Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

When Roadmap Completion Pays Off: A Two-Trace Analysis of Multi-Query Robot Path Planning

Meshaal Mouawad

mm4922@msstate.edu

Mississippi State University

Research Article

Keywords:

Posted Date: November 20th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11790998/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

When Roadmap Completion Pays Off: A Two-Trace Analysis of Multi-Query Robot Path Planning

Meshaal Mouawad[1*]

[1]Department of Electrical and Computer Engineering, Mississippi State University, Mississippi State, MS 39762, USA.

Corresponding author(s). E-mail(s): mm4922@msstate.edu;

Abstract

A reusable robot roadmap stores candidate motions as a graph and retains collision-test results for edges shared across start–goal requests. For an observed request sequence and specified operation prices, we determine whether full validation of reusable edges reduces modeled work and which request boundary minimizes it. Two executions supply the comparison: on-demand (lazy) validation throughout and completion before the first request. Every completion date adds checks on persistent edges still untested by lazy planning at the final request. When service on the completed graph is independent of earlier requests, the second trace supplies search work and terminal checks on edges incident to the current start or goal. Direct execution verified reconstruction for 17,280 schedules on 216 forty-request workloads and two search engines. With persistent and terminal checks priced equally and positive prices evaluated separately for each workload, some completion beat permanent laziness on 151 of 216 restarted A* workloads and 183 of 216 query-local Lifelong Planning A* workloads. Delaying completion beat both upfront completion and permanent laziness on one of 216 and 40 of 216 workloads, respectively. The largest limiting gains over the better endpoint were 0.258% and 3.361% as collision-test price approached zero. Under permanent laziness, median paired timing ratios favored restarted A* over query-local repair on generated maps. Lazy terminal attachment gave lower median suffix time than eager attachment on all 60 matched triggered A* workloads. The criterion distinguishes losses that changing the purchase date could reverse from purchases that no date would repay.

Keywords: Robot path planning, Multi-query planning, Collision checking, Lazy search, Visibility graph, Amortization

Categories: (3), (5).
MSC2020: 68T40, 68R10.
ORCID: 0000-0002-1152-8324.

1 Introduction

Geometric path planning computes a collision-free route between a robot's initial and target configurations. A configuration specifies the position and any orientation or joint coordinates needed to describe the robot's placement. When successive start–goal requests share an unchanged environment, a planner can retain earlier collision-test results and decide whether to check the remaining reusable path segments in advance. For an observed query sequence and specified operation prices, this study establishes whether that investment would reduce total modeled work and which request boundary would minimize it. The criterion

is derived analytically and verified by directly executing every completion position on common query sequences.

Path planning operates within a navigation system: perception supplies information about the surroundings, localization estimates the robot’s configuration, and control translates the planned motion into commands [1, 2]. A *robotics roadmap* represents possible motion through a graph whose vertices are selected configurations and whose edges are local paths between them. A planning request, or *query*, specifies a start and a goal. Connecting these configurations to an existing roadmap allows graph search to identify a route through its feasible edges [3, 4]. The graph can serve successive queries, making the computational work invested in its construction and validation reusable. Figure 1 locates this representation within the navigation system and identifies the geometric roadmap family used here.

In the polygonal maps considered here, the roadmap is built from obstacle vertices and candidate straight-line connections. Candidates between fixed obstacle vertices are *persistent*: their geometry remains unchanged across queries. Candidates incident to the current start or goal are *terminal attachments*: they connect that request to the reusable graph. Collision validation determines whether a whole candidate segment avoids the forbidden obstacle region. A *lazy* planner first searches with unresolved candidates provisionally available, checks selected segments on the resulting route, and searches again when a collision invalidates that route [5, 6]. Persistent results accumulate as requests are served. *Completion* resolves every remaining persistent candidate as feasible or blocked, so subsequent searches begin with the persistent collision states already known.

Earlier research establishes choices about how much geometry to validate and how to reuse the resulting information. Song, Thomas, and Amato compared complete, partial, and deferred roadmap validation and measured both construction and query time [7]. Benhamouche and colleagues separate fixed obstacle visibility from changing start–goal connections, preparing the static part offline and accelerating later attachment [8]. Hartmann and colleagues prioritize earlier validation effort when solving new queries [9]. These contributions make validation depth and reuse explicit planning choices.

Ordered service costs also determine when to acquire reusable information. In database tuning, Bruno and Chaudhuri proved an optimal creation and removal schedule for an index, an auxiliary structure used to serve queries, on a known workload [10]. For an initially lazy roadmap, each request can resolve more persistent candidates before completion. The question is how to derive the additional validation cost and subsequent query-service costs from that evolving cache, so that every completion position can be compared with continued laziness. This comparison distinguishes a losing date from a purchase that no date would repay. Counting only the checks outstanding at a chosen trigger cannot make that distinction: some would eventually be performed by lazy planning as well, whereas others would remain unchecked when the sequence ends.

The proposed analysis reconstructs that comparison from two reference executions, termed *traces* because they record the operations performed for each successive query. One trace remains lazy throughout. The other completes the persistent graph before the first query and retains lazy validation of terminal attachments. Every policy that eventually completes tests the same persistent candidate set; its excess persistent checking over the lazy reference is therefore determined by the candidates that reference still leaves unchecked at the final horizon. When completed-graph query service depends on the current query and graph, independently of earlier requests, the second trace also supplies the terminal and search work after any completion position. Combining the shared lazy prefix with that completed suffix gives an exact work comparison. The order of those savings determines the best finite completion date. Completion pays if the savings from that date through the horizon exceed the cost of the extra persistent checks.

Three connected questions organize the study: how can two reference traces recover the work of every completion point; which unused persistent checks and subsequent service savings determine whether completion pays; and how do terminal validation after a common completion event, search implementation, and retention of search state affect measured execution? The first two questions concern the analytical decision and its direct computational verification. The third connects that decision to paired implementation

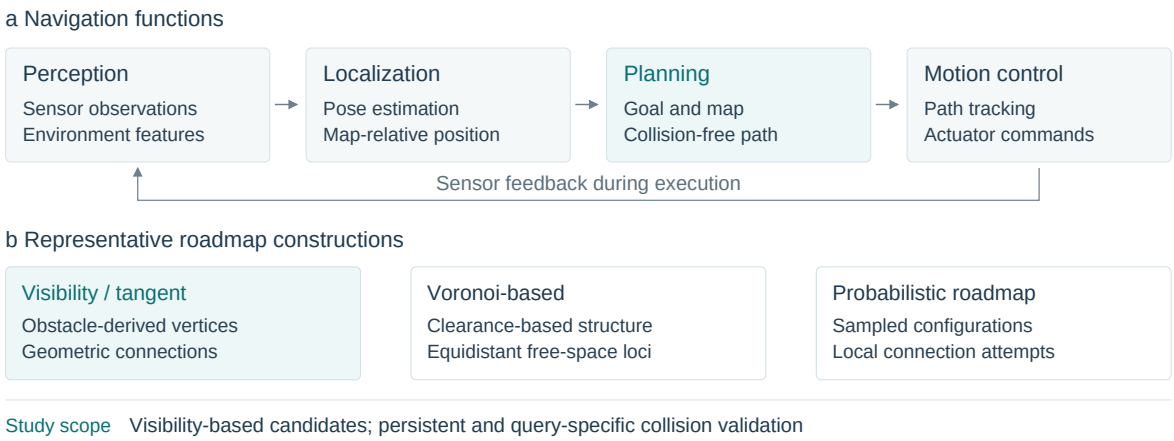


Fig. 1 Navigation functions and representative roadmap constructions. Perception and localization supply environmental and robot-state information; planning determines a route; control produces motion commands [1, 2]. Visibility graphs use obstacle-derived vertices and straight connections, Voronoi roadmaps follow clearance structure, and probabilistic roadmaps connect sampled configurations [3, 4]. The visibility/tangent branch supplies the candidate graphs evaluated here. This figure is a conceptual organization of these functions. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

comparisons: terminal attachment after the same completion, restarted A* versus within-query incremental repair, and retention versus resetting of labels within the incremental engine. The search comparisons keep collision validation permanently lazy.

The evidence combines the proof with 17,280 directly executed completion schedules on 216 forty-query workloads and two search engines. Exact cost comparisons distinguish an unfavorable completion date from a workload on which no date is favorable. They also identify a strict delay advantage over both upfront completion and permanent laziness on one of 216 restarted A* workloads and 40 of 216 query-local Lifelong Planning A* workloads, with favorable positive collision-test prices evaluated separately for each workload. Complementary timing measurements across fourteen planning modes compare search implementations and attachment. Replays from identical reconstructed checkpoints isolate the attachment policy. Together, these results establish a finite-sequence reference for assessing the decision to acquire complete persistent collision information, the additional value of selecting its acquisition time, and the recurring work required to use it.

2 Geometric planning and reusable roadmap information

A robot can reuse what it has learned about a fixed obstacle map even as its start and goal change. A collision check on a segment between two obstacle vertices remains valid for the next request. A distance computed from the previous start may need repair. The distinction matters for completion because checking every remaining persistent edge settles its feasibility, while each request still requires a shortest-path search.

2.1 Configuration space and the geometric path objective

A geometric planner specifies robot placement by a configuration, expressed through generalized coordinates such as position, orientation, or joint values. The *configuration space* is the set of all such configurations. Collision constraints identify the forbidden configurations, and the remaining admissible configurations form free space [11]. For planar translation with fixed orientation, two position coordinates suffice. A finite robot can be represented by expanding physical obstacles according to its footprint and planning the motion of a reference point around the expanded obstacles. This

transformation makes the geometry of the robot part of the feasibility condition rather than an adjustment performed after route selection.

The path objective determines how feasible alternatives are compared. Here, each route consists of consecutive straight segments through positions $p_0, \ldots, p_k$, with $p_0 = s$ and $p_k = t$ denoting the start and goal. The objective is minimum Euclidean length:

$$L(p_0, \ldots, p_k) = \sum_{i=0}^{k-1} \|p_{i+1} - p_i\|_2. \qquad (1)$$

Here $\|\cdot\|_2$ denotes the Euclidean norm. Each segment must lie in the admissible geometric region. Figure 2 shows two feasible routes on constructed geometry: both contain three segments, yet their total lengths differ. Consequently, minimizing the number of graph edges traversed does not in general minimize Eq. (1). The computation required to identify the minimum-length route is measured separately from the distance that route represents. This separates *path quality*, the geometric objective, from *planning cost*, the computational quantity investigated here.

For the fixed-orientation square model, an axis-aligned rectangular obstacle is expanded by the square's half-side in each coordinate direction. A point path for the square's centre in the transformed space then represents a collision-free placement of the body, as shown in Figure 3. The study treats strict obstacle interiors as forbidden and permits boundary contact. Under that convention, contact between the reference path and an expanded boundary corresponds to contact between the square and the physical obstacle. The experimental footprint and coordinate precision are specified in Section 4.

2.2 Visibility and tangent connections as a path representation

A roadmap represents possible motions by vertices and candidate connections. In a geometric roadmap the obstacle structure supplies these connections, whereas a probabilistic roadmap connects sampled configurations [3]. In a cell decomposition, adjacency between regions of free space supplies a representation of connectivity [4]. The experiment uses the same geometric roadmap for all policies so that a difference in planning work can be traced to how its edges are validated and searched.

For point translation among polygonal obstacles, a shortest collision-free path can be represented by straight segments whose interior turning points belong to a subset of obstacle vertices [12]. The local argument explains why. A bend in open free space can be replaced by its chord, yielding a shorter feasible path unless the adjacent segments are collinear. A turn in the interior of a straight obstacle side can likewise be eliminated or moved to a boundary vertex. Connecting mutually visible obstacle vertices, together with the start and goal, therefore supplies a finite weighted graph for the polygonal shortest-path problem. Here, visibility is the geometric condition that a segment avoids forbidden interiors.

Geometric reduction uses this structure before full collision validation. Tangent-graph approaches retain connections that can support locally shortest routes around obstacle boundaries [13]. A path is locally taut when replacing a pair of consecutive segments by a shorter chord would intersect an obstacle. Local tangency can reject a proposed edge whose direction is unsuitable at an incident obstacle vertex; global collision validation must still test the surviving segment against intervening obstacles. Rohnert's preprocessing/query formulation and sparse visibility hierarchies exploit related geometric structure [14, 15]. Figure 2 makes the reduction explicit: the constructed full graph has 24 valid edges, of which 21 remain after the local tangency filter, while the shortest-path length remains 9.285 coordinate units. This is an edge-set comparison on fixed geometry. The experimental planners then share the same reduced candidate set, so changing a validation policy leaves their represented geometric alternatives unchanged.

