data and retraining the system, then revalidating it, which can take years and cost more than most cities or companies can afford.

This project asked whether there is a cheaper, faster, and more transparent path. Our research question, as posed in the project proposal, was: how can pretrained autonomous driving models be adapted across diverse operational design domains to achieve safe and reliable deployment, without retraining them from scratch? The project's answer, developed and tested over the 2025–2026 project year, is to leave the pretrained model alone and improve the one small decision it feeds: the choice among the several candidate paths the model already proposes. Section 2 explains the background needed to see why that decision exists and why it matters. Sections 3 through 5 describe what we built, how we tested it, and what we found. Sections 6 through 9 cover implications, project outputs, data management, and funding.

2 Background

2.1 How a self-driving car decides what to do

A self-driving vehicle runs a continuous loop with a few distinct jobs, shown in Figure 1. Cameras, laser rangefinders (lidar), radar, and digital maps tell the vehicle what is around it. Software then interprets that information: where the lanes are, where other cars, cyclists, and pedestrians are, and where each of them is likely to go next. A planning component then proposes what the vehicle itself should do over the next several seconds. Finally, a control component turns the chosen plan into steering, braking, and acceleration.

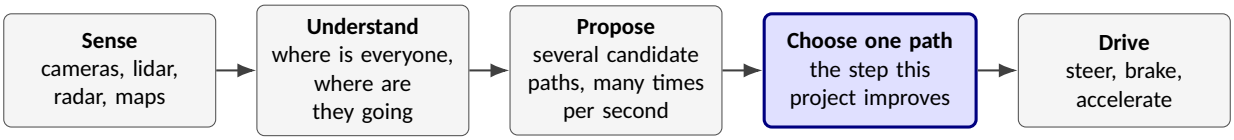

Figure 1: How a self-driving vehicle works, in five steps. This project improves the fourth step: choosing which of several proposed paths the vehicle actually follows.

The key fact this project builds on is in the third and fourth steps. Modern planning software, especially the newer kind built with machine learning, does not produce one plan. It produces several plausible candidate paths at once, typically covering the next eight seconds, and it refreshes them many times per second. The candidates genuinely differ: one may leave more room around a cyclist, another may give a smoother ride, another may make more progress through an intersection. Figure 2 shows the situation. Some component must then pick exactly one candidate for the vehicle to execute.

Surprisingly, in today's systems that final pick often gets little attention. A common default is simply to execute the planner's first or highest-ranked suggestion. But the order in which a planner happens to generate its candidates is a byproduct of how it was trained, not a judgment about what matters right now, in this scene, on this road. That gap between "what the planner happens to rank first" and "what is actually best here" is the opportunity this project exploits.

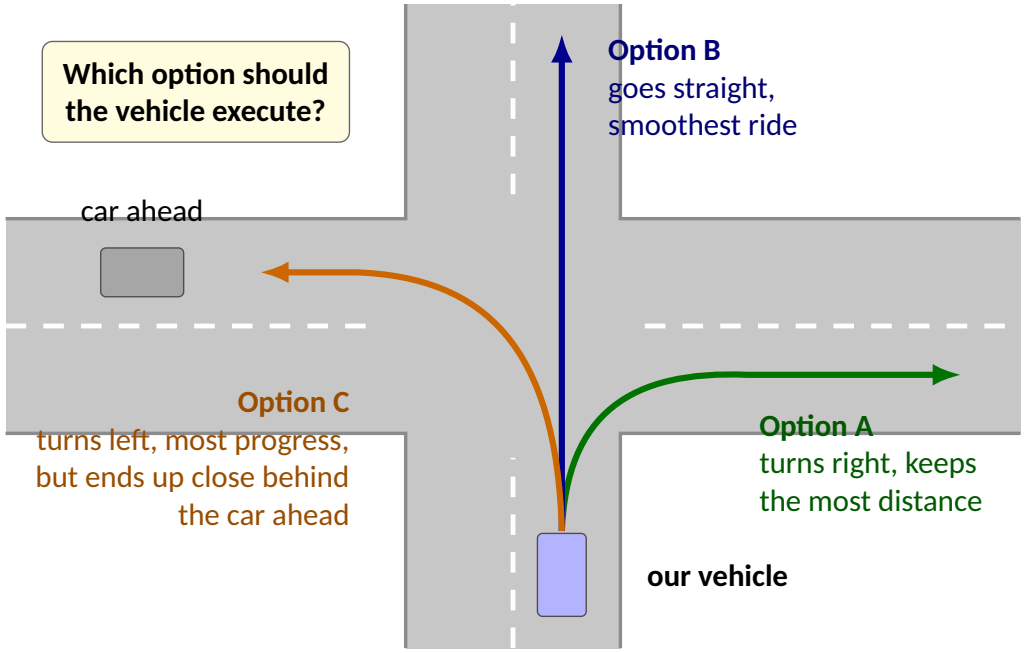

Figure 2: The decision this project improves. The planner proposes several reasonable paths for the next few seconds, and each has different strengths. The vehicle must execute exactly one of them.

2.2 The software stack: what each layer is for

Figure 1 compressed the vehicle's software into five steps. Because the rest of this report keeps referring to pieces of that software, it is worth one section to lay out the full stack accurately. Figure 3 shows the standard architecture of a connected and automated vehicle (CAV) software system. Each layer consumes the output of the layer above it, and the whole loop repeats continuously, typically ten or more times per second.

Sensing measures the physical world. Cameras capture appearance, lidar (laser ranging) measures precise distances, radar measures distance and speed even in rain or fog, and satellite positioning plus inertial sensors track the vehicle's own motion. Each sensor covers weaknesses of the others, which is why vehicles carry several kinds.

Perception and localization turn raw measurements into a usable picture. Perception detects and tracks the objects that matter, including vehicles, cyclists, pedestrians, lane markings, and traffic signals, and estimates each one's position, size, and velocity. Localization answers a deceptively hard question, "where exactly am I?", by matching sensor data against a detailed digital map, usually to within a few centimeters. The maps themselves, often called high-definition maps, store lane geometry, connectivity, speed limits, and signal locations in machine-readable form.

Prediction looks a few seconds into the future. Given the tracked objects, it forecasts where each is likely to go: will that car yield, is that pedestrian about to cross? Predictions are inherently uncertain, so modern predictors produce several plausible futures for each road user rather than a single guess.

Planning decides what the vehicle itself should do, and it operates at three distinct scales. Route planning picks which streets to take, the job of an ordinary navigation app. Behavior planning