2.3 Persistent geometry and query-specific attachment

The preprocessing/query distinction has a concrete form in an unchanged polygonal map. Let the persistent vertices be the obstacle vertices and the persistent candidates be the locally admissible segments between them. Their coordinates and geometric validity are invariant across start–goal queries. A new query adds candidates incident to

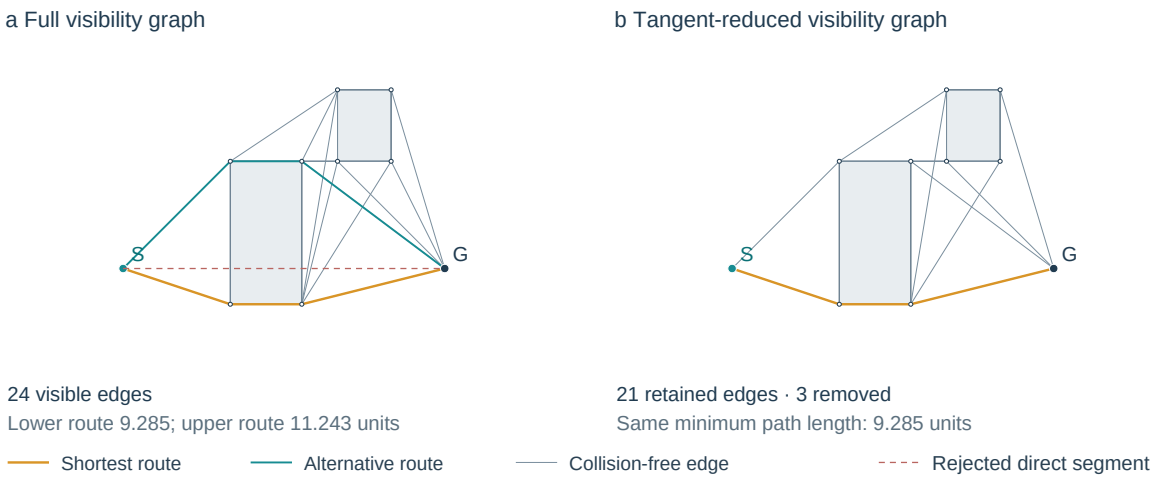


Fig. 2 Full and tangent-reduced visibility graphs on identical constructed geometry. (a) All 24 collision-free edges are shown, together with feasible lower and upper detours of 9.285 and 11.243 coordinate units. The red dashed direct segment is blocked and is not part of the graph. (b) Applying the local tangency filter removes three edges; the 21 retained edges preserve the 9.285-unit shortest path. Both detours use three segments, illustrating why edge count and Euclidean path length are different objectives. Boundary contact is permitted. These are explanatory edge-set reconstructions, not runtime measurements. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

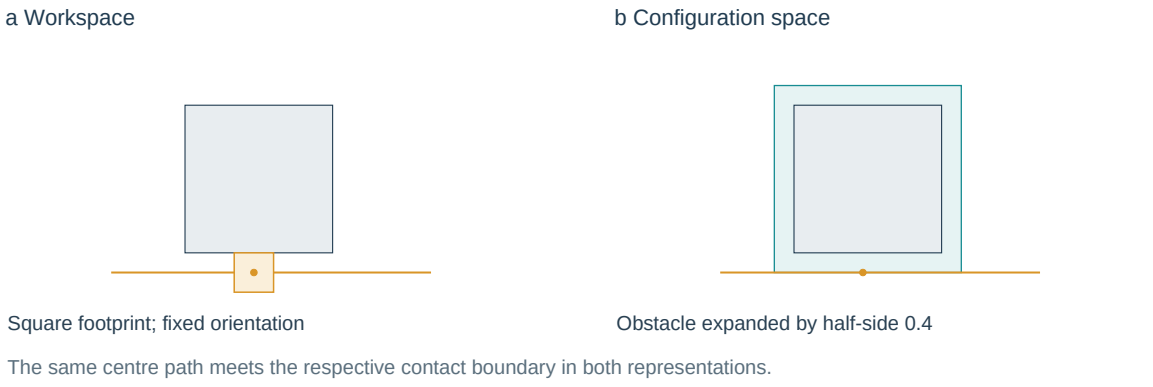


Fig. 3 Workspace and configuration-space representations of the same fixed-orientation square motion. (A) The physical footprint constrains how closely the centre can approach a rectangular obstacle. (B) Expanding the rectangle by the square's half-side transfers that constraint to a point representing the centre. The illustrative half-side is 0.4 coordinate units. The path touches the expanded boundary and the square touches the physical boundary, consistently with the stated contact convention. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

its start or goal, including the direct start–goal segment. These terminal candidates attach the query to the persistent graph. Their geometry changes when either endpoint changes, even though the obstacle model remains fixed.

Figure 4 shows two constructed queries on the same map. A checked segment between obstacle vertices keeps its collision result for both queries. Each set of terminals has its own connections to validate. Completing the persistent graph settles the status of every remaining obstacle-to-obstacle candidate, including blocked ones. The planner must still decide which connections to check for each new start and goal.

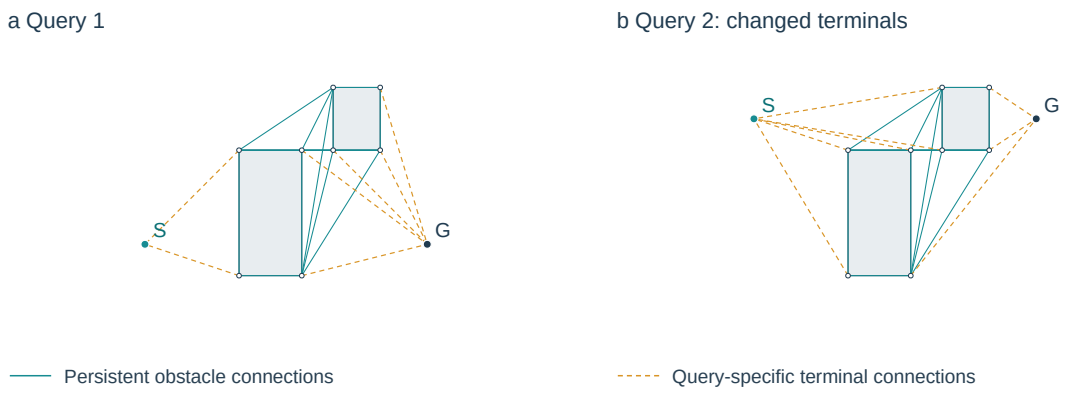


Fig. 4 Persistent roadmap edges and query-specific attachments on unchanged constructed geometry. Teal edges join fixed obstacle vertices. Amber edges connect the current start and goal to the graph and are replaced when the query changes. Dashed styling identifies the terminal role in this diagram; only geometrically visible attachments are drawn. Collision status and edge lifetime are distinct attributes, formalized in the validation-state diagram. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

2.4 Validation state and shortest-path state

Candidate generation specifies which motions may be considered; collision checking determines which of those motions are feasible. Each candidate has an *unknown*, *valid*, or *blocked* state. A valid edge retains its Euclidean length as its search cost; a blocked edge receives infinite cost. Figure 5 shows how a geometric predicate resolves unknown validity and how the resulting state is retained. For persistent edges, that state survives query replacement. For terminal edges, it belongs to the current start–goal instance.

Lazy search initially assigns unknown edges a lower bound on their validated cost and computes a provisional shortest path. With Euclidean candidate costs, a feasible edge keeps that cost and an infeasible edge becomes infinite. The planner checks selected provisional-path edges and repeats search after a discovered collision [5, 6]. When the provisional shortest path is fully validated, its cost equals its true cost, while every competing provisional path still lower-bounds its own true cost. It is therefore shortest in the candidate graph. The independent full-visibility reference used later checks that the candidate representation also preserves the intended geometric optimum on the evaluated queries.

Distance labels record the best known costs from the current start to graph vertices. Changing the start or blocking an edge can require distance updates even when the stored collision results remain valid. Incremental search selectively repairs affected labels [16]. Section 4 distinguishes engines that begin each request with fresh labels from one that also carries labels forward. This difference determines whether a completed graph and the current request fully specify the work of a new search.

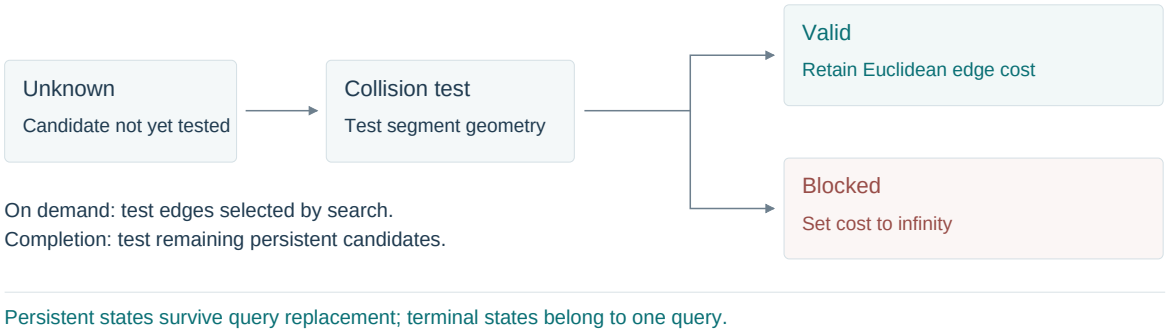


Fig. 5 Collision-validation state and retention lifetime. A geometric test resolves an unknown candidate as valid or blocked. Valid edges retain finite geometric costs; blocked edges are excluded from feasible solutions by infinite cost. Eager and lazy schedules determine when testing occurs. Persistent collision states remain available on the fixed map, whereas terminal states belong to the current query. The map is supplied in every state; unknown validity denotes a segment not yet tested. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

3 Related work on validation across planning requests

When a robot receives a new destination on an unchanged map, earlier work can help it in two ways. A roadmap can already encode useful routes through the obstacles, and the planner can already know which candidate motions are blocked. The first saves construction or search. The second lets the planner use or discard a motion without checking it again. Checking every remaining obstacle-to-obstacle connection would settle its status for all later requests. The computational return depends on which checks those requests would have needed and how much search they would have required.

3.1 Geometric preparation and reusable connections

Obstacle geometry determines which straight motions can belong to a collision-free route. Lozano-Pérez and Wesley used that geometry to construct feasible connections [12]. Lozano-Pérez represented an entire robot placement by a point in *configuration space*, where the forbidden region consists of placements in which the robot intersects an obstacle [11]. Once that region is fixed, the same obstacle features can support routes between different endpoints. Rohnert exploited this property by preprocessing convex polygonal obstacles for subsequent shortest-path queries [14]. Benhamouche and colleagues made the division explicit for visibility graphs. They preprocessed the subgraph joining fixed obstacle vertices and used the regions visible from the current endpoints, represented as visibility polygons, to determine their attachments [8]. Thus a new request can reuse the validity of an obstacle-to-obstacle segment even when it needs new connections to reach that segment.

How much geometry to prepare depends on the path objective. Liu and Arimoto showed that locally shortest routes can be represented by straight segments tangent to two obstacle boundaries and by convex boundary segments between contact points [13]. Oh and Leong exploited a related property on grids. A *taut* path turns tightly around obstacles at its bends, a condition they used to organize a sparse visibility graph [15]. Other exact representations group reachable points into intervals on grids or polygonal navigation meshes [17, 18]. Shen and colleagues combined a navigation mesh with precomputed path information for Euclidean queries [19]. A vehicle with kinematic constraints may instead require a smooth path that minimizes physical travel time, as in Jiang and colleagues' formulation [20]. For the geometric length objective studied here, all completion policies receive the same tangent-filtered candidates. A common candidate set also requires consistent decisions at obstacle boundaries. Shewchuk developed robust predicates for such geometric decisions [21]. Our experiments independently verify the stored geometry and reference path lengths.

A reusable graph can also grow with experience. Kavraki and colleagues connected sampled robot configurations during a preparation phase, then attached each start and goal to the resulting probabilistic roadmap [3]. Li and colleagues used the queries themselves to build the roadmap, adding selected parts of one request's search tree for use by later requests [22]. For polygonal robots that rotate as well as translate, Zhang, Ye, and Yu linked visibility graphs at discrete orientations to serve multiple queries [23]. The external map group in our evaluation uses obstacle layouts from their release. These constructions determine which connections are available for reuse. Given a fixed set of candidates, the next choice is when to learn which ones are feasible.

3.2 Selective collision validation and search effort

Bohlin and Kavraki let a search use an unchecked edge provisionally at its collision-free cost. This *optimistic* estimate allows the planner to find a candidate route and spend collision tests on its edges. Finding a blocked edge can require another search, while candidates off all provisional routes can remain unchecked [5]. Sánchez and Latombe used delayed checking in probabilistic planning for coordinated robot motion [24]. Song, Thomas, and Amato compared no, partial, and complete validation during roadmap construction, together with different orders of checking a candidate path. Their experiments showed that the preparation level changes both construction and query time; validating regions seldom used by queries can consume work without a corresponding return [7]. Their evaluation identifies the preparation–query tradeoff across validation levels. The question pursued here is a different one within that tradeoff: given the same candidate graph and an evolving cache of validated edges, what is the modeled cost of completing the remaining edges at each boundary in a finite request sequence? The principle of selective checking also applies when samples continue to accumulate. Karaman and Frazzoli proved that the cost of paths returned by their probabilistic roadmap variant, PRM*, converges almost surely to the optimum under their sampling and connection conditions [25]. Hauser's Lazy-PRM* defers collision tests until a candidate path could improve the current solution [26].

Selective checking can save geometric tests and increase the work of finding a valid path. Dellin and Srinivasa separated the search for a provisional shortest path from the choice of which edge to check. Their forward selector tests the first unknown edge along that path [6]. Mandalika and colleagues varied when search stops to perform a check and measured the resulting trade between evaluation and graph exploration [27]. For one source and target, Haghtalab and colleagues proved that an edge selector optimal for an individual instance exists and derived query-complexity bounds under a probabilistic model of edge validity [28]. Here the selector stays fixed across alternatives. The decision is whether to end selective checking of persistent edges at a boundary between requests.

3.3 Collision information and search-state lifetime

Earlier collision tests can guide where a planner spends its next unit of effort. In Effort Informed Roadmaps, Hartmann, Strub, Toussaint, and Gammell used a bidirectional search to favor paths whose edges had already been validated. They found initial solutions more quickly on their tested multi-query problems [9]. Hartmann, Ortiz-Haro, and Toussaint addressed a further issue in manipulation. Moving an object changes some collision relationships while leaving others intact. By checking those relationships separately, the planner could carry the unaffected results into later tasks [29]. Pasricha and Roncone reused simulated motions across queries while respecting the robot's dynamic constraints. Their planner, Lazy-BoE, defers full simulation and collision checking until a candidate trajectory requires them; experiments with a seven-joint arm measured time to a first feasible trajectory and subsequent solution quality [33]. These methods reuse selected motions or collision results as requests arrive.

Even a rejected sample can guide exploration. Zhang and colleagues used failed samples near a stalled search to locate regions for connecting narrow passages [34]. Zhao and colleagues used obstacle locations inferred from rejected samples to steer search-tree growth for a maintenance arm [35]. In a fixed roadmap, a blocked edge has a simpler continuing use. Every subsequent search can exclude it.

Table 1 Planning decisions and the information used to make or evaluate them.