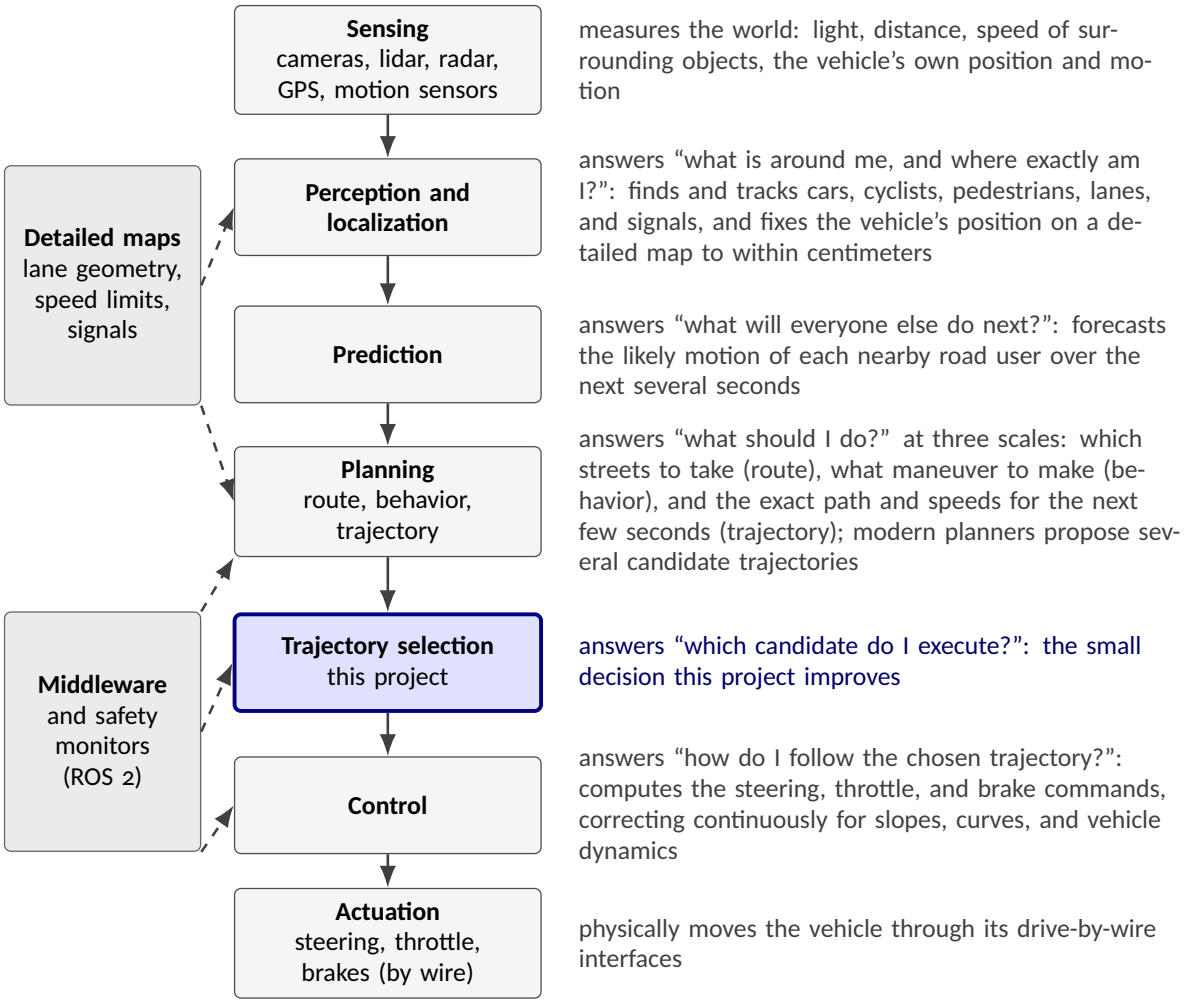

Figure 3: The software stack of a connected and automated vehicle, and what each layer is for. This project adds an explicit, learnable trajectory selection layer between planning and control, leaving every other layer unchanged.

makes maneuver-level decisions: change lanes, yield at the merge, stop for the signal. Trajectory planning (also called motion planning) produces the precise output the vehicle needs: an exact path with speeds along it, covering the next few seconds, that is physically drivable and avoids obstacles. In classical systems these were separate hand-engineered modules; in newer systems, machine-learned planners such as the two used in this project handle the behavior and trajectory scales together, and, as Section 2.1 explained, they naturally produce several candidate trajectories at once.

Trajectory selection, the highlighted layer, is where this project lives. Something must reduce the planner's several candidates to the single trajectory the vehicle will follow. Today this step is often implicit, buried in the planner's internal ranking. Our contribution is to make it an explicit, learnable, and auditable layer of the stack.

Control answers "how do I actually follow the chosen trajectory?" It runs faster than every layer above it, typically tens to hundreds of times per second, computing the steering angle, throttle, and braking that keep the vehicle on the chosen trajectory while compensating for slopes, curves, tire behavior, and disturbances. Control is a mature engineering discipline; given a good trajectory, modern controllers follow it closely. That is exactly why the quality of the trajectory handed to control matters so much: control faithfully executes whatever it is given, good or bad.

Actuation is the hardware layer: drive-by-wire interfaces that physically turn the wheel and apply throttle and brakes on electronic command.

Two supporting elements run alongside the chain. **Middleware** is the communication fabric that lets dozens of software components exchange data reliably and on time; the open-source standard is ROS 2 (Robot Operating System 2), which Autoware is built on. **Safety monitors** watch the whole system and trigger fallback behavior, such as a controlled stop, if a component fails or reports implausible values.

This layered picture explains why the project's approach is economical. The expensive layers to build and retrain are perception, prediction, and planning, which contain large learned models. The layer we modify sits below them, is small, and consumes only their outputs. Improving it requires no new sensors, no new maps, no retraining of any large model, and no change to control or actuation.

2.3 Operational design domains: why place matters

An operational design domain, or ODD, is the formal description of where and when an automated system is designed to work: road types, speed ranges, weather, lighting, traffic mix, and local rules. The concept originated in aviation and is now standard in automated vehicle engineering and regulation. A system's safety case applies inside its ODD and says nothing outside it.

ODDs matter to this project because they are where the economics of automated driving break down. Pittsburgh's dense urban grid, Pittsburgh's steep and narrow rural routes, and Singapore's left-hand traffic are three different ODDs. A planner trained mostly on one of them has learned habits that may not fit the others. Collecting enough local data to retrain the planner for every new ODD is the expensive path. This project's premise is that much of the needed adjustment is not about generating different paths at all. The planner often already proposes a suitable path among its candidates; what changes from place to place is which candidate should win. Adjusting how candidates are judged is vastly cheaper than reteaching the planner how to generate them.

2.4 Open-source driving software, and why this project uses it

Most commercial self-driving systems are proprietary: their software cannot be inspected, modified, or reused by outsiders. Open-source software is the opposite: the code is public, anyone may use and modify it, and improvements can be shared. For public agencies and researchers, open source matters for two reasons. It makes independent safety scrutiny possible, and it gives smaller companies, cities, and countries access to technology they could never build alone.

The center of gravity for open-source automated driving is Autoware, a complete self-driving software platform maintained by the Autoware Foundation, an international consortium of companies and universities. Autoware is used by researchers and companies worldwide, and it implements the full stack of Figure 3 as interchangeable modules (perception, localization, prediction, planning, control) communicating over the ROS 2 middleware, which is what makes it possible to insert and test a new layer the way this project does. This project made a deliberate scoping decision to build everything on and for this open ecosystem: the two planners we adapt (Trajectron++ and Diffusion Planner) are public research systems, the driving stack we test in is Autoware, the datasets are public research benchmarks, and the project's own code is publicly released as open source (Section 7.2). The intended contribution is a reusable, inspectable decision layer for the open self-driving software stack.

2.5 Learning from human drivers

How should the vehicle judge which candidate path is best? Human drivers make this kind of judgment constantly and mostly well. The research field has long used an idea called learning from demonstration, closely related to what specialists call inverse reinforcement learning: instead of programming the judgment by hand, observe what experienced humans actually did in recorded driving, and work backward to infer the priorities that explain their behavior. This project uses exactly that idea, applied to recorded driving logs in which we can see both what a human driver did and what the planner's candidates would have been in the same situation.

2.6 Testing in simulation: open loop and closed loop

Testing automated driving on public roads is expensive and, for unproven methods, unsafe. This project therefore tests in simulation, which is standard practice and allows enormous scale: hundreds of thousands of recorded situations can be replayed exactly, and every method can be compared on the identical scenes. Two kinds of simulation matter in this report. In *open-loop* testing, each decision is graded on paper: we ask what the method would have chosen in a recorded scene and score that choice, but the choice does not change what happens next, because the recording is fixed. In *closed-loop* testing, the vehicle actually drives inside a virtual world: its decisions change its position, which changes what it sees next, and mistakes compound exactly as they would on a real road. Closed-loop testing is the more demanding and more realistic standard, and this project used both.

3 What We Built: Choosing Better Without Retraining

3.1 The idea in one picture

Our method is called CAMP (short for Conic Atom Meta-Policy, a name that matters only to specialists). Figure 4 shows how it works. A useful way to think about it: the planner is a kitchen that offers a short menu of dishes. Retraining the planner means retraining the kitchen, which is slow and expensive, and afterward you must taste-test everything again. CAMP instead learns to order well from the menu that is already offered.

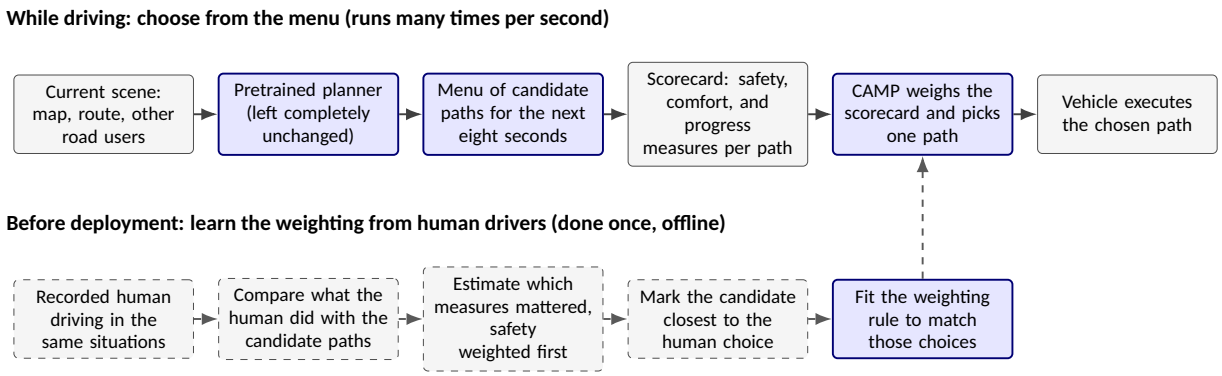

Figure 4: How CAMP works. While driving (top row), the pretrained planner is left untouched; CAMP only scores the planner's candidate paths on an explicit scorecard and picks one. Before deployment (bottom row), the weighting of the scorecard is learned from recorded human driving.

Three design choices carry the weight.

First, an explicit scorecard. Every candidate path is measured on a list of concrete, human-readable criteria: how close it comes to other road users, whether it would leave the road, whether it would break the speed limit or enter an intersection against a red light, how hard it accelerates and jerks (the standard measure of ride discomfort), how much progress it makes toward the destination, and how consistent it is with the plan the vehicle was already executing a moment earlier (sudden plan changes make a vehicle erratic). Depending on the setting we use nine or sixteen such measures. Because the scorecard is explicit, anyone can audit what the vehicle is optimizing, which is not true of decisions buried inside a large neural network.

Second, weights learned from human driving. A scorecard needs weights: how much should clearance matter relative to smoothness, or progress relative to caution? Rather than set these by hand, we estimate them from recorded human driving, as described in Section 2.4. Concretely, for each recorded scene we compare the path the human actually took against the candidate paths the planner would have offered, and we find the weighting under which the human's choice comes out best. We additionally require, as a built-in constraint, that safety measures receive at least as much weight as comfort measures, and comfort at least as much as everything else. The result is a weighting grounded in observed human judgment but with safety guaranteed to come first.

Third, weights that adapt to the situation. A fixed weighting is already useful, but the right trade-off changes with context: a narrow street crowded with pedestrians calls for different priorities than an empty arterial road. The full version of CAMP therefore lets the weights vary with the scene, computed from the planner's own internal summary of the current situation. This is precisely

the mechanism that lets one system adapt across operational design domains: nothing about the planner changes between Pittsburgh and Singapore, but the weighting of the scorecard does. In our experiments we compare both versions, called fixed weights and scene-adjusted weights in the figures below.

3.2 Why the training is trustworthy and cheap

One more property matters, and it can be stated without mathematics. The problem of fitting CAMP's weights belongs to a class that mathematicians call convex optimization. Convex problems have a special guarantee: they have no misleading partial solutions, so a standard solver reliably finds the best possible fit, the same way every time. This is different from training a large neural network, which is a trial-and-error process without such guarantees. In the accompanying technical paper we prove this property of CAMP and use a classical technique called Benders decomposition, dating to 1962, to solve the problem efficiently even with hundreds of thousands of training scenes. Two practical consequences follow. Training CAMP takes modest computing resources, not data-center scale. And the training outcome is reproducible, which matters for any process that a safety regulator might one day need to certify. Readers who want the precise formulation will find it in the paper listed in Section 7 and in the appendices: Appendix A gives a compact technical summary, Appendix B explains the underlying two-stage optimization problem, why it is inherently non-convex, and how a convex envelope bounds it, and Appendix C reports controlled synthetic experiments on when that convexification works well and when it does not.

4 How We Tested It

Our evaluation principle is shown in Figure 5: every method being compared sees the identical scenes and the identical candidate menus, and is graded on the identical report card. Differences in results can then only come from how each method chooses.

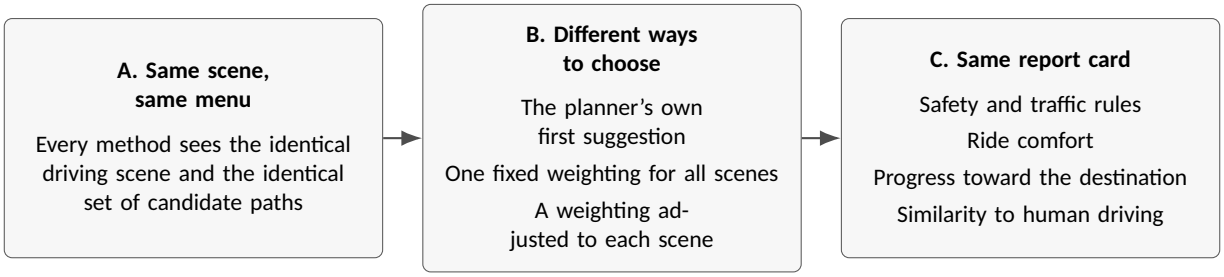

Figure 5: The comparison design. Because every method chooses from the same menu in the same scenes and is graded the same way, any difference in outcomes is caused by the choosing rule alone.

We measured outcomes that a non-specialist can read directly:

- **Safety violations**: the share of scenes in which the chosen path failed at least one safety check, such as coming too close to another road user, leaving the drivable road, or exceeding the speed limit.
- **Collisions and near-collisions**: the share of time the chosen path overlapped or nearly overlapped another road user's position.

- **Staying on the road**: the share of time the path remained fully within the drivable area.
- **Closeness to human driving**: how far, in meters, the chosen path strayed from what the human driver actually did in the same situation, averaged over the path and at its endpoint. Human behavior is not a perfect standard, but large deviations from it are a warning sign.
- **Ride comfort**: standard measures of acceleration and jerk, the physical quantities passengers feel as rough or smooth.
- **Progress**: distance advanced toward the destination, to check that safety is not being bought with paralysis.

The evaluation had three stages of increasing realism.

Stage 1, open-loop selection at scale. We used nuScenes, a public research dataset recorded in Boston and Singapore, with a fixed, publicly available planner called Trajectron++. The test set contains 66,843 scenes, and in each one the planner offers 50 candidate paths. We compared CAMP against the planner's own top suggestion, against a conventional neural network trained to re-rank the same candidates, against directly retraining the planner (the expensive alternative CAMP is meant to avoid), and against a theoretical best case.

Stage 2, repeated planning across three cities. We used nuPlan, a larger public dataset built for planning research, with Diffusion Planner, a state-of-the-art open-source planner distributed publicly for the Autoware ecosystem. Here the planner replans continuously, and each menu has eight timed candidate paths. CAMP was trained on recorded scenes from Boston and Pittsburgh (in two sizes, 50,000 and 200,000 scenes) and tested on 24,246 held-back Boston and Pittsburgh scenes it had never seen. Crucially, the same trained system was then tested on 32,328 scenes from Singapore, a city absent from training, with different road geometry and left-hand traffic. This is a direct test of adaptation across operational design domains.

Stage 3, closed-loop driving. Finally we put CAMP inside the full Autoware stack (Autoware 1.8.0 with Autoware Universe 0.51.0) together with the open-source Scenario Simulator v2, and let the vehicle actually drive 120 standardized test scenarios, including situations involving limited detection range, speed differences with other traffic, and traffic signals. All four CAMP variants (fixed and scene-adjusted weights, trained on the smaller and larger populations) drove the identical 120 scenarios, 480 simulated runs in total, with every decision feeding back into the vehicle's next situation.

No new road data and no personal information were collected for this research. Everything ran on existing public datasets and open-source software under their licenses.

5 What We Found

5.1 Better choices, up to the limit of the menu

Figure 6 shows the central result of Stage 1. When the vehicle simply executes the planner's own first suggestion, 82.0 percent of the 66,843 test scenes contain at least one safety violation. With CAMP choosing from the same 50 candidates, the violation rate falls to 60.2 percent.

The most important number in this figure is the red line. In 60.2 percent of the scenes, none of the planner's 50 candidates passed every safety check. A choosing rule can only pick from what it is offered; when no safe option is on the menu, no rule can fix that. So 60.2 percent is a hard floor,

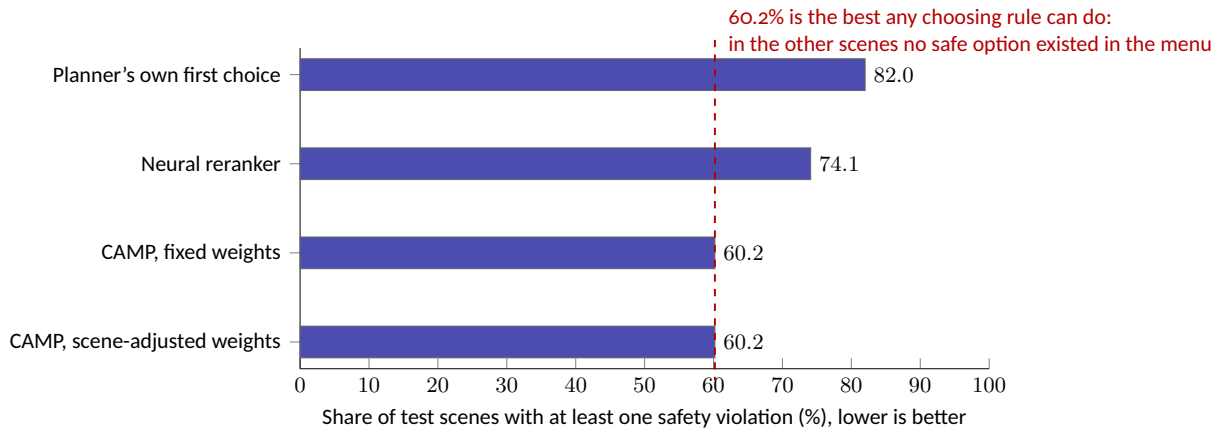

Figure 6: Stage 1 result on 66,843 test scenes with 50 candidate paths each. Both CAMP variants reach 60.2 percent, which is exactly the floor: in those scenes, 60.2 percent of the total, not one of the 50 candidates passed every safety check, so no choosing rule could do better. A conventional neural reranker trained on the same information does not reach the floor.

and CAMP reaches it exactly, meaning that whenever a safe candidate existed, CAMP found it. CAMP also cut our measure of risk in the worst tenth of scenes by nearly half (from 1.95 to 1.08 on the scale used in the technical paper). And the menu itself matters: when we shrank the menu from 50 candidates to 12, the best attainable violation rate worsened by 4.5 percentage points for every method.