Study	Information available	Decision or comparison
Song et al. [7]	Roadmap approximation level and candidate-path validation order	How much validation to perform during construction and querying.
Benhamouche et al. [8]	Preprocessed obstacle visibility and the current endpoints	How to attach each new query to the fixed graph.
Hartmann et al. [9]	Earlier validated paths in a reusable roadmap	Which paths to prioritize using retained validation effort.
Hartmann et al. [29]	Collision results stored for separate subproblems of manipulation tasks	Which unchanged collision relationships can be reused after objects move.
Mandalika et al. [27]	Partial search and edge evaluations within a request	When to interrupt search to check an edge.
Lim et al. [30]	Previous distance estimates and newly evaluated edges	Which search information to repair after a change.
Díaz Ramos et al. [31]	Neural-planner training cost and mean online saving per query	How many queries recover the training cost.
Guarda and Palli [32]	Topological-graph construction and repeated graph/grid search costs	When repeated graph queries recover construction cost.
Bruno and Chaudhuri [10]	Index creation cost and ordered service costs with and without an index	Which creation and removal schedule minimizes known-workload cost.
Present study	Persistent candidates left unchecked at the horizon and per-query service in the two reference runs	Whether full validation pays and which boundary minimizes modeled work on the recorded sequence.

The search calculation itself can also persist. Koenig, Likhachev, and Furcy stored distance estimates at graph vertices and updated affected estimates when edge costs changed. This *incremental repair* is the basis of Lifelong Planning A* (LPA*) [16]. Lim and colleagues combined repair with deferred edge evaluation for replanning, including problems with a moving start [30]. Huang and colleagues maintained a sampled graph under environmental changes and expensive collision checks [36]. This distinction between retaining edge validity and retaining distance estimates matters for comparing completion dates. Two runs that reach the same fully checked graph can arrive with different distance estimates. If each request initializes its own search state and uses the same deterministic rules, the completed graph and the current endpoints determine its terminal checks and search work. Its measured operation counts can then serve as the post-completion counts for that request whenever the graph was completed before serving it.

3.4 Amortization over an ordered query sequence

Preparation costs are *amortized* when the work saved over subsequent requests recovers the initial expenditure. Díaz Ramos, Arikan, and Baraniuk compared the cost of training a multigoal planner with its mean online saving per query [31]. Guarda and Palli compared building a topological graph and querying it repeatedly with repeated searches on a grid [32]. The quantity used to measure savings matters. In Dahlmann and colleagues' multi-trailer planner, road segments supplied as collision-free avoid static collision checks during search, while expansion of motion primitives and selection of search nodes still take time [37]. In Wittmann and colleagues' dual-arm experiment, planning each arm's path separately saved time at that stage. Coordinating the two paths made the complete procedure slower [38]. These results motivate counting both the collision tests a completed graph entails and the search work it changes.

Ordered savings also determine when to acquire a reusable computational structure. In database tuning, Bruno and Chaudhuri considered an index,

an auxiliary structure that changes the cost of serving a query. They accumulated the signed difference between service with and without that index and proved an optimal creation and removal schedule for a known workload [10]. The index model charges configuration transitions and obtains alternative query costs from optimizer information. For an irreversibly completed lazy roadmap, the missing quantities are different: the net additional persistent checks after accounting for those that later lazy queries would perform, and the per-query terminal-validation and search costs under both graph states. The two executed traces and the fixed-graph service condition in Section 5 supply these quantities. The minimum-prefix scheduling calculation then follows the established signed-savings principle.

For an initially unchecked roadmap, some of the checks purchased at completion would be performed anyway by later lazy searches. The additional persistent cost across the workload is therefore the cost of the candidates that permanent laziness still leaves unchecked at the final request. The return on those checks depends on the *order* of requests. A proposed completion can claim only the differences in terminal checks and search work from that point onward. The question is whether any such sum covers the additional checks and, if so, whether waiting improves on immediate completion. Table 1 relates this decision to its closest precedents.

Under history-independent service on the completed graph, one permanently lazy run and one initially completed run on the same ordered requests supply the counts for every possible completion point. Following the benchmark practice of comparing recorded runs on common instances [39], we test the reconstructed counts against direct executions at each point.

4 Shared planning model and search state

The completion comparison requires common candidate motions and an explicit account of the state used to serve each query. Every policy below receives the same map and ordered requests, constructs the same candidate edges, uses the same collision predicate, and minimizes Eq. (1). The search engines differ in which distance labels survive a collision discovery or a change of request. These dependencies determine the completed-query trace property used in Section 5. The completion controllers and terminal-validation controls are defined subsequently in Section 6.

4.1 Inputs, output, and shared representation

The input comprises a static collection of disjoint polygonal obstacles and an ordered sequence of start–goal queries. The output of each query is a collision-free polygonal path minimizing summed Euclidean edge length in the represented admissible region, or a failure status. Strict obstacle interiors are forbidden and boundary contact is permitted. An endpoint inside an obstacle produces an invalid-endpoint status. Coincident valid start and goal positions produce a zero-length route. These cases are handled explicitly rather than inferred from the normal search loop.

The persistent vertices are the obstacle vertices. Candidate generation tests the directions of the two boundary segments incident to each corner relative to a proposed line. The local tangent test rejects strict opposite-side configurations; connections that pass it still require a full collision check. This preliminary test reduces the candidates without substituting local tangency for global visibility. Each retained candidate has one stored state: unknown, valid, or blocked. Unknown and valid edges have Euclidean cost during provisional search; blocked edges have infinite cost. All implementations retain the same adjacency entries, which are the candidate connections stored in each vertex's neighbor list, including entries with infinite cost. This common storage representation prevents edge removal from becoming an additional intervention.

At a new request, the program removes the previous terminal connections, updates the start and goal positions, and inserts new terminal candidates. A terminal-to-corner candidate must pass the local tangent condition at the corner. The direct start–goal candidate is also considered. When a terminal coincides with an obstacle vertex, the zero-length connection between those positions is omitted; admissible nonzero terminal connections are still generated. Persistent collision results

remain stored; terminal results are local to the current request. This implements the two lifetimes illustrated in Figure 4.

For the square-robot settings, each axis-aligned physical rectangle is expanded as in Figure 3. The experimental half-side is the smaller of 0.005 times the map span and 0.2 times the minimum inter-obstacle gap. The expanded coordinates are rounded to nine decimal places and define the represented configuration-space geometry for every planner and the independent reference. Physical rectangles and the nominal half-side remain in the input record. The square keeps a fixed orientation along its route.

4.2 Restarted and incremental shortest-path search

A discovered collision changes an edge cost and can invalidate the current provisional path. The restarted engine resolves that change by running A* from the beginning [40]. It orders vertices by the sum of the current cost-to-come and a Euclidean lower bound on the remaining path length. The straight-line heuristic is admissible because every feasible route between the same endpoints is at least as long as their Euclidean separation. Persistent collision states survive each restart, while distance labels and the priority queue are rebuilt. This provides the reference against which selective repair of search state is measured.

The incremental engine uses LPA* to identify distances that need correction [16]. Let $g(v)$ be the stored distance label at vertex v, let $c(u, v)$ be the current search cost of an edge from a neighboring vertex u, and let $\mathrm{Pred}(v)$ be the neighbors from which v can be reached. The one-step value supported by those neighbors is

$$\begin{aligned} &\mathrm{rhs}(s) = 0, \\ &\mathrm{rhs}(v) = \min_{u \in \mathrm{Pred}(v)} \{g(u) + c(u, v)\}, \quad v \neq s. \end{aligned} \tag{2}$$

The minimum of an empty set is infinite. The one-step value evaluates the best arrival at v through its predecessors under the current edge costs. A vertex is locally inconsistent when $g(v) \neq \mathrm{rhs}(v)$. Invalidating an edge can introduce such inconsistencies; LPA* propagates the required corrections through its priority queue. The implementation uses optimized updates of neighboring vertices and stops when the goal is consistent and no better queue key remains.

Two incremental variants distinguish the lifetime of this search state. *Query-local LPA** initializes its labels for a new request, then repairs them after invalidated edges within that request. *Retained LPA** also carries labels into the next request. It updates vertices affected by removal and insertion of terminal edges, and recomputes the priority keys of inconsistent vertices when the goal changes. Both retain persistent collision results. Thus three different information policies are compared: restart the search, repair within a request, or repair across requests as well.

All engines use the same forward edge selector. It checks unknown edges of a provisional shortest route in order from start to goal, stopping at the first blocked edge. A collision is recorded and the search is restarted or repaired. A route is returned once its edges are valid; failure is returned if no provisional route remains. This shared rule follows the search/evaluation separation introduced by LazySP and generalized lazy search [6, 27]. It prevents different edge selectors from becoming an additional treatment in the engine comparison.

5 A two-trace criterion for whether and when to complete

Suppose the planner completes its roadmap immediately before a given query. Up to that boundary its execution agrees with a planner that remains lazy. From that boundary onward every persistent edge has a known status. If the completed graph and current query determine the service work, the remainder can be read from a second execution that completed the graph upfront. This section uses those two records to compare every completion date with permanent laziness.

5.1 Query service and persistent validation

Consider H queries on a fixed persistent candidate set E_P, initially unvalidated. In the permanently lazy execution, let $V_H \subseteq E_P$ be the persistent candidates tested by horizon H, and write $U_H = |E_P \setminus V_H|$. Thus U_H counts persistent candidates the permanently lazy execution leaves unchecked

through that horizon. A policy I_k follows that lazy execution through query $k-1$, completes every remaining persistent candidate before query k, and then continues with lazy terminal validation. The alternatives are $k = 1, \ldots, H$ and permanent laziness, denoted L.

For each query i, record terminal checks t_i and adjacency/repair visits a_i. A* counts examinations of stored neighbors; query-local LPA* additionally counts the explicit self visit in its repair loop. Superscripts L and F identify the permanently lazy trace and the fully completed trace. Define the *query-service saving*

$$d_i = c_T(t_i^L - t_i^F) + c_A(a_i^L - a_i^F), \tag{3}$$

where c_T and c_A are nonnegative work weights for a terminal check and an adjacency/repair visit. A positive d_i means that the completed graph reduces weighted service work on that query; a negative value means that it increases it. Persistent tests are accounted for separately with nonnegative weight c_P. The resulting work measure is

$$W = c_P P + c_T T + c_A A, \tag{4}$$

with P, T, and A denoting the corresponding accumulated operation counts. Each weight has units of work per named operation. This definition permits sensitivity analysis without assigning an unmeasured nanosecond value to a counter.

The required trace property is that the completed-graph service of a query is independent of earlier queries. It holds when the search engine initializes its distance labels and queue for each new query and uses the same deterministic selector, candidate generation, adjacency order, and tie handling. Once all persistent states are known, the current query and the common graph determine its terminal-check and adjacency/repair-visit counts. Restarted A* and query-local LPA* have this property in the present implementation. Figure 6 illustrates how the lazy prefix and completed suffix determine an intermediate policy.

5.2 Exact completion identity and decision criterion

Theorem 1 (Two-trace completion identity) *For the fixed-graph query-service model above, suppose persistent test results remain valid and each candidate is tested at most once. If completed-graph service is independent of the preceding queries, then for every* $1 \le k \le H$,

$$W(I_k, H) - W(L, H) = c_P U_H - \sum_{i=k}^{H} d_i. \tag{5}$$

Proof Every I_k tests all $|E_P|$ persistent candidates by H: the lazy prefix tests a subset, and completion tests its complement. Permanent laziness tests $|V_H|$ candidates. Their persistent-work difference is therefore $c_P(|E_P| - |V_H|) = c_P U_H$, independently of k. Their query-service operations agree before k. After completion, the history-independent service property makes the operations of I_k equal to the fully completed trace on every query $i \ge k$. Subtracting lazy service from completed service on the suffix gives $-\sum_{i=k}^{H} d_i$, proving the relation. □

Interpretation of the identity. Equation (5) distinguishes the size of the completion operation from its excess persistent work. Immediately before completion, U_{k-1} candidates remain unchecked. Of those, U_H would still be unchecked if the planner remained lazy through the horizon. Both policies pay for the other candidates by the horizon, even though they may test them on different queries. Section 8.4 works through this difference on an observed workload: 429 candidates remain at the chosen boundary, 20 are later needed by lazy continuation, and 409 remain excess checks at the final request.