For comparison, we also did the expensive thing this project is designed to avoid: retraining the planner itself with a safety objective. Retraining produced paths that imitate human driving more closely and ride more smoothly, but its violation rate (69.7 percent) stayed well above the floor that CAMP reached without touching the planner. The two approaches improve different things, and the comparison confirms that better choosing is not a substitute for a better planner, nor the reverse. Better choosing is, however, available at a tiny fraction of the cost.

5.2 The benefits carried over to an unseen region

Figure 7 shows the headline Stage 2 results. In Boston and Pittsburgh, on 24,246 test scenes, CAMP with scene-adjusted weights reduced collisions and near-collisions from 3.42 percent to 3.01 percent of the measured exposure, kept the vehicle fully on the road a larger share of the time, and tracked human driving more closely (for example, the endpoint of the chosen eight-second path landed on average about one meter closer to where the human driver actually ended up).

The Singapore columns are the test that matters most for this project's purpose. Singapore was completely absent from training, and its roads differ in geometry and drive on the left. Even so, the same trained system reduced collisions and near-collisions, increased time fully on the road by two percentage points, and tracked human driving more closely than the planner's default choice. Adaptation at the decision layer, learned in one region, carried most of its value into another. That is the project's research question answered in the affirmative, within the limits of simulation.

Honesty requires reporting what did not improve. Choosing more cautiously costs a little progress: a few hundredths of a meter of forward advance per eight-second window in these tests. Training

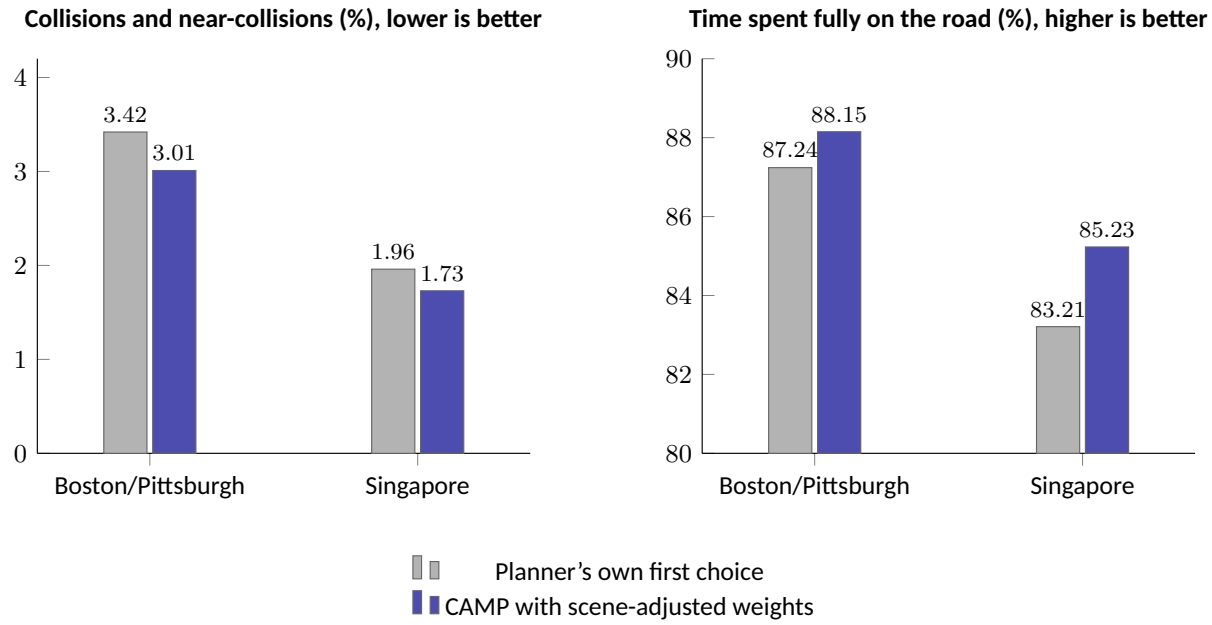

Figure 7: Stage 2 results. CAMP improves safety-related outcomes over the planner's default choice both in the training region (Boston and Pittsburgh, 24,246 test scenes) and in Singapore (32,328 scenes), a region never seen during training. The right chart's vertical axis starts at 80 percent to make the differences visible.

on 200,000 scenes instead of 50,000 improved human-likeness and comfort further, but slightly increased collision exposure in Boston and Pittsburgh, and in Singapore both CAMP variants slightly increased the rate of entering intersections on a red light relative to the planner's default. No single adjustment improved every measure at once. We consider this one of the project's useful findings: evaluations of automated driving should always report safety, rule compliance, comfort, and progress separately, because improvements in one can quietly purchase losses in another.

5.3 In the driver's seat: closed-loop results

Stage 3 is the sternest test, because early mistakes compound. Figure 8 summarizes it. Across all 480 closed-loop runs (four CAMP variants, 120 scenarios each) there was not a single collision. The variants trained on more data passed more scenarios (100 and 96 of 120, versus 98 and 95 for the smaller training sets), kept meaningfully more distance from the nearest other road user (about 0.3 to 0.4 meters more at the closest approach), and produced smoother motion on five of six standard comfort measures.

5.4 Limits of these findings

All results in this report come from simulation on recorded and synthetic scenarios; no public-road testing was performed. The safety checks and measures, while standard in the research field, are proxies for real-world safety, not guarantees. The floor result in Stage 1 shows that a choosing rule is only as good as the menu it is offered, so CAMP complements, and cannot replace, continued

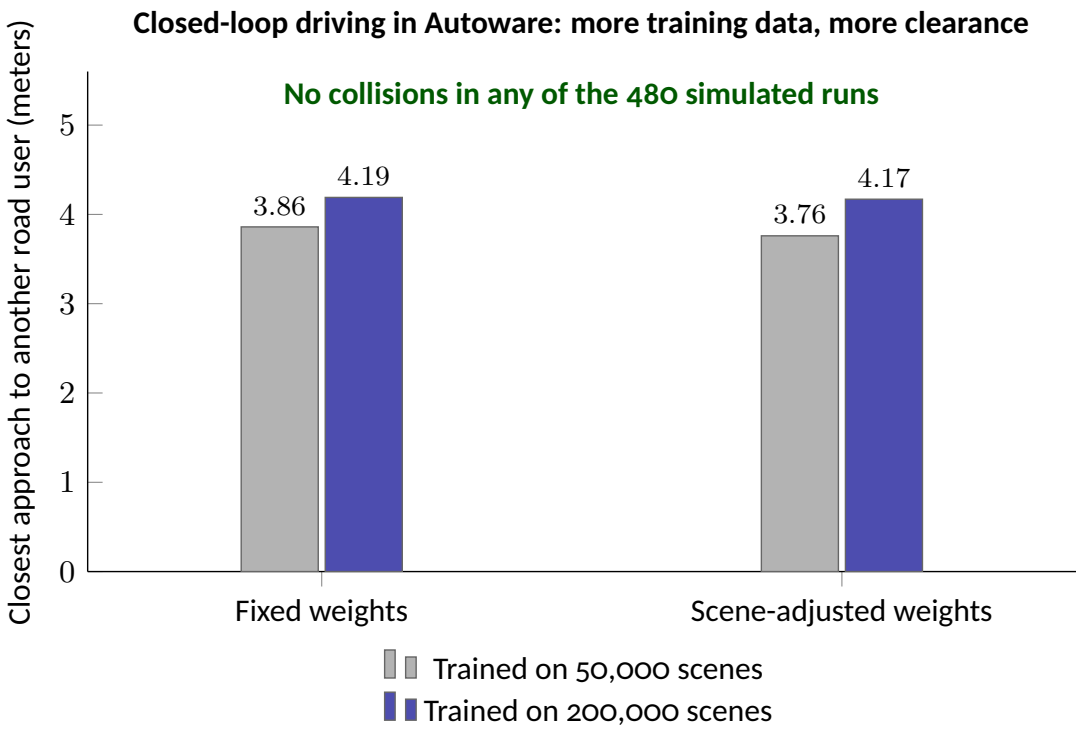

Figure 8: Stage 3 result: full closed-loop driving inside the open-source Autoware platform. More training data increased the margin kept from other road users, and no run ended in a collision.

improvement of the underlying planners. These are the honest boundaries within which the findings above hold.

6 What This Means: Conclusions and Recommendations

The project set out to make pretrained self-driving models adaptable across operational design domains without retraining them. The results support a clear conclusion: improving the final choice among a planner's existing candidate paths is a practical, inexpensive, and auditable way to improve safety-relevant behavior, and much of the improvement survives the move to a region the system has never seen. For the transportation community, we draw four recommendations.

1. **Evaluate the menu and the choice separately.** Our floor result shows that overall safety depends on two distinct questions: does the system propose at least one safe path (menu quality), and does it execute a safe path when one exists (choice quality)? An evaluation that reports only end-to-end outcomes cannot tell these apart, yet the remedies are completely different. Agencies and developers assessing automated driving in a new environment should ask for both numbers.

2. **Prefer auditable adaptation where possible.** A decision rule expressed as weights on explicit, named safety and comfort measures can be read, debated, constrained (safety before comfort, as we did), and revalidated cheaply after a change. Adaptation buried in retrained model weights offers none of that. Where an auditable mechanism can deliver the needed improvement, it is the better engineering and the better public policy.

3. **Report trade-offs, not single scores.** Every improvement we measured purchased something: caution cost a little progress, more training data slightly worsened one rule-compliance measure while improving comfort and human-likeness. Single-number safety scores hide such exchanges. Evaluation protocols should report safety, rule compliance, comfort, and progress side by side.

4. **Invest in the open-source stack.** Everything in this project runs on public datasets and open-source software, and its outputs return to that ecosystem. Open infrastructure is what lets a method developed in Pittsburgh be scrutinized in Washington and deployed in a smaller city or another country without a licensing gatekeeper. It is also what makes results like ours independently reproducible.

Natural next steps are integration of the selection layer as a reusable Autoware component, evaluation on physical test platforms, and extending the learning-from-drivers step to data from rural and infrastructure-limited road networks, where low-cost adaptation has the most to offer.

7 Outputs, Outcomes, and Technology Transfer

7.1 Publication

The project's methods, proofs, and complete experimental results are documented in: Xinchen Lin, Yidi Miao, and Peter Zhang, "CAMP: Decision-Layer Adaptation for Fixed Trajectory Generators," manuscript prepared for submission to the INFORMS Journal on Computing. No publication from the project had appeared as of the date of this report; the URL or DOI will be added to

the Safety21 project record (https://ppms.cit.cmu.edu/projects/detail/593) when available. Project research was presented at the 2025 INFORMS Annual Meeting in October 2025.

7.2 Open-source software

The project's central technology-transfer output is software for the open-source autonomous driving ecosystem, publicly available under the MIT license at

https://github.com/Fake-fate11/camp-core

As of the date of this report, the repository provides the CAMP implementation for both planners used in this project, including the selector that plugs into the Diffusion Planner's interface in the Autoware ecosystem; a compact set of trained deployment parameters (the eight-candidate configuration fitted on the 50,000-scene Boston and Pittsburgh population, supporting both the fixed-weight and scene-conditioned modes); the scripts used to build training records, train the selector, and evaluate it; interface documentation; and automated tests. Consistent with the licensing discussed in Section 8, the repository deliberately excludes the large benchmark datasets, third-party model weights, and bulky intermediate experiment artifacts, and instead documents how to obtain those from their providers. The pipeline runs against the public interfaces of Autoware Universe, the Diffusion Planner, and Scenario Simulator v2, so the selection layer can be used and inspected by anyone building on that stack. The repository will continue to evolve alongside the journal review process, so file-level details may change; the Safety21 project record will point to the current version, and remaining replication materials such as experiment configurations, seeds, and result files are planned to accompany the journal publication, following the project's data management plan.

7.3 Education and workforce development

Two Carnegie Mellon doctoral students, Xinchen Lin and Yidi Miao, conducted research on this project. They were trained in risk-aware optimization, learning from human demonstrations, and large-scale autonomous driving simulation, including closed-loop work with an industry-grade open-source driving stack.

7.4 Partnerships

The project maintained an academic partnership with Babacar Mbaye Ndiaye (Mathematics, Cheikh Anta Diop University, Dakar, Senegal), reflecting the project's goal of transferring automation safely to infrastructure-limited regions. The project's technology-transfer channel to industry is the open-source release described above, which any adopter of the Autoware ecosystem can use without a bilateral agreement.

8 Data Management and Availability

The research used two existing public benchmark corpora, nuScenes (https://www.nuscenes.org) and nuPlan (https://www.nuscenes.org/nuplan), both published by Motional with documentation and descriptive metadata at those addresses, and publicly distributed open-source models and simulation assets from the Autoware ecosystem (https://autoware.org), each under its own

license. Those licenses do not permit redistribution of the raw data or model weights, so the project's public code repository (Section 7.2) takes the standard approach of documenting how the records used in this research are rebuilt from the providers' assets rather than redistributing them. Artifacts produced by the project itself (experiment configurations, seeds, result files, and analysis scripts) are archived in accordance with the project's data management plan, with copies on Carnegie Mellon University storage and public release through the code repository and Carnegie Mellon's KiltHub repository alongside the journal publication. Dataset URLs and descriptive metadata will be posted to the Safety21 project record when the release is public. No personally identifiable information was collected.

9 Funding and Matching Support

This project was funded by the Safety21 National University Transportation Center under U.S. Department of Transportation grant 69A3552344811, with $98,047 in UTC funds awarded for the 2025–2026 project year. The required non-federal match was provided by Carnegie Mellon University's Heinz College of Information Systems and Public Policy as in-kind support valued at $98,047, consisting of academic-year instructional effort associated with the course Decision Making Under Uncertainty in Spring 2025 and Spring 2026. The total project budget from all sources was $196,094.

References

1. Salzmann, T., Ivanovic, B., Chakravarty, P., and Pavone, M. Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data. In *Computer Vision, ECCV 2020*.
2. Zheng, Y., Liang, R., Zheng, K., Zheng, J., Mao, L., Li, J., Gu, W., Ai, R., Li, S. E., Zhan, X., and Liu, J. Diffusion-based planning for autonomous driving with flexible guidance. In *The Thirteenth International Conference on Learning Representations*, 2025.
3. Caesar, H., Bankiti, V., Lang, A. H., Vora, S., Liong, V. E., Xu, Q., Krishnan, A., Pan, Y., Baldan, G., and Beijbom, O. nuScenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.
4. Caesar, H., Kabzan, J., Tan, K. S., Fong, W. K., Wolff, E., Lang, A. H., Fletcher, L., Beijbom, O., and Omari, S. nuPlan: A closed-loop ML-based planning benchmark for autonomous vehicles. In *CVPR Workshop on Autonomous Driving*, 2021.
5. Ivanovic, B., Harrison, J., and Pavone, M. Expanding the deployment envelope of behavior prediction via adaptive meta-learning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2023.
6. Rockafellar, R. T., and Uryasev, S. Optimization of conditional value-at-risk. *The Journal of Risk*, 2(3):21–41, 2000.
7. Benders, J. F. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.
8. The Autoware Foundation. Autoware: open-source software for autonomous driving. `https://autoware.org` (accessed August 2026).

9. Scenario Simulator v2: scenario testing framework for Autoware. Documentation. https://tier4.github.io/scenario_simulator_v2-docs/ (accessed August 2026).

10. Lin, X., Miao, Y., and Zhang, P. CAMP Core: decision-layer adaptation for frozen trajectory generators. Software repository, MIT License. https://github.com/Fake-fate11/camp-core (accessed August 2026).

A Technical Summary for Specialist Readers

This appendix compresses the method for readers with an optimization or machine learning background; full details and proofs are in the manuscript listed in Section 7. For each scene, a frozen pretrained generator supplies a scene representation and a finite ordered pool of candidate trajectories. Each candidate is described by a vector of normalized convex cost attributes (collision exposure, clearance and time-to-collision deficits, road and lane departure, speed and signal violations, progress, acceleration, yaw, jerk, and previous-plan continuity). The selector scores candidates by a weighted sum of attributes; weights lie on the probability simplex and are either a learned constant or an affine function of the generator's scene embedding. Supervision comes from demonstration calibration: a convex preference fit over logged human trajectories, with ordered group constraints (safety at least comfort at least other), identifies a within-pool target candidate per scene, and targets are frozen before selector training. Training minimizes the conditional value at risk (at level 0.9) of a structured hinge loss that ranks the target ahead of all other candidates, plus quadratic regularization. Because scores are affine in the trainable parameters and pools are finite, the problem is a convex piecewise-linear-quadratic program, solved by delayed constraint generation with exact full-pool separation (finite-cut Benders decomposition), which scales to fits on 200,000 scenes. Figure 9 reproduces the project's working diagram of the training loop from the earlier Trajectron++ instantiation.