Decision criterion from the identity. Signed cumulative service differences already underpin the index-acquisition scheduling framework of Bruno and Chaudhuri [10]. Equation (5) derives the net roadmap-validation charge and obtains the required query-service differences from the two deterministic executions. Let $D_j = \sum_{i=1}^{j} d_i$ and $D_0 = 0$. Minimizing the resulting work over all finite completion dates gives

$$k^* \in \arg\min_{1 \le k \le H} D_{k-1}. \tag{6}$$

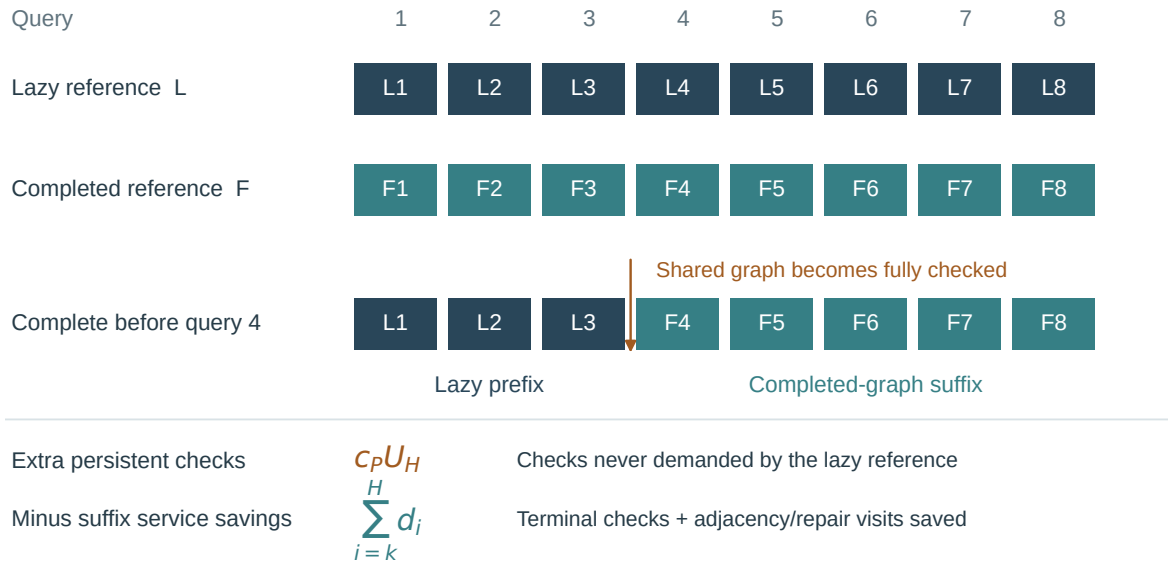


Fig. 6 Constructed illustration of the two-trace calculation. An intermediate policy takes its pre-completion query service from the permanently lazy trace and its subsequent query service from the completed-roadmap trace. Every finite completion policy ultimately tests the same persistent candidate set. Relative to permanent laziness, its additional persistent work is therefore the set still unused at the horizon, U_H, independently of the chosen boundary k. The comparison is determined by this investment and the accumulated suffix service saving. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

Completion improves on permanent laziness exactly when

$$D_H - \min_{0 \le j < H} D_j > c_P U_H. \tag{7}$$

For fixed terminal and adjacency weights, changing the persistent-test price changes the cost of every finite completion by the same amount relative to permanent laziness. Their ordering is therefore fixed when only the persistent-test price varies. Varying a common price for both persistent and terminal checks also changes the query-service savings. A running prefix minimum computes the best finite date and its comparison with laziness for every horizon in $O(H)$ arithmetic operations once the two traces are available. Explicitly listing the work of every eligible date at every horizon yields $H(H+1)/2$ entries.

Corollary 2 (When delaying completion helps) *At a fixed horizon, completing before query k costs less than completing before query 1 exactly when $D_{k-1} < 0$. If all preceding prefix sums are nonnegative, upfront completion weakly dominates every later finite completion point.*

Proof Subtract Eq. (5) at $k = 1$ from its value at k. The persistent term cancels and the difference is D_{k-1}. □

The order of service savings matters. A delayed policy beats upfront completion when the early queries collectively cost less to serve lazily. To beat permanent laziness as well, the remaining queries must save enough service work on the completed graph to repay its excess persistent checks.

5.3 Cost frontiers and controller evaluation

Each completion–horizon pair has three integer coefficients:

$$\Delta W_{k,H} = c_P U_H + c_T \Delta T_{k,H} + c_A \Delta A_{k,H}, \tag{8}$$

Algorithm 1 Best completion decisions from two operation traces

Require: Traces for Q queries: lazy (U_i, t_i^L, a_i^L) and completed (t_i^F, a_i^F); weights c_P, c_T, c_A
Ensure: Best finite completion point and comparison with permanent laziness at every horizon
Set cumulative service saving $D \leftarrow 0$, minimum eligible prefix $M \leftarrow 0$, prefix index $j \leftarrow 0$
for $H = 1, \ldots, Q$ **do**
  $D \leftarrow D + c_T(t_H^L - t_H^F) + c_A(a_H^L - a_H^F)$
  Set best finite completion $k^* \leftarrow j + 1$ and excess work $\Delta \leftarrow c_P U_H - D + M$
  Report (k^*, Δ) and choose completion if $\Delta < 0$; otherwise choose permanent laziness
  if $D \leq M$ **then**
    $M \leftarrow D$; $j \leftarrow H$ ▷ Prefix becomes eligible at the next horizon
  end if
end for

where $\Delta T_{k,H} = \sum_{i=k}^{H}(t_i^F - t_i^L)$ and $\Delta A_{k,H} = \sum_{i=k}^{H}(a_i^F - a_i^L)$. Retaining these coefficients describes the full cost region in which a completion point improves on laziness. For a one-parameter comparison at positive collision-test prices, set $c_P = c_T = \rho$ and $c_A = 1$. Then ρ is the cost of one collision test in units of adjacency/repair-visit work, and

$$\begin{aligned} \Delta W_{k,H}(\rho) &= \rho \Delta C_{k,H} + \Delta A_{k,H}, \\ \Delta C_{k,H} &= U_H + \Delta T_{k,H}. \end{aligned} \tag{9}$$

Each candidate yields an affine inequality, so the exact favorable intervals follow directly, including zero or negative collision-count differences.

When every finite completion has $\Delta C_{k,H} > 0$, define

$$\rho^*(H) = \max\left\{0, \max_{1 \le k \le H} \frac{-\Delta A_{k,H}}{\Delta C_{k,H}}\right\}. \tag{10}$$

Some completion improves on permanent laziness exactly for $0 < \rho < \rho^*(H)$. At larger collision-test prices, even the best completion point cannot recover its extra tests through saved adjacency/repair visits. When $\rho^*(H) > 0$, equality gives a tie with the lazy baseline. A zero frontier means that no positive price favors completion. The stored separate coefficients also permit unequal persistent and terminal test prices.

The *hindsight reference* chooses the least modeled work across permanent laziness and all finite completion dates on the observed sequence. Algorithm 1 computes that choice at specified weights. For a controller that completes and loses to permanent laziness, the reference determines whether another date would have won. For a controller that stays lazy, it identifies any missed opportunity to complete.

6 Completion controllers and terminal-validation controls

The two-trace criterion evaluates every completion position on an observed workload. To examine how a particular rule chooses among those positions, the experiment also runs controllers that decide from information available as requests arrive. A separate paired comparison changes terminal validation after the same persistent completion event. These controls distinguish selecting when to complete from selecting how to attach later queries, using the shared representation and search engines in Section 4.

6.1 When the planner requests completion

The count-based controller uses work already observed to decide whether to complete the graph before another request arrives. Let U_q be the number of persistent candidates whose validity remains unknown after request q. Let R_q be the cumulative number of vertices processed in candidate searches after the first search of each request. Processing means removing a vertex from the search queue for expansion or distance repair. The rule requests completion when

$$R_q \ge \alpha U_q, \qquad \alpha = 4. \tag{11}$$

The factor assigns four repeated-search vertex processings to one unknown persistent edge. It is the coefficient carried from the preceding archived experiment and is fixed in this evaluation. Each

newly tested persistent candidate reduces U_q by one, so the signal credits information already obtained. When the condition holds, the remaining validation is performed *before the next request*. A decision after the final request therefore incurs no completion work within the measured sequence.

Because U_q decreases as collision results accumulate, the threshold uses the remaining validation cost rather than the original construction budget. A fixed-initial-budget A* control replaces U_q with the initial number of persistent candidates while retaining the same conversion factor. This comparison isolates whether crediting already resolved candidates changes the completion decision.

A measured-time variant uses a different description of the same accumulated work. Its left-hand side is cumulative time in candidate searches after the first search and in the resulting edge-repair operations. Its right-hand side is the mean duration of collision checks observed so far multiplied by U_q. This mean includes observed construction checks where applicable. The signal contains no future request information. Timers are present in all variants, so the comparison does not introduce instrumentation only in the time-based policy. The count signal and the time signal are evaluated as declared decision rules.

6.2 What the completion event changes

At the same count-triggered completion event, two policies perform the same outstanding checks on the persistent graph (Figure 7). The *bundled* policy then checks every candidate for the new start and goal before searching each subsequent request. The *persistent-only* policy checks terminal candidates on demand. Comparing these branches holds the completion date fixed. The two-trace analysis uses persistent-only completion to compare different dates while keeping terminal validation lazy throughout.

Algorithm 2 states the sequence for the paired count-triggered policies. The search engine is selected independently. The algorithm makes the timing of completion visible: the decision is recorded after a request and executed only if another request arrives. The same completed graph can therefore be tested with two attachment policies without changing when it was constructed.

6.3 Accounting for the recurring collision checks

A count identity expresses exactly which geometric tests the paired intervention removes. For request q, let T_q be all terminal candidates and $S_q \subseteq T_q$ the subset evaluated by persistent-only completion. Let q_c be the first request served after the common completion event. Let $C_B(Q)$ and $C_I(Q)$ be the cumulative collision-call counts of bundled and persistent-only completion through request Q. Then

$$C_B(Q) - C_I(Q) = \sum_{q=q_c}^{Q} \big(|T_q| - |S_q|\big), \qquad (12)$$

where B denotes bundled completion and I persistent-only completion. The vertical bars denote set size. If completion has not occurred, the sum is empty and the difference is zero. Each summand is the number of terminal candidates that eager attachment checks but lazy attachment does not request. Since $S_q \subseteq T_q$, this contribution is nonnegative.

The identity follows by matching operations. Before q_c, the policies execute the same checks. At completion, they evaluate precisely the same unknown persistent candidates. Thereafter all persistent states are known. For each later request, bundled completion evaluates every member of T_q once, while persistent-only completion evaluates every member of S_q once and retains its result for that request. Subtracting and summing gives Eq. (12). The relation predicts the collision-call difference without requiring a timing model. Search and distance-label repair can change when unchecked terminal edges remain available, so the empirical analysis records those operations and total elapsed time alongside the exact count difference.

7 Evaluation design

The evaluation asks whether two reference executions reproduce the work of every completion date and how often a delayed date improves on both upfront completion and permanent laziness. Each possible completion is executed directly on the same ordered requests to check the reconstruction. The resulting operation counts support comparisons over a continuous range of collision-test prices.

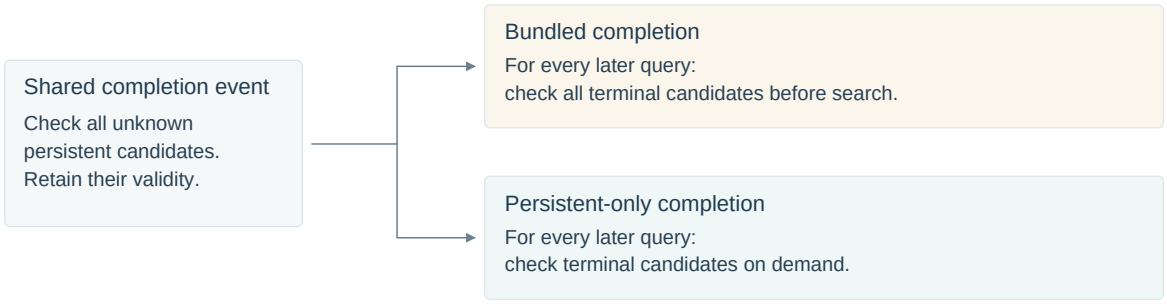


Fig. 7 The intervention separates a once-per-map investment from a recurring request policy. Both branches complete the same unknown persistent connections at the same event. Bundled completion additionally checks all candidates involving each later S and G before search. Persistent-only completion checks these terminal candidates on demand. The schematic defines the controlled contrast; it makes no assertion about which branch is faster. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

The fourteen-mode timing experiment in Figure 8 compares search implementations, cross-request retention of LPA* labels, terminal attachment, and the completion signal on those workloads.

7.1 Map–robot settings and query distributions

A *map–robot setting* is one obstacle geometry evaluated under a specified robot model; it is distinct from a robot configuration, which describes an individual placement. The primary corpus contains 40 generated point-map settings: four families, two obstacle counts, and five seeds per combination. The families are rectangles, convex polygons, concave polygons, and corridor walls; obstacle counts are eight and sixteen. Seeds 10–14 select the maps from the previously archived corpus. Twenty external point maps use the released Small fixed/random-start geometry of Zhang and colleagues [23]. Twelve square-robot settings derive from rectangle and corridor maps with seeds 10–12. This gives 72 map–robot settings derived from 60 source maps. Table 2 keeps the derived finite-body settings separate from their source geometries.

For each setting, the dispersed sequence samples across a window extending by 15% of the map span beyond the obstacle bounding box in each direction. The localized sequence samples in the horizontal 10–40% and vertical 25–75% ranges of that window. The shifted sequence uses that left region for the first twenty requests and the horizontal 60–90% range for the next twenty. The minimum start–goal separation is 0.3 times the map span for dispersed requests and 0.06 times the span for the other regimes. Endpoints strictly inside obstacles and pairs below the declared minimum separation are rejected. Queries are not selected according to planner success or measured performance. Seeds are derived from the setting identifier, regime, and a fixed study identifier. Stored inputs and their hash precede the released measurement run.

Figure 9 shows two actual requests from an archived shifted workload. Request 12 lies in the left region and request 23 in the right region. Within each twenty-request half, the first route with the largest segment count was selected to show an obstacle detour. This illustration selection does not change the evaluation corpus. A short route in one region can use only a small part of the candidate graph. Moving the request region can then make different connections useful while leaving the obstacle geometry unchanged.