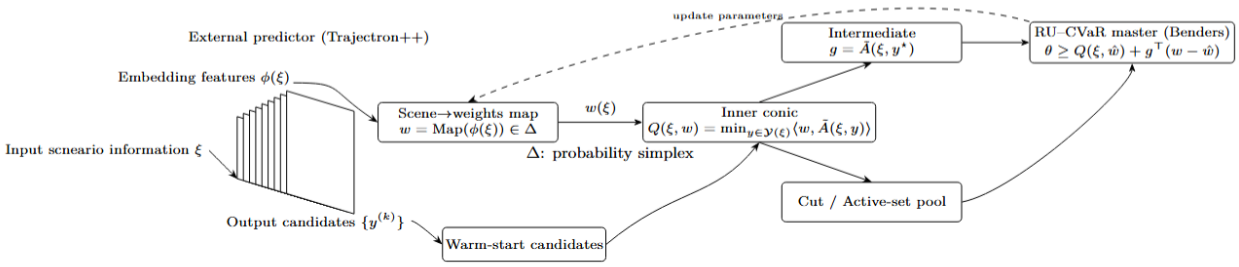

Figure 9: Working diagram of the CAMP training loop (Trajectron++ instantiation): a frozen predictor supplies embeddings and candidates, an inner conic problem generates Benders cuts, and a risk-averse (CVaR) master problem updates the scene-to-weights map.

B The Two-Stage Formulation, Its Non-Convexity, and the Convex Envelope Bound

This appendix, also for specialist readers, develops the optimization structure behind Section 3.2: why the natural training problem is a two-stage program, why it is inherently non-convex, and how a convex envelope yields a tractable and safety-aligned bound.

B.1 The two-stage structure

Candidate selection has the shape of a two-stage stochastic program. The first-stage (here-and-now) decision is made before the driving scenes are seen: choose the selector parameters, either a single weight vector $w \in \Delta_{R-1} = \{w \in \mathbb{R}^R_{\geq 0} : \mathbf{1}^\top w = 1\}$ over the R cost atoms, or a policy $w(\xi) = \Theta\,\phi(\xi)$ mapping scene features to weights. The second-stage (recourse) decision is made per scene ξ_i, after the frozen generator has revealed its candidate pool with atom vectors $A_{ik} \in \mathbb{R}^R_{\geq 0}$, $k = 1, \ldots, K$: execute the candidate with the best score. The realized second-stage value is

$$Q_{\min}(\xi_i, w) \;=\; \min_{k \leq K} \langle w, A_{ik} \rangle, \tag{1}$$

and the natural (deployed-cost) training problem is

$$\min_{w \in \Delta_{R-1}} \; F_{\min}(w) \;=\; \frac{1}{N} \sum_{i=1}^{N} Q_{\min}(\xi_i, w), \tag{2}$$

optionally with the empirical average replaced by a risk measure such as CVaR_α over scenes.

B.2 Why the problem is inherently non-convex

Each map $w \mapsto \langle w, A_{ik} \rangle$ is linear, so $Q_{\min}(\xi_i, \cdot)$, a pointwise minimum of linear functions, is *concave* and piecewise linear. The objective $F_{\min}$ is therefore concave, and problem (2) is a concave minimization over a polytope. This is non-convex in the way that matters most: a concave function attains its minimum over a polytope at an extreme point, so every vertex of the simplex is a candidate local optimum, and there can be up to R distinct local minima with no gradient signal connecting them. Worse, the vertex solutions are exactly the degenerate ones: all weight on a single atom, a selector that judges trajectories by, say, jerk alone. The non-convexity here is not a numerical nuisance; the global optimum of the self-scored objective is itself an undesirable, brittle object. (The objective is self-scored because the weighting being optimized also defines the cost being reported; driving the score down is not the same as choosing well.) Appendix C.2 visualizes both pathologies on synthetic data.

B.3 The convex envelope bound

Define the upper envelope over the same finite pool,

$$Q_{\max}(\xi_i, w) \;=\; \max_{k \leq K} \langle w, A_{ik} \rangle, \qquad F_{\max}(w) \;=\; \frac{1}{N} \sum_{i=1}^{N} Q_{\max}(\xi_i, w). \tag{3}$$

As a pointwise maximum of linear functions, $Q_{\max}(\xi_i, \cdot)$ is convex and piecewise linear, and by construction

$$Q_{\min}(\xi_i, w) \;\leq\; Q_{\max}(\xi_i, w) \quad \text{for every } w, \tag{4}$$

so $F_{\max}$ is a convex upper bound on the deployed cost $F_{\min}$. Minimizing $F_{\max}$ instead of $F_{\min}$ has three virtues. First, tractability: the epigraph formulation $\theta_i \geq \langle w, A_{ik} \rangle$ for all k is a linear program, and with a CVaR objective it remains a convex program by the Rockafellar–Uryasev construction. Second, a certificate: for the solution $\hat{w}$, the deployed cost satisfies $F_{\min}(\hat{w}) \leq F_{\max}(\hat{w})$, a computable guarantee that no candidate the selector could be forced to take costs more than the optimized bound; controlling the worst candidate in the pool is itself safety-aligned. Third,

non-degeneracy: minimizing a convex function over the simplex favors balanced interior weightings rather than vertices.

The price is the *envelope gap* $F_{\max}(w) - F_{\min}(w) \geq 0$, the within-pool spread of scores. When candidates in a pool score similarly under w, the gap is small and the surrogate optimum is provably near-optimal for the deployed cost; when pools contain strongly conflicting candidates, the surrogate is conservative. The gap is computable from data at any w, so it serves as a practical diagnostic (Appendix C uses it this way).

B.4 Solution by cutting planes, and the role of demonstrations

At scale, the epigraph of $F_{\max}$ has $N \times K$ constraints, most of them inactive. Benders-style delayed constraint generation exploits this: at the current iterate $\hat{w}$, the separation oracle $k_i^{\max} \in \arg\max_k \langle \hat{w}, A_{ik} \rangle$ identifies, per scene, the single violated cut $\theta_i \geq \langle w, A_{ik_i^{\max}} \rangle$, which by Danskin's theorem is an exact supporting hyperplane of $Q_{\max}(\xi_i, \cdot)$. Because the cut family is finite, the procedure terminates finitely at the optimum of the full program. This is the mechanism that made fits on 200,000 scenes practical in this project.

One clarification connects this appendix to the method actually deployed. The envelope surrogate needs no labels, but for the same reason it carries no information about which trade-off among atoms is *desirable*; it only controls the worst case. CAMP therefore combines the convex machinery above with supervision from human demonstrations (Section 3.1): the demonstration-calibrated targets pin down the preference direction, a convex hinge loss ranks each target ahead of its pool, and the same finite-cut solver applies. The synthetic experiments in Appendix C quantify exactly this division of labor.

C Synthetic Experiments: When Does Convexification Work?

To examine when the convex envelope of Appendix B is a good surrogate and when it is not, we ran controlled experiments on synthetic data designed for this report. They use no driving data; their purpose is to isolate the optimization phenomena, which is difficult in the full driving pipeline where many factors move at once. So that the experiments can be reproduced from this document alone, the complete program that generated Figures 10 and 11, with its fixed random seeds, is listed in Section C.6.

C.1 Setup

Each synthetic scene i has $K = 8$ candidates scored on $R = 3$ nonnegative atoms, so that the weight simplex can be searched exhaustively on a dense grid (step 0.01, about 5,000 weightings) and global optima of the non-convex objective are known. Candidates are drawn as

$$A_{ik} \;=\; c_i \;+\; s\left[(1-\lambda)\, B_{ik} \;+\; \lambda\, C_{ik}\right] \;+\; \text{noise}, \tag{5}$$

where c_i is a scene-level base cost, s scales candidate dispersion, and $\lambda \in [0, 1]$ mixes two structural regimes. In the *dominance* regime B ($\lambda = 0$), each candidate has a quality level applied to all atoms alike, so some candidates are simply better and the same one wins under almost any weighting. In the *trade-off* regime C ($\lambda = 1$), each candidate is cheap on one atom and expensive on the others, so which candidate wins depends entirely on the weighting. Real candidate pools mix the two. We use 600 training and 600 held-out test scenes per run and average over 10 random seeds.

C.2 The geometry of the problem, visualized

Figure 10 plots the two objectives of Appendix B along one edge of the weight simplex for a mixed instance ($\lambda = 0.6$, $s = 1.0$). The deployed objective $F_{\min}$ is concave, with local minima at both endpoints, which are single-atom weightings; its value at the ends (1.86) beats the balanced midpoint (2.09) precisely because a degenerate weighting can always find some candidate that looks cheap on one atom. The surrogate $F_{\max}$ is convex with a well-behaved interior minimum. The shaded envelope gap (0.56 cost units at the midpoint, about 27 percent of the deployed cost) is the price of convexity in this instance. Across the ten-seed sweep, direct minimization of the deployed objective landed on a simplex vertex, meaning more than 95 percent of the weight on a single atom, in 110 of 110 runs. Non-convexity here is not a solver inconvenience: the global optimum of the self-scored problem is always the degenerate solution.