The edge-set comparison in Figure 9 reconstructs the full visibility graph and its tangent-reduced subset for each recorded request. All visible connections are retained in the left panels, and all connections surviving the tangency filter are retained in the right panels. Independent geometric tests verify every drawn graph edge; a weighted reference search verifies that both graphs retain the recorded shortest-path length. The enlargements

Algorithm 2 Serving requests with matched persistent completion

Require: Obstacle map; ordered requests; selected search engine; terminal policy (bundled or persistent-only)
Ensure: One shortest valid route or failure status per request, with operation records

Construct persistent candidates; mark their validity unknown
Set completion-pending and completed to false; set repeated-search count to zero
for each request q with start s and goal t **do**
    if completion-pending and not completed **then**
        Check every unknown persistent candidate; store each result
        Mark completed; record the changed persistent edge costs
    end if
    Remove old terminal edges and update s, t
    if an endpoint is invalid or $s = t$ **then**
        Record invalid-endpoint failure or the zero-length valid route; reset search state
    else
        Create the new terminal candidates
        if completed and terminal policy is bundled **then**
            Check all terminal candidates and store their states for this request
        end if
        Initialize or repair search state according to the selected engine
        repeat
            Find a minimum-cost provisional route, or establish disconnection
            If a route exists, check its unknown edges in order until the first collision
            If a collision is found, set that edge cost to infinity and restart or repair
        until a fully valid route is found or no provisional route exists
        Record the route or failure and the measured operations
    end if
    Accumulate repeated-search vertex processings; count unknown persistent edges U_q
    if not completed, $U_q > 0$, and $R_q \geq 4U_q$ **then**
        Mark completion-pending for the next request
    end if
end for

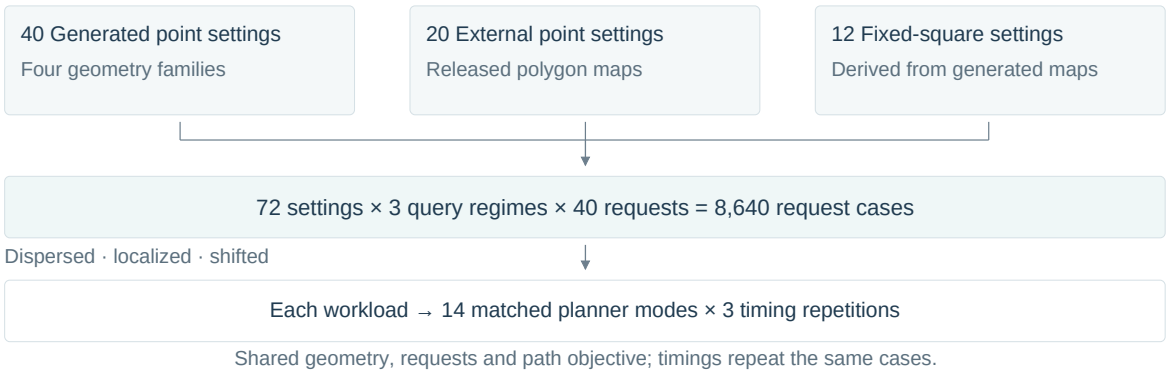


Fig. 8 Units and controls in the recorded experiment. A map–robot setting combines obstacle geometry with the point or fixed-square model. A workload is one ordered sequence of forty requests on that setting. All fourteen modes receive the same workload; the three timing repetitions are repeated executions, not new maps. Source-map clustering retains the relationship among the three request regimes. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

distinguish a permitted boundary-coincident segment from penetration into a forbidden obstacle interior. These reconstructions explain the representation without changing the measured planning runs.

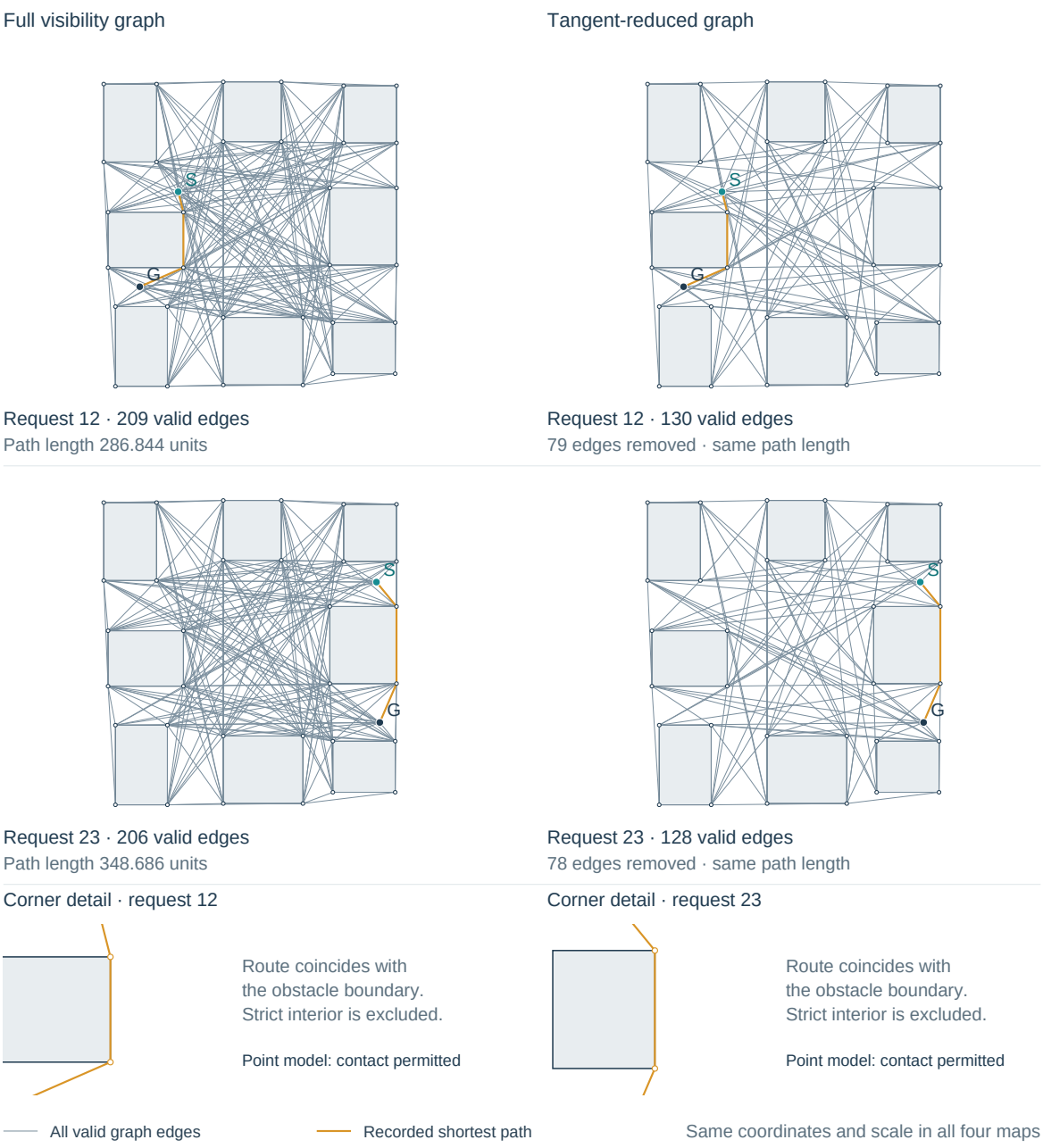


Fig. 9 Full and tangent-reduced graphs for archived requests 12 and 23 from `rectangles_8_10__shifted`. Every valid edge is drawn, using identical coordinates and the same 34 vertices within each pair. Edge counts decrease from 209 to 130 and from 206 to 128, respectively. Both graphs retain the recorded shortest-path lengths: 286.844 and 348.686 coordinate units. Thin amber lines reproduce the archived paths. Enlargements show the boundary-coincident segment in each path; boundary contact is permitted for these point-robot cases. Exact-rational geometric tests and weighted Dijkstra searches verify the drawn edge sets and equal lengths. This reconstruction is outside the recorded timings. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

Table 2 Evaluation material. Each setting has three 40-request sequences. The square settings derive from generated source maps and are analyzed as a separate group

Group	Settings	Workloads	Request cases
Generated point maps	40	120	4,800
External point maps	20	60	2,400
Fixed-orientation square	12	36	1,440
Total	72	216	8,640

The three regimes vary which parts of the same persistent graph a sequence is likely to query. For the shifted regime, the change in request region also has a known position in the sequence. Across all regimes the sampling windows constrain endpoint selection and the obstacle geometry stays fixed.

7.2 Exhaustive validation of the two-trace criterion

For each of the 216 workloads, restarted A* and query-local LPA* generate the permanently lazy and fully completed reference traces. This gives 432 engine–workload combinations and 864 reference traces. The completed reference resolves every persistent candidate before query 1 and retains lazy terminal validation. All inputs, collision predicates, candidate ordering, selectors, and search code are identical to the archived implementation.

The direct comparison reconstructs the lazy state immediately before each query $k = 1, \ldots, 40$. An independent copy of that state completes its unresolved persistent candidates and executes queries k through 40. Setting the planner's existing completion-pending flag invokes the same completion operation used by the original controller. The search and geometry routines are unchanged. Each suffix query is compared with the completed reference on eight recorded service fields: terminal checks, terminal candidates, processed vertices, adjacency/repair visits, explicit one-step recomputation calls, queue insertions, search calls, and extra vertex processings after the first candidate search. The three cumulative coefficients in Eq. (8) are checked separately as integers at every horizon $H \geq k$. Each returned length is compared with the frozen independent full-visibility reference. The lazy traces are also checked against eleven archived operation fields per query.

The extension measures operation counts. Its protocol and input and source hashes were fixed before execution on 18 November 2026. The recorded environment uses Python 3.12.14, NumPy 2.3.5, and pandas 2.2.3 on Linux x86-64. Online Resource 2 contains the traces, every directly checked coefficient, the environment record, and the execution and analysis scripts. The archived `scans` field records adjacency/repair visits, including LPA* self visits; `rhs_updates` counts calls to the routine for recomputing the one-step value in Eq. (2).

7.3 Exact cost regions and descriptive sensitivity analysis

For each engine, workload, and horizon, the analysis asks two questions under the one-parameter model $c_P = c_T = \rho$ and $c_A = 1$ in Eq. (9). First, over which positive ρ intervals does some completion date cost less than permanent laziness? Second, over which intervals does a delayed date cost less than both upfront completion and permanent laziness? Exact rational inequalities give the first set of intervals. Intersecting the strict comparisons against both alternatives gives the second. The endpoints of strict intervals are open. When selecting a policy, ties favor permanent laziness, then the latest finite completion point.

The graphical sensitivity display uses 81 logarithmically spaced prices from 1 to 10,000. Five additional values, 4, 16, 64, 256, and 1,024, make the tabulated examples interpretable. These prices are declared operation-cost scenarios. The exact interval calculation covers favorable regions between or outside these sampled prices. For each delayed date, its fractional gain over the better endpoint policy is a ratio of affine functions of ρ. Its denominator changes branch only where upfront completion and permanent laziness tie. Evaluating that crossing and the limits $\rho \to 0^+$ and $\rho \to \infty$ gives the continuous supremum, because each ratio has a derivative of fixed sign

between crossings. Results retain the generated, external, and fixed-square groups and all three query regimes. Individual engine–workload combinations are the descriptive units; timing repetitions add no independent units to this analysis.

For a triggered workload, the fraction U_{40}/U_{k-1} measures how much of the outstanding validation purchased at query k would still be unused by lazy continuation after query 40. A fraction of one means that every check purchased at completion remains unnecessary to that continuation. The archived count rule is also compared with the hindsight minimum under the same weights. Relative excess work is its work minus that minimum, divided by the minimum. The comparison uses the rule's recorded completion query and preserves the no-completion outcome when its threshold never fires.

7.4 Complementary timing comparisons and execution

The search-state comparison holds validation permanently lazy. The attachment comparison holds the count-triggered completion event fixed and changes the treatment of later terminals. Fourteen modes are measured, where a mode is one specified combination of search engine and validation policy. Eager A* supplies a complete-validation reference and fixed-initial-budget A* supplies a threshold control. Each of the three search engines is combined with permanent laziness, bundled count-based completion, persistent-only count-based completion, and measured-time bundled completion. The paired contrasts compare A* with query-local LPA*, retained with query-local LPA*, bundled with persistent-only completion, and count-based with measured-time completion. Each contrast keeps the workload and the other specified policy choices fixed.

Each workload is repeated three times for every mode. Modes run serially in a deterministic shuffled order. The resulting 362,880 timed policy–request executions correspond to 8,640 distinct request cases. Cumulative time includes the initial graph construction, removal and insertion of terminal edges, geometric checking, search, repair, heuristic rekeying, completion, and instrumentation. It excludes input generation, result serialization, reference construction, analysis, and figure rendering. The archived timing record identifies Python 3.13.5 and a Linux x86-64 platform with glibc 2.41. It also records the executed source hashes and the input hash.

ChatGPT assisted with implementation and analysis scripting (see the AI declaration). The archived timing run supplies the full-sequence timing comparisons. Online Resource 1 identifies the run by its input and executed-source hashes and supplies each map, query, policy, and repetition record. A separate matched-checkpoint experiment measures the post-completion suffix of these same triggered workloads.

7.5 Verification and measures

Geometric agreement establishes that computational comparisons concern equivalent solutions. Verification therefore precedes interpretation of timing differences. Every mode and repetition must agree on route length or failure for a given request. Independent verification then constructs a full visibility graph using exact-decimal integer predicates, without importing the planner's tangent test or search engine. Decimal coordinates are scaled to a shared integer grid. Segment–polygon decisions test exact orientations, boundary contacts, and open subsegments between collinear vertex contacts. Dijkstra search supplies a reference for every request. Length comparisons use relative tolerance 10^{-9} and absolute tolerance 10^{-8} in coordinate units. The geometric decisions are exact for the stored decimal inputs. The integer decisions are additionally cross-checked against the archived rational-arithmetic implementation on selected segments and analytic fixtures.

The first repetition retains every route, permitting its endpoints, summed length, and individual segments to be checked independently. For geometric checking, duplicate undirected segments are counted once within each map–robot setting, since the obstacle geometry belongs to that setting. Additional implementation tests exercise changing weighted graphs, moving endpoints, invalid endpoints, coincident endpoints, and analytic geometric cases. An accounting check evaluates Eq. (12) for every matched bundled/persistent-only request, while also checking equality of their switch requests and persistent collision counts.

The performance record distinguishes collision calls, processed vertices, adjacency/repair visits,

priority-queue insertions, explicit one-step recomputation calls, candidate-search calls, and elapsed time. A processed vertex is removed from a search queue for expansion or repair; an adjacency/repair visit counts one examination of a stored neighbor or the explicit self entry in an LPA* repair loop. Incremental updates can inspect neighbors without processing an equal number of queue vertices. Reporting both measures avoids treating fewer processed vertices as an automatic reduction in all search work.

7.6 Paired analysis and uncertainty

Each comparison uses the same ordered requests, so the analysis pairs results within a workload. For a mode and workload, cumulative runtime is summarized by the median of its three repetitions. A paired ratio divides this median by the corresponding median of the comparator on the same workload. Group summaries are medians of these paired ratios, not ratios of pooled means. A ratio below one favors the numerator. Results are reported at one, five, twenty, and forty completed requests.

Uncertainty is estimated by repeatedly drawing source maps with replacement and recalculating the paired summary. This cluster-bootstrap procedure uses 5,000 resamples and retains all three query regimes whenever a map is drawn. The 2.5th and 97.5th percentiles define the reported 95% interval. Generated, external, and square groups are kept separate. Repeated timings are not treated as independent maps. The square settings are not pooled with their parent geometries to increase an effective sample size. Comparisons restricted to workloads where the common count rule triggered completion are descriptive paired-condition analyses. All-workload summaries are reported separately so that the triggered subset does not replace the complete evaluation.

7.7 Timing phases and matched-checkpoint replay

A cumulative runtime comparison includes work performed before the intervention. To determine whether that common prefix influences the observed ordering, the original per-request records are partitioned within each repetition at its recorded completion request. The pre-event phase includes initial construction and all earlier requests. The completion phase is the separately instrumented persistent-completion block. The request-work phase contains the remainder of the completion request and every later request. Their sum is checked against the original end-to-end total in integer nanoseconds. Phase differences are calculated between matched runs before taking medians. Ranking stability is assessed from the number of repetitions favoring each policy, and pre-event operation profiles are compared independently of elapsed time. This analysis includes all 129 count-triggered engine–workload combinations: 60 for A*, 34 for query-local LPA*, and 35 for retained LPA*.

A focused follow-up controls the shared state at the intervention. For every triggered engine–workload combination, the unchanged planner replays the preceding requests to reconstruct a checkpoint immediately before completion. Independent copies of that checkpoint execute bundled and persistent-only completion. Each copy receives the same remaining requests, collision cache, graph, and search state; the terminal-validation policy is the changed input. Checkpoint construction and copying are outside the timed region. The measured suffix includes the persistent-completion block and all remaining request work. Its route lengths and twelve operation counters per request are compared with the archived implementation outcomes.

The follow-up uses one unrecorded warm-up suffix per policy and twenty recorded paired repetitions without adaptive stopping. The two policies execute consecutively within each pair; their order alternates, producing ten executions in each order. Cyclic garbage collection is performed before the pair and suspended during both suffixes. The process is pinned to one available logical CPU. The captured environment identifies an AMD EPYC 9V74 processor, Python 3.13.5, NumPy 2.3.5, and a Linux x86-64 platform. The timer is the monotonic `perf_counter_ns` clock. This environment record belongs to the follow-up; the original run's metadata is retained separately.

Within each workload, the follow-up ratio divides the median persistent-only suffix time by the median bundled suffix time over the twenty repetitions. Group medians and 95% intervals use 5,000 source-map-cluster bootstrap resamples of the triggered subset, retaining every included query

regime for a selected map. Paired-repeat rankings and separate summaries for the two execution orders assess timing stability. Replaying reconstructed checkpoints adds timing observations on the same maps and route requests. Online Resource 1 provides the fixed protocol, hashes, raw suffix records, phase calculations, and all order-stratified summaries.

8 Results

The two reference traces reproduced the operation counts for every tested completion date on the forty-request workloads. These counts locate the collision-test prices at which completion saves work and identify the cases in which delaying it saves more than either endpoint policy. The elapsed-time comparisons use paired workload ratios. A ratio below one means that the numerator policy was faster on the same workload.

8.1 Geometric agreement and verification of the paired intervention

All fourteen modes and all three repetitions agreed with the common length outcome on the 8,640 request cases. The independent full-visibility construction verified 8,640 reference lengths. All 120,960 archived first-repetition routes passed the endpoint and length checks; 16,843 distinct setting–segment pairs passed the independent exact-decimal geometric check. The rational-predicate cross-check examined 1,440 selected segments without a disagreement. The planning comparisons therefore concern the work required to obtain the same verified shortest-path length.

For the bundled and persistent-only policies, the count-triggered completion request and the persistent collision counts agreed throughout all 77,760 paired request comparisons. Equation (12) held exactly in each comparison. On workloads where the rule triggered, both branches therefore reached the same completed persistent graph at the same request before applying their respective terminal policies.

8.2 Every completion point is recovered by the two traces

The direct experiment executed 17,280 completion branches across the 432 engine–workload combinations. These branches served 354,240 suffix queries. All 2,833,920 comparisons of post-completion service fields agreed with the completed reference, and all 1,062,720 cumulative coefficient comparisons satisfied Eq. (8) exactly. Across the two reference traces and the direct branches, 388,800 returned lengths agreed with the independent geometric references. The 190,080 checked lazy-trace operation fields also agreed with their archived values. There were no discrepancies.

This agreement validates the reconstruction on the tested workloads. At a fixed horizon, the excess persistent-check count is identical for all completion dates. The completed trace supplies the service counts on each remaining query, regardless of the preceding lazy prefix. Two 40-query operation traces per engine–workload pair therefore suffice for the full 40-point comparison.

8.3 Excess collision checks bound the price at which completion pays

At the forty-query horizon, every one of the 17,280 engine–workload–completion-date comparisons incurred more total collision checks than permanent laziness. The smallest increase was 50 checks. Thus all observed completion alternatives fall under the positive-excess-check case of Eq. (10): the favorable region, when it exists, ends at a finite collision-test price.

For restarted A*, 65 of 216 workloads had a zero frontier, meaning that no completion point reduced modeled work at any positive price. The corresponding count for query-local LPA* was 33 of 216. Thus some finite completion improved on permanent laziness at a workload-specific positive price on 151 of 216 A* workloads and 183 of 216 query-local LPA* workloads. Across all workloads, the median frontier was 2.262 adjacency/repair-visit work units per collision test for A* and 6.916 for query-local LPA*. Their largest frontiers were 608.966 and 704.122, respectively. Table 3 retains the corpus groups and includes the zero-frontier workloads in each summary.

Completion frontiers differed across the three request regimes. All 72 dispersed workloads per

Table 3 Exact completion frontiers at forty queries. The critical price ρ^* is measured in adjacency/repair-visit work per collision test. Medians and quartiles include zero frontiers. A zero frontier means that no finite completion reduces modeled work at any positive collision-test price

Engine	Corpus	Workloads	Zero frontiers	Median ρ^*	Interquartile range
Restarted A*	Generated	120	45	0.464	[0.000, 9.301]
Restarted A*	External	60	14	4.890	[0.616, 22.425]
Restarted A*	Fixed square	36	6	3.998	[0.287, 20.390]
Query-local LPA*	Generated	120	20	3.132	[0.098, 29.664]
Query-local LPA*	External	60	8	11.793	[1.682, 46.764]
Query-local LPA*	Fixed square	36	5	14.415	[1.950, 48.733]

engine had a positive frontier. Localized requests produced zero frontiers on 34 of 72 A* workloads and 16 of 72 query-local LPA* workloads; shifted requests produced 31 and 17. The median frontiers for dispersed requests were 24.701 and 69.163, compared with 0.182 and 1.146 for localized requests. Permanent laziness tested a median 25.80% of persistent candidates under dispersed requests, 2.10% under localized requests, and 2.80% under shifted requests, with identical coverage summaries for the two engines. The dispersed regime thus combined greater persistent coverage with higher median frontiers on the evaluated workloads.

Figure 10a shows the resulting sensitivity to the price assigned to a collision test. At $\rho = 16$, some completion improved on permanent laziness for 52 of 216 A* workloads and 83 of 216 query-local LPA* workloads. At $\rho = 1{,}024$, none did. These counts describe the exact affine comparison under the declared weights; the interval calculation supplies the boundaries between the displayed prices.

8.4 When a delayed completion beats both endpoint policies

At horizon 40, delaying completion strictly improved on both upfront completion and permanent laziness on one of 216 A* workloads and 40 of 216 query-local LPA* workloads, with favorable positive prices evaluated separately for each workload. Of those 40 query-local cases, the sampled prices from 1 to 10,000 detected 25. In the remaining fifteen, the entire favorable interval lay below one.

The single A* case, the localized workload on the external random-start Small4 map, makes the acquisition and timing comparisons concrete (Figure 10b). Of its 432 persistent candidates, permanent laziness checks 23 by query 40, leaving $U_{40} = 409$. Before query 15, 429 candidates remain unchecked; lazy continuation would eventually test 20 of them. Completing there therefore adds 409 persistent checks over the whole workload. At $\rho = 1$, permanent laziness costs 6,912 work units, upfront completion 6,375, and completion before query 15 costs 6,360. The delayed date saves 15 adjacency visits relative to upfront completion at the same collision count, giving $15/6{,}375 \approx 0.235\%$ improvement. Its excess over permanent laziness is $408\rho - 960$: 409 extra persistent checks, one fewer terminal check, and 960 fewer adjacency visits. Delays before queries 11–15 beat both endpoints for $0 < \rho < 40/17$. The largest query-local LPA* gain at $\rho = 1$ is 3.247%, obtained by completing before query 11 in the shifted workload on the external random-start Small5 map.

The occurrence of a strict improvement and its magnitude are distinct. Over every positive price and completion date, the largest possible fractional gains at horizon 40 have suprema of 0.258% for A* and 3.361% for query-local LPA*, approached as $\rho \to 0^+$. Among the 40 query-local workloads admitting an interior gain, the median workload-specific supremum is 0.191%. Across all forty horizons, an interior gain occurs on 26 of 8,640 A* workload–horizon pairs and 1,172 of 8,640 query-local pairs, involving one and 41 distinct workloads, respectively. Thus most of the completion opportunity is captured by comparing upfront completion with permanent laziness; the exact date supplies a smaller, measurable additional opportunity in these sequences.

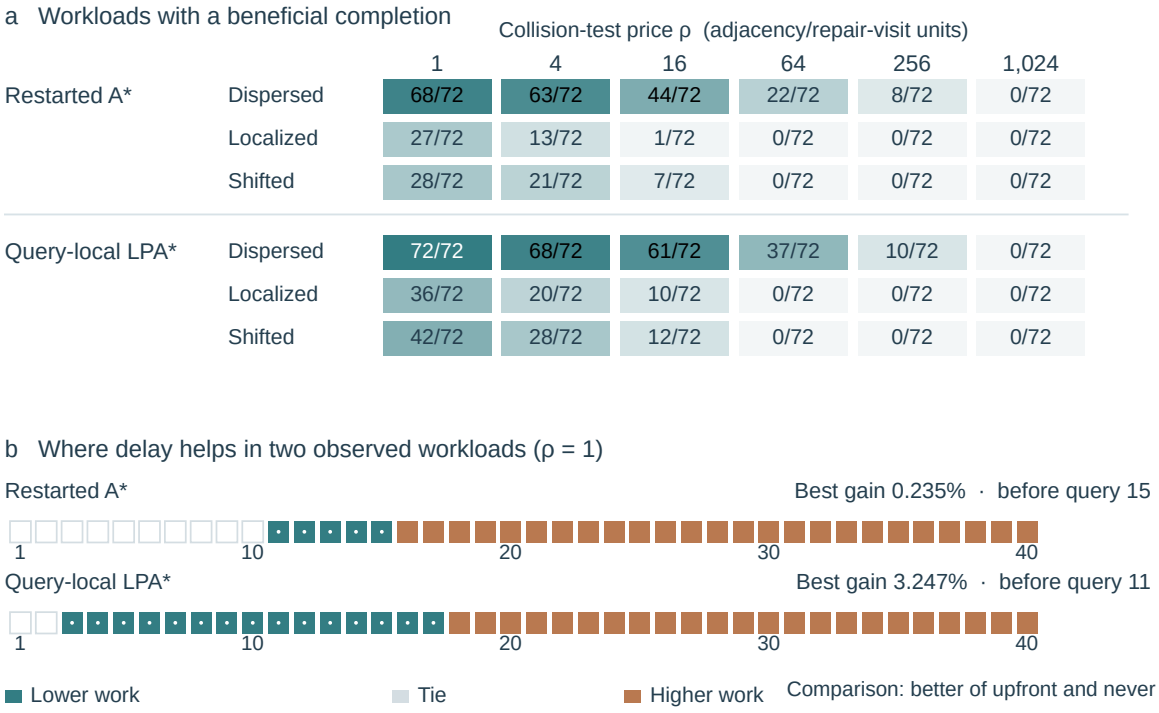


Fig. 10 Completion opportunity at a forty-query horizon. (a) Each cell reports the number of workloads for which some finite completion has lower modeled work than permanent laziness; every engine–regime group contains 72 workloads. Color redundantly encodes the proportion. The horizontal variable is the collision-test price ρ, with adjacency/repair-visit price one. (b) Every completion boundary is shown for two observed examples at $\rho = 1$. Dotted teal cells indicate lower work than both upfront completion and permanent laziness; outlined cells indicate a tie with their better outcome; brown indicates higher work. The cases are the largest strict delay gains at this price within each engine: the external random-start Small4 map with localized queries for A* and the random-start Small5 map with shifted queries for query-local LPA*. The latest tied best boundaries are queries 15 and 11. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

8.5 The count rule conflates completion opportunity with timing

The outstanding checks at the controller's switch can be separated into those that lazy continuation eventually needs and those that remain unused at query 40. Across the 60 triggered A* workloads, the median unused fraction was 90.5%, with interquartile range 64.8–98.6%. Across 34 triggered query-local LPA* workloads it was 72.3%, with interquartile range 63.9–95.9%. Figure 11a shows every triggered workload. In median counts, the respective switches purchased 222.5 and 148.5 checks, of which 191 and 141 remained unused; these are separately calculated workload medians.

The exact completion reference distinguishes different causes of a controller's loss. At the illustrative price $\rho = 16$, 24 of the 60 A* switches reduced work. Seventeen switches increased work even though another completion date would have reduced it, and nineteen increased work on sequences for which no date could win. Eleven additional workloads offered a favorable completion but never triggered. Query-local LPA* produced 20 beneficial switches, twelve timing losses, and two switches without any profitable date, while missing 51 favorable workloads through no switch (Figure 11b). These partitions follow directly from the recorded controller date and the minimum over all dates.

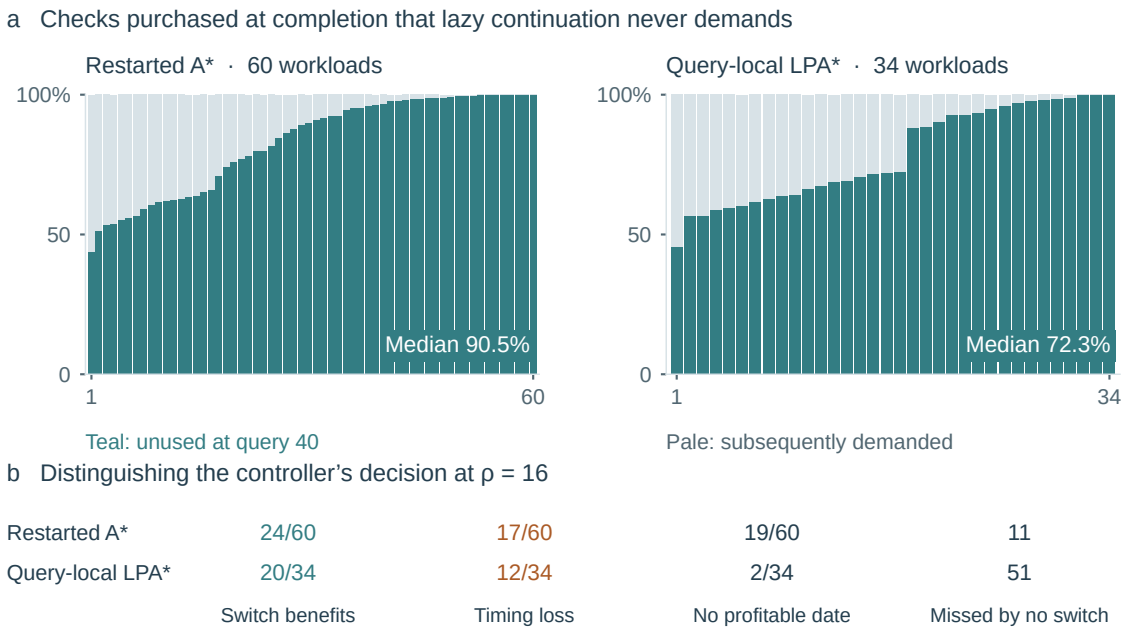


Fig. 11 Purchased validation and the controller's decision. (a) Each column is one triggered workload, sorted by U_{40}/U_{k-1} within the engine. Teal is the fraction of checks performed at completion that permanent lazy continuation still leaves unused at query 40; pale grey is the fraction it subsequently demands. The fraction is calculated for each workload and the median is taken across workloads. (b) At the declared model price $\rho = 16$, a timing loss means that the controller's chosen date increases work although some other date reduces it. No profitable date means that no finite completion reduces modeled work relative to permanent laziness. The last column counts favorable workloads on which the controller never completes and uses the full 216-workload population. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

8.6 Search-state reuse changes the distribution of computational work

Under permanent lazy validation, the elapsed-time comparisons distinguish the choice of search implementation from retaining labels across requests. With this common validation policy, Table 4 compares repairing distances within a request against restarting A*, then compares carrying LPA* labels across requests against resetting them at each request. Figure 12 reports processed vertices, adjacency/repair visits, and total time for the within-request comparison.

In the generated group, the median processed-vertex ratio was 0.909, while the adjacency/repair-visit ratio was 3.618 and the total-time ratio was 1.286. Fewer queue processings thus coincided with more examinations of stored entries. When labels survive a query change, the engine must update them for removed and inserted terminal edges and rekey inconsistent vertices when the goal changes. The retained/query-local ratios in Table 4 measure the combined effect of carrying those labels forward and repairing them.

8.7 Query-specific validation after matched persistent completion

The second comparison follows the fork in Figure 7: both policies complete the same persistent candidates, then handle later terminal connections differently. The count-based A* rule triggered completion on 60 of 216 workloads. Persistent-only completion was faster than bundled completion on 55 of those 60 matched workloads, with a pooled descriptive median total-time ratio of 0.750. Table 5 retains the group denominators and includes both incremental engines. Figure 13 shows every workload in the triggered condition.

The collision-count difference has a direct operational explanation. Across the three A* timing

Table 4 Search-state and validation comparisons at forty requests. The first two rows keep validation permanently lazy; the third compares eager and lazy validation under restarted A*. Values are median paired total-time ratios with 95% source-map-cluster intervals

Numerator / comparator	Generated	External	Fixed square
Query-local LPA* / A*	1.286 [1.248, 1.339]	1.304 [1.194, 1.354]	1.366 [1.245, 1.398]
Retained / query-local LPA*	1.257 [1.209, 1.316]	1.209 [1.169, 1.251]	1.158 [1.071, 1.255]
Eager / lazy A*	5.090 [4.021, 6.363]	5.294 [4.712, 6.076]	3.393 [2.743, 4.441]

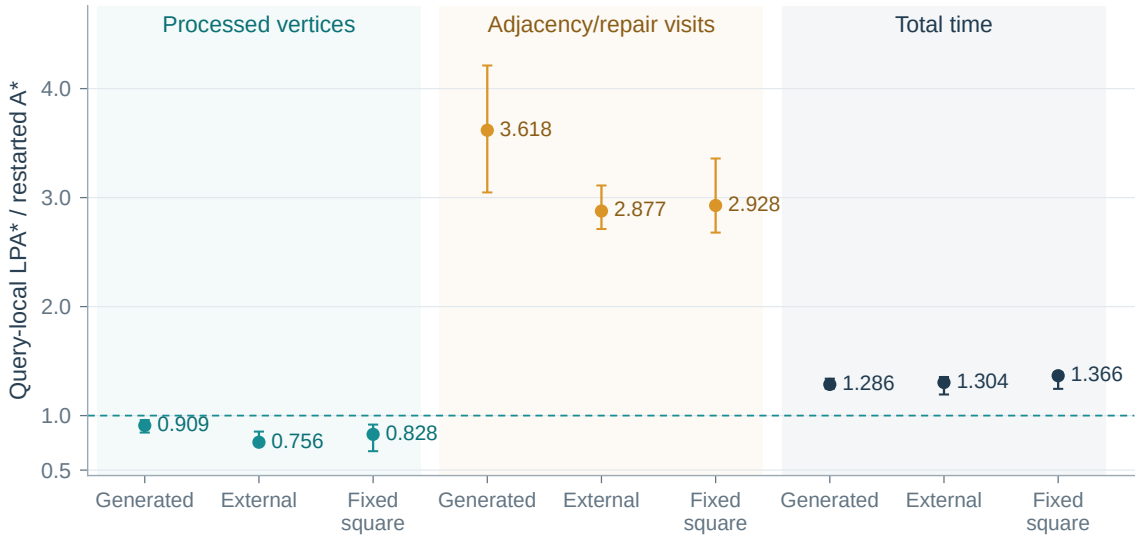


Fig. 12 Search-work measures separated into aligned metric groups. Query-local LPA* is compared with restarted A* under permanent laziness after forty requests. Each group contains the generated, external, and fixed-square settings in that order. Points and adjacent labels give median paired workload ratios; whiskers are the 95% source-map-cluster bootstrap intervals. The dashed reference denotes equal cost. Fewer processed vertices coexist with more adjacency/repair visits and longer total time in these summaries. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

repetitions, bundled completion incurred 130,134 additional terminal tests. Those are the candidates that on-demand attachment never needed to evaluate, exactly as counted by Eq. (12). Search can become more expensive in the opposite direction: a still-unknown terminal connection may invalidate a provisional route and require another search or repair. The incremental-engine rows preserve that tradeoff. For example, the generated query-local LPA* subset has a median persistent-only/bundled time ratio of 1.014 despite the nonnegative reduction in collision calls.

8.8 Persistent completion relative to permanent lazy validation

Comparison with permanent lazy validation evaluates the remaining investment in persistent completion. The paired terminal contrast above isolates the attachment schedule; this contrast includes the decision to complete the persistent graph itself. On the same 60 switched A* workloads, persistent-only completion was faster than permanent laziness on eight. Table 6 reports the triggered-only comparison; Table 7 reports all workloads, including those on which the controller never completed the graph. The two summaries distinguish behavior after an

Table 5 Persistent-only / bundled total time on the count-triggered subset. Ratios and faster counts are descriptive within the paired condition; n denotes workloads, not timing repetitions

Engine	Stratum	n	Median ratio	Faster
A*	Generated	28	0.799	24/28
A*	External	17	0.589	16/17
A*	Fixed square	15	0.862	15/15
Query-local LPA*	Generated	14	1.014	6/14
Query-local LPA*	External	11	0.714	11/11
Query-local LPA*	Fixed square	9	0.832	6/9
Retained LPA*	Generated	15	0.909	11/15
Retained LPA*	External	11	0.809	11/11
Retained LPA*	Fixed square	9	0.787	7/9

observed switch from behavior across the complete request population.

Figure 14 follows cumulative cost as more requests are served. Every point includes initial construction, so the graph shows whether subsequent reuse has recovered the preparation cost by that horizon. The generated-map medians remain close to, or above, the permanently lazy reference for the completion variants. Removing eager terminal checking resolves a specific additional cost, while the benefit of the persistent checks still depends on what the remaining requests use.

The external shifted workload in Figure 15 had the largest final persistent-only/permanently-lazy A* ratio among the 60 switched workloads and was selected on that basis. Permanent laziness performed 60 persistent checks over forty requests. Persistent-only completion performed 239, including 181 at the completion event before request 35. Both policies performed the same 171 terminal checks. The difference in persistent totals measures excess checks over the whole sequence, whereas the event count measures how much work fell on the completion request.

The corresponding median totals are 17.216 ms for permanent laziness, 40.608 ms for persistent-only completion, and 31.185 ms for bundled completion. Bundling adds 199 terminal tests relative to persistent-only completion in this case, while reducing adjacency/repair visits from 14,570 to 13,231. The two completion policies have identical pre-event operation counts in each repetition, although their pre-event elapsed-time curves differ. That timing difference precedes the intervention and cannot be attributed to the later terminal rule. The trace therefore illustrates the measured operations and timing variability without assigning every elapsed-time difference to one counter.

8.9 Count-based and measured-time completion signals

The final comparison changes the signal used to request completion. The time rule includes later candidate searches and edge repair, while the count rule includes processed vertices. Table 8 reports completed runs under both rules. Its denominators include three repetitions per workload because measured-time decisions can vary between repetitions. The count-triggered decision is deterministic for the fixed inputs and tie order. A pending decision after request 40 is not a completed construction event.

For A*, the measured-time rule completes in fewer runs than the count rule in every group. For either incremental engine, the measured-time rule completes in more. The all-workload time ratios in Table 7 and Figure 16 include the cost of making and executing those decisions. The direction of this frequency change is consistent with the different processing, inspection, and repair costs measured above. Frequency alone does not indicate success: additional completed checks must still prove useful to the remaining requests for the investment to reduce total time.

8.10 Timing robustness before and after completion

Reconstruction of all 36,288 stored cumulative records reproduced their original totals exactly. In the 387 paired triggered runs, corresponding to 129 engine–workload combinations and three

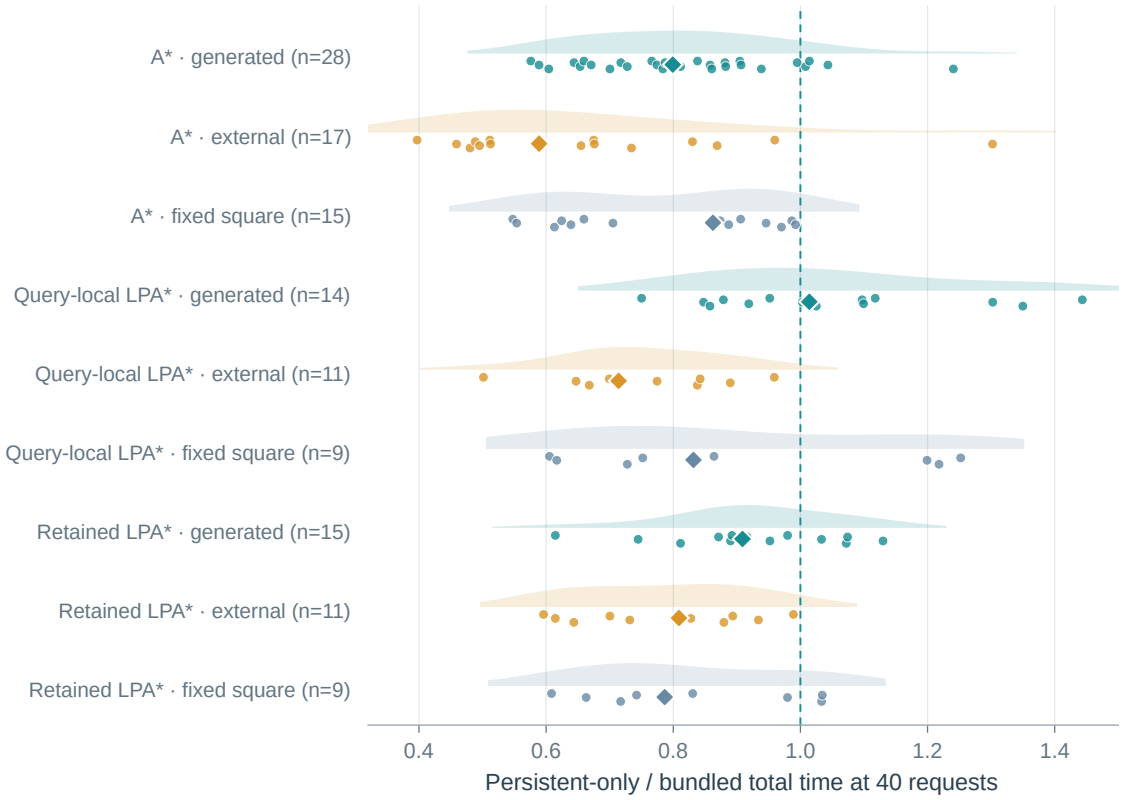


Fig. 13 Workload-level effects of changing terminal validation after the same count-triggered persistent completion. The raincloud display retains every archived persistent-only/bundled total-time ratio: points are individual workloads and diamonds are group medians. Each ratio uses the median of three timings per policy. Shaded profiles are descriptive Gaussian kernel-density curves using Scott bandwidth, normalized to equal height within each row; they add no observations and do not encode uncertainty intervals. The vertical reference marks equal time. All triggered workloads, including unfavorable cases, are retained. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

Table 6 Persistent-only / permanently lazy total time within the count-triggered subset. Parentheses give workloads where persistent-only completion is faster, over the subset denominator. These are descriptive conditional summaries

Engine	Generated	External	Fixed square
A*	1.276 (4/28)	1.231 (1/17)	1.137 (3/15)
Query-local LPA*	1.115 (2/14)	1.107 (1/11)	1.060 (3/9)
Retained LPA*	1.035 (4/15)	1.109 (2/11)	1.054 (1/9)

repetitions, the pre-event operation profiles were identical between bundled and persistent-only completion. Elapsed times nevertheless differed. Table 9 separates the original end-to-end ordering from the suffix ordering. For A*, 55 workload medians favored persistent-only completion end to end, and 46 favored it in all three repetitions. The corresponding suffix counts were 60 and 54. Five A* workload median rankings changed when the pre-event phase was excluded; the query-local and retained LPA* comparisons had two and three such changes. These are phase-specific measurements of the same original runs.