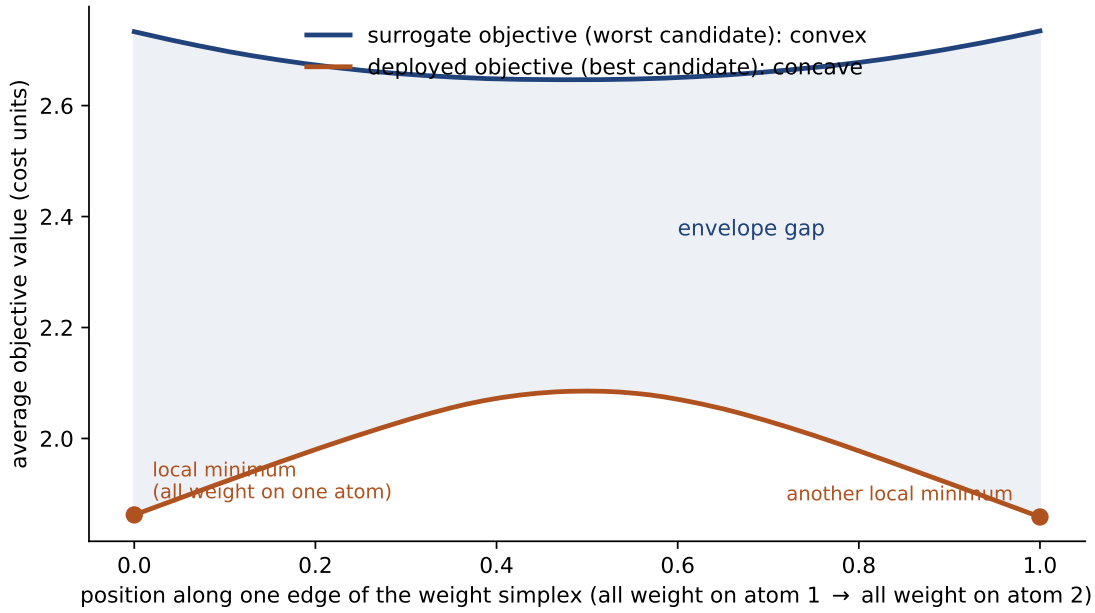

Figure 10: The two objectives along one edge of the weight simplex (synthetic data, $\lambda = 0.6$, $s = 1.0$). Minimizing the deployed objective directly is concave minimization: its minima sit at degenerate single-atom weightings. The convex surrogate has a balanced interior minimum and upper bounds the deployed cost everywhere; the shaded region is the envelope gap.

C.3 Judging selection quality fairly

Comparing training objectives requires a yardstick outside both of them, because the self-scored deployed cost rewards exactly the degeneracy just described. We therefore draw, per run, a hidden true preference $w^\star$ (from a Dirichlet distribution over the simplex), use it only for evaluation, and score any learned weighting w by the true cost of the candidates it actually selects on held-out scenes: $J(w) = \frac{1}{N}\sum_i \langle w^\star, A_{i\hat{k}_i(w)}\rangle$ with $\hat{k}_i(w) = \arg\min_k \langle w, A_{ik}\rangle$. We report regret normalized between the best and worst achievable J on the grid, so 0 percent means selecting as well as the best weighting in the family and 100 percent means the worst.

C.4 Results

Figure 11 shows the sweep over the conflict level λ at dispersion $s = 1.0$. Three findings stand out.

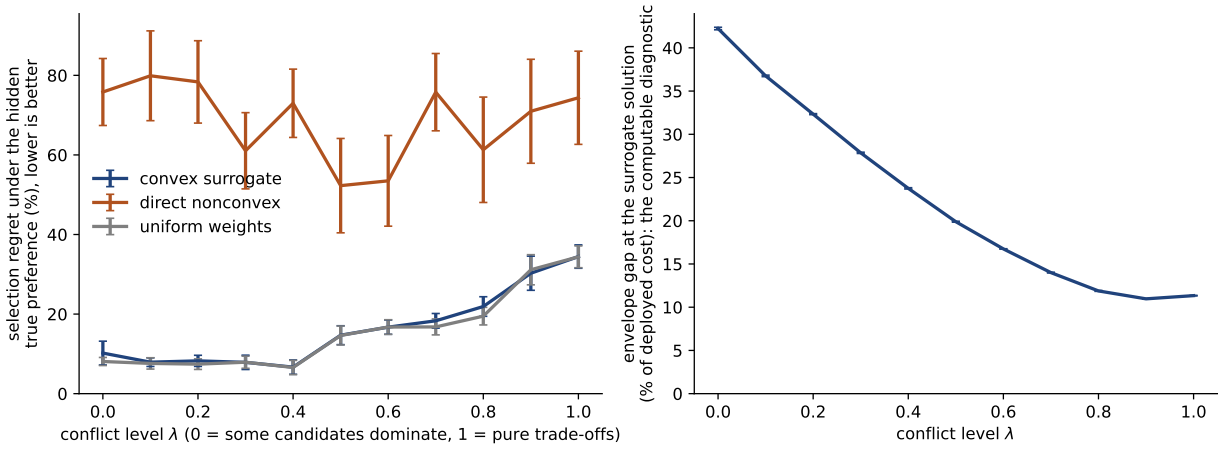

Figure 11: Synthetic sweep over the conflict level λ ($s = 1.0$, ten seeds, error bars show standard errors). Left: selection regret under the hidden true preference for three training policies. Right: the computable envelope-gap diagnostic at the surrogate solution.

Direct non-convex training fails everywhere. Minimizing the deployed objective itself, even solved to global optimality by exhaustive grid search, produces 52 to 76 percent regret across all conflict levels, far worse than either alternative. Its vertex solutions judge candidates by a single arbitrary atom. This is the strongest argument for convexification, and it has nothing to do with computational difficulty: even a perfect solver for the non-convex problem yields a bad selector, because the problem it solves is the wrong one.

The convex surrogate is sound but preference-blind. The surrogate's regret is low (10.2 percent) when pools have dominance structure ($\lambda = 0$): almost any sensible weighting picks the dominant candidate, and the surrogate is a sensible weighting. As trade-offs strengthen, its regret grows to 34.5 percent at $\lambda = 1$, and it tracks the uninformed uniform-weights baseline (8.1 to 34.4 percent) almost exactly. This is the honest boundary of convexification-without-labels: the envelope controls the worst case and avoids degeneracy, but it contains no information about which trade-off the true preference favors. When the pool forces genuine trade-offs, that information must come from somewhere else, and in CAMP it does: the demonstration-calibrated targets of Section 3.1 supply it, at which point training is convex *and* preference-informed, with no envelope needed.

The gap diagnostic measures certification, not regret. The envelope gap at the surrogate solution (right panel) shrinks from 42 percent of deployed cost at $\lambda = 0$ to 11 percent at $\lambda = 1$: in high-conflict pools the candidates' totals are similar, so the bound (4) is tight and the surrogate value is an accurate certificate of deployed cost, even while preference regret is at its largest. The two notions of "working well," certifying the deployed cost and matching the true preference, are distinct, and only the first is measurable without labels. Practitioners should read a small gap as a tight cost guarantee, not as evidence that the learned trade-off is the desired one.

C.5 Scope of these experiments

These experiments are illustrative and deliberately small: three atoms, linear scoring, a stylized generator, and a fixed (not scene-conditioned) selector. They are not evidence about driving performance; the driving evidence is in Sections 5 and in the accompanying manuscript. Their role is to make three structural facts concrete: minimizing the self-scored deployed objective is degenerate by construction, the convex envelope repairs the degeneracy and certifies a cost bound, and demonstration supervision, not convexification, is what supplies preference information when candidate pools embody real trade-offs.