For the adverse case in Figure 15, the persistent-only minus bundled pre-event differences were 6.662, 10.700, and 9.841 ms in the three recorded

Table 7 All-workload total-time ratios to permanent laziness with the same engine at forty requests. Brackets are 95% source-map-cluster intervals

Completion policy	Generated	External	Fixed square
Bundled A*	1.014 [0.984, 1.062]	1.013 [0.976, 1.146]	1.100 [1.015, 1.168]
Persistent-only A*	1.028 [1.004, 1.063]	1.047 [0.986, 1.095]	1.019 [0.958, 1.042]
Initial-budget A*	1.024 [0.992, 1.062]	1.039 [1.001, 1.080]	0.995 [0.945, 1.066]
Time-triggered A*	1.007 [0.989, 1.030]	1.027 [1.007, 1.058]	0.996 [0.968, 1.046]
Time-triggered query-local LPA*	1.037 [1.011, 1.072]	1.019 [0.995, 1.073]	1.101 [1.038, 1.206]
Time-triggered retained LPA*	1.043 [0.999, 1.087]	1.063 [1.016, 1.110]	1.097 [1.042, 1.337]

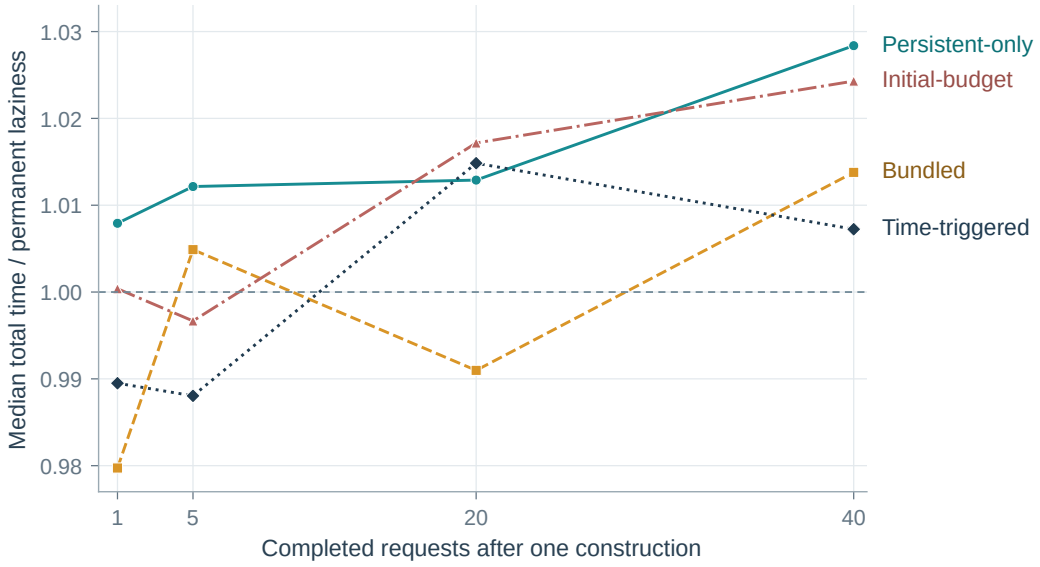


Fig. 14 Directly labelled cumulative cost trajectories on generated maps. At each of the four horizons, a marker gives the median paired total-time ratio to permanently lazy A*, including initial construction. Persistent-only, bundled, and initial-budget denote the count-triggered A* policies; time-triggered denotes measured-time bundled completion. Lines join the observed horizon summaries. The horizontal reference denotes equal total time. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

repetitions. Their median was 9.841 ms. The corresponding total differences had a median of 9.796 ms, while the completion-plus-request suffix differences were −3.593, 0.917, and −0.045 ms. Their median was therefore −0.045 ms, with persistent-only faster in two repetitions. These paired phase statistics are computed from individual runs; they are not differences between the medians plotted in Figure 15. The completion phase itself had unequal elapsed times despite the same collision-test count. The cumulative adverse ordering therefore includes a substantial timing difference already present before the terminal-policy intervention.

The original comparison with permanent laziness also exhibits repetition variability. Among the same 60 triggered A* workloads, eight median end-to-end ratios favored persistent-only completion; two workloads favored it in every repetition, forty favored permanent laziness in every repetition, and eighteen had mixed rankings. Online Resource 1 records repetition-level rankings for every contrast and horizon, including the all-workload and measured-time comparisons. The unfavorable aggregate completion comparison remains present in the original data.

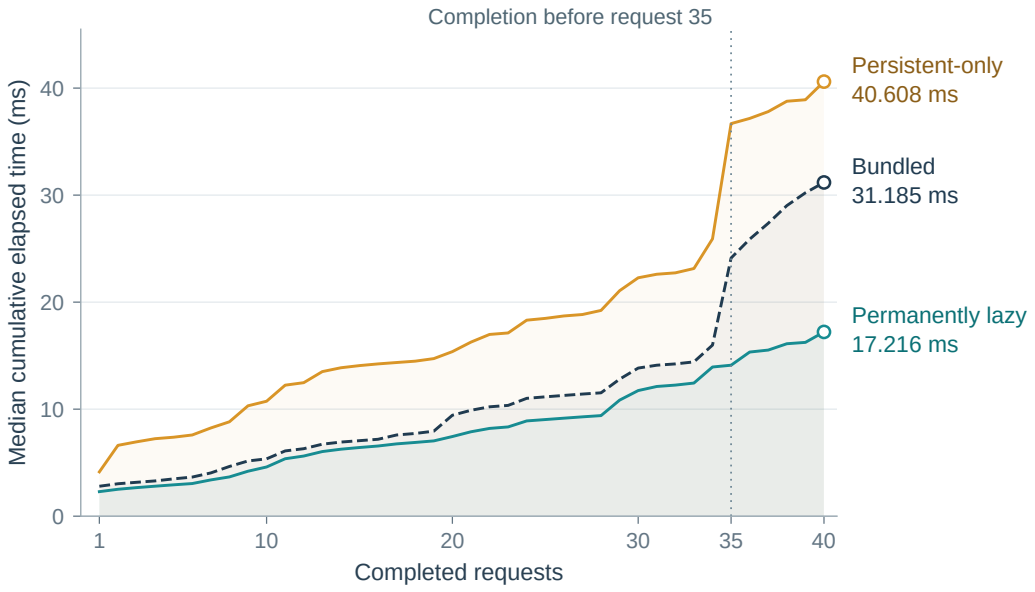


Fig. 15 Observed adverse-case trace for `maprandomStartAndGoalSmall8_shifted`. Lines and endpoint labels show the median cumulative elapsed time across three runs, including initial construction; translucent fills share a zero baseline and are not stacked. The event marker locates completion before request 35. Final totals are 17.216 ms for permanent laziness, 31.185 ms for bundled completion, and 40.608 ms for persistent-only completion. The case was selected post hoc by the largest final persistent-only/permanently-lazy A* ratio among switched workloads. Pre-event timing differences occur despite equal operation counts; Section 8.10 separates the phases within each repetition and reports the matched-checkpoint follow-up. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

Table 8 Completed runs under the count signal followed by the time signal in each cell. The column denominator includes three repetitions of every workload. A switch after the final request does not count as completed construction

Engine	Generated (360)	External (180)	Fixed square (108)
A*	84; 33	51; 23	45; 22
Query-local LPA*	42; 119	33; 59	27; 55
Retained LPA*	45; 105	33; 48	27; 51

Table 9 Timing-rank stability in the original count-triggered comparisons. The numerator is persistent-only completion and the comparator is bundled completion. A median win uses the ratio of policy medians within a workload; a stable win or loss requires the same direction in all three original repetitions. The suffix includes completion and subsequent request work, excluding the common prefix

Engine	n	End-to-end ranking			Suffix ranking	
		Median wins	Stable wins	Mixed	Median wins	Stable wins
A*	60	55	46	13	60	54
Query-local LPA*	34	23	19	9	25	24
Retained LPA*	35	29	24	9	28	25

Stable end-to-end losses are 1, 6, and 2, respectively; stable suffix losses are 0, 6, and 3. All original timing totals remain included in Tables 5–7.

The matched-checkpoint follow-up isolates the suffix from this prefix variation. All 96,960 timed suffix request executions matched the reference

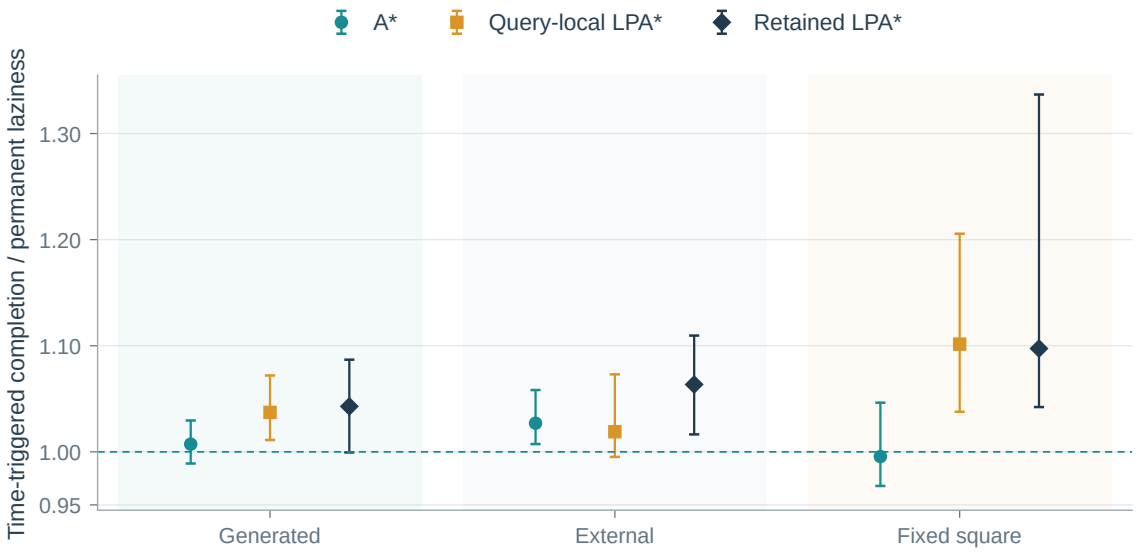


Fig. 16 Measured-time bundled completion relative to permanent laziness with the same search engine at forty requests. The three shaded lanes identify generated, external, and fixed-square settings; distinct markers identify the engines. Points are median paired workload ratios and whiskers are the 95% source-map-cluster bootstrap intervals. The dashed reference denotes equal time. Every workload enters the summaries, including those without a completion event. Rendered from the specified data or coordinates using ChatGPT-assisted Python scripting; see the AI declaration.

Table 10 Matched-checkpoint replay of persistent-only versus bundled completion. Ratios compare suffix times, including the common completion operation, over twenty paired repetitions per workload. Intervals are 95% source-map-cluster bootstrap intervals. "Faster" counts workload median ratios below one; "All 20" counts workloads favoring persistent-only in every paired repetition

Engine	Setting	n	Ratio [95% interval]	Faster	All 20
A*	Generated	28	0.666 [0.576, 0.771]	28	24
A*	External	17	0.421 [0.376, 0.512]	17	16
A*	Fixed square	15	0.708 [0.485, 0.811]	15	10
Query-local LPA*	Generated	14	0.869 [0.754, 1.280]	8	4
Query-local LPA*	External	11	0.604 [0.507, 0.679]	11	8
Query-local LPA*	Fixed square	9	0.658 [0.526, 1.422]	6	6
Retained LPA*	Generated	15	0.766 [0.720, 1.074]	10	7
Retained LPA*	External	11	0.588 [0.528, 0.694]	11	8
Retained LPA*	Fixed square	9	0.642 [0.529, 1.073]	6	6

The suffix begins at a checkpoint reconstructed from the archived request prefix. It excludes earlier construction and request costs, so these ratios do not replace end-to-end ratios. Generated, external, and derived-square settings remain separate.

route lengths, and all 1,163,520 checked operation-field values matched the corresponding archived execution. Persistent-only completion had a lower median suffix time on all 60 A* workloads, on 25 of 34 query-local LPA* workloads, and on 27 of 35 retained LPA* workloads (Table 10). It was faster in every one of the twenty paired repetitions on 50, 18, and 21 workloads, respectively. The direction of the median comparison agreed between the two execution-order strata on 59 of 60, 33 of 34, and 34 of 35 workloads.

In the replay of the adverse