C.6 Complete code for these experiments

The following Python program (using only the NumPy and Matplotlib libraries) generated every number and figure in this appendix. All random seeds are fixed in the code, so running it reproduces the results exactly.

```
#!/usr/bin/env python3
"""Synthetic experiments for Appendix C: when does the convex envelope
surrogate for candidate selection work well, and when not?

Setting (fixed-weight selector, R=3 atoms so the simplex can be searched
globally by a dense grid):
  scene i has K candidates with atom-cost vectors A_ik in R^3 (nonneg).
  deployed objective  Fmin(w) = mean_i min_k <w, A_ik>   (concave in w)
  surrogate objective Fmax(w) = mean_i max_k <w, A_ik>   (convex in w)
Data generator with two structural regimes mixed by lambda:
  benign/dominance: candidate k has level t_k, cost ~ t_k on every atom
     -> the same candidate is best under (almost) any weighting
  trade-off/conflict: candidate k is cheap on one atom, expensive on the
     others -> which candidate is best depends strongly on the weighting
  A_ik = c_i + s * [ (1-lam)*benign_ik + lam*conflict_ik ] + noise, >= 0.
"""
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

rng_global = np.random.default_rng(0)
R, K = 3, 8

def make_scenes(N, lam, s, rng):
    c = 1.5 + 0.5 * rng.random((N, 1, 1))                  # scene base cost level
    # benign/dominance component: per-candidate quality level on all atoms
    t = rng.random((N, K, 1))                              # 0 (good) .. 1 (bad)
    benign = np.repeat(t, R, axis=2)
    # conflict component: candidate k cheap on atom m(k), expensive elsewhere
    m = rng.integers(0, R, size=(N, K))
    conflict = np.ones((N, K, R))
    for r in range(R):
        conflict[..., r] = np.where(m == r, 0.0, 1.0)
    conflict += 0.15 * rng.standard_normal((N, K, R))
    A = c + s * ((1 - lam) * benign + lam * conflict)
    A += 0.05 * rng.standard_normal(A.shape)
    return np.clip(A, 0.0, None)

def simplex_grid(step=0.01):
```

```
    pts = []
    n = int(round(1 / step))
    for a in range(n + 1):
        for b in range(n + 1 - a):
            pts.append((a * step, b * step, 1 - (a + b) * step))
    return np.array(pts)                                    # (G,3)

W = simplex_grid(0.01)                                      # G ~ 5151

def objectives(A, W, chunk=800):
    """Return Fmin(w), Fmax(w) over the grid for scene tensor A (N,K,R)."""
    fmin = np.empty(len(W)); fmax = np.empty(len(W))
    for lo in range(0, len(W), chunk):
        w = W[lo:lo + chunk]                                # (g,3)
        S = np.einsum("nkr,gr->nkg", A, w)                  # (N,K,g)
        fmin[lo:lo + len(w)] = S.min(axis=1).mean(axis=0)
        fmax[lo:lo + len(w)] = S.max(axis=1).mean(axis=0)
    return fmin, fmax

# ---------------------------------------------------------------------
# Figure C1: 1-D slice showing concavity of Fmin vs convexity of Fmax
# ---------------------------------------------------------------------
rng = np.random.default_rng(7)
A = make_scenes(600, lam=0.6, s=1.0, rng=rng)
ts = np.linspace(0, 1, 201)
Wedge = np.stack([1 - ts, ts, np.zeros_like(ts)], axis=1)  # edge e1 -> e2
fmin_e, fmax_e = objectives(A, Wedge)

fig, ax = plt.subplots(figsize=(7.2, 4.2))
ax.plot(ts, fmax_e, color="#20437c", lw=2.2,
        label="surrogate objective (worst candidate): convex")
ax.plot(ts, fmin_e, color="#b0521f", lw=2.2,
        label="deployed objective (best candidate): concave")
ax.fill_between(ts, fmin_e, fmax_e, color="#20437c", alpha=0.08)
imid = len(ts) // 2
ax.annotate("envelope gap", xy=(ts[imid], (fmin_e[imid] + fmax_e[imid]) / 2),
            xytext=(0.6, (fmin_e[imid] + fmax_e[imid]) / 2), fontsize=10,
            color="#20437c")
for t0 in (0.0, 1.0):
    ax.plot([t0], [fmin_e[0 if t0 == 0 else -1]], "o", color="#b0521f", ms=7)
ax.text(0.02, fmin_e[0] + 0.03, "local minimum\n(all weight on one atom)",
        fontsize=9, color="#b0521f")
ax.text(0.72, fmin_e[-1] + 0.03, "another local minimum", fontsize=9,
        color="#b0521f")
ax.set_xlabel("position along one edge of the weight simplex "
              "(all weight on atom 1 $\\rightarrow$ all weight on atom 2)")
ax.set_ylabel("average objective value (cost units)")
ax.legend(frameon=False, loc="upper center")
ax.spines[["top", "right"]].set_visible(False)
fig.tight_layout()
fig.savefig("fig_slice.pdf")
plt.close(fig)

# ---------------------------------------------------------------------
# Figure C2: sweep of conflict level, judged by a hidden true preference
# ---------------------------------------------------------------------
# The self-scored objectives cannot judge themselves: minimizing Fmin over w
# is degenerate (any weighting makes its own choices look cheap). We therefore
# draw a hidden true preference w_true, used ONLY to evaluate the candidate
```

```
# that each learned weighting actually selects:
#   J(w) = mean_i <w_true, A_{i, argmin_k <w, A_ik>}>
def selection_quality(A, W, w_true, chunk=800):
    """True cost of the candidate selected by each grid weighting. (G,)"""
    true_cost = np.einsum("nkr,r->nk", A, w_true)          # (N,K)
    out = np.empty(len(W))
    for lo in range(0, len(W), chunk):
        w = W[lo:lo + chunk]
        S = np.einsum("nkr,gr->nkg", A, w)                 # (N,K,g)
        pick = S.argmin(axis=1)                            # (N,g)
        out[lo:lo + len(w)] = np.take_along_axis(
            true_cost[:, :, None].repeat(pick.shape[1], axis=2),
            pick[:, None, :], axis=1)[:, 0, :].mean(axis=0)
    return out

lams = np.linspace(0, 1, 11)
seeds = range(10)
S_DISP = 1.0
policies = ["convex surrogate", "direct nonconvex", "uniform weights"]
reg = np.zeros((3, len(lams), len(seeds)))
gapn = np.zeros((len(lams), len(seeds)))
vertex_hits = 0; runs = 0
for li, lam in enumerate(lams):
    for si, seed in enumerate(seeds):
        rng = np.random.default_rng(1000 * seed + li * 17)
        w_true = rng.dirichlet(np.full(R, 3.0))
        Atr = make_scenes(600, lam, S_DISP, rng)
        Ate = make_scenes(600, lam, S_DISP, rng)
        fmin_tr, fmax_tr = objectives(Atr, W)
        i_sur = np.argmin(fmax_tr)                         # convex surrogate
        i_deg = np.argmin(fmin_tr)                         # direct nonconvex
        vertex_hits += int(np.max(W[i_deg]) > 0.95); runs += 1
        J = selection_quality(Ate, W, w_true)
        i_uni = np.argmin(np.abs(W - 1 / R).sum(axis=1))   # uniform weights
        Jbest, Jworst = J.min(), J.max()
        for pi, idx in enumerate([i_sur, i_deg, i_uni]):
            reg[pi, li, si] = 100 * (J[idx] - Jbest) / (Jworst - Jbest)
        # normalized envelope gap at the surrogate solution (diagnostic)
        gapn[li, si] = ((fmax_tr[i_sur] - fmin_tr[i_sur]) / fmin_tr[i_sur])

fig, axes = plt.subplots(1, 2, figsize=(10.6, 4.0))
colors = {"convex surrogate": "#20437c", "direct nonconvex": "#b0521f",
          "uniform weights": "#7f7f7f"}
for pi, name in enumerate(policies):
    m = reg[pi].mean(axis=1); e = reg[pi].std(axis=1) / np.sqrt(len(seeds))
    axes[0].errorbar(lams, m, yerr=e, color=colors[name], lw=2, capsize=2.5,
                     label=name)
axes[0].set_xlabel("conflict level $\\lambda$ (0 = some candidates dominate, "
                   "1 = pure trade-offs)")
axes[0].set_ylabel("selection regret under the hidden\ntrue preference (%), "
                   "lower is better")
axes[0].set_ylim(bottom=0)
axes[0].legend(frameon=False)
m = gapn.mean(axis=1); e = gapn.std(axis=1) / np.sqrt(len(seeds))
axes[1].errorbar(lams, 100 * m, yerr=100 * e, color="#20437c", lw=2,
                 capsize=2.5)
axes[1].set_xlabel("conflict level $\\lambda$")
axes[1].set_ylabel("envelope gap at the surrogate solution\n(% of deployed "
                   "cost): the computable diagnostic")
```

```
axes[1].set_ylim(bottom=0)
for ax in axes:
    ax.spines[["top", "right"]].set_visible(False)
fig.tight_layout()
fig.savefig("fig_sweep.pdf")
plt.close(fig)

# ------------------------------ summary ------------------------------
print("=== slice experiment (lam=0.6, s=1.0) ===")
print(f"envelope gap at midpoint: {fmax_e[imid]-fmin_e[imid]:.3f} cost units")
print(f"Fmin at edge ends: {fmin_e[0]:.3f}, {fmin_e[-1]:.3f}; "
      f"Fmin at midpoint: {fmin_e[imid]:.3f}")
print("=== sweep (s=1.0) ===")
for pi, name in enumerate(policies):
    print(f"{name:18s} regret% lam=0: {reg[pi,0].mean():5.2f} | "
          f"lam=0.5: {reg[pi,5].mean():5.2f} | lam=1: {reg[pi,10].mean():5.2f}")
print(f"envelope gap%      lam=0: {100*gapn[0].mean():5.1f} | "
      f"lam=0.5: {100*gapn[5].mean():5.1f} | lam=1: {100*gapn[10].mean():5.1f}")
print(f"direct nonconvex optimum at a simplex vertex (max weight > 0.95): "
      f"{100*vertex_hits/runs:.1f}% of {runs} runs")
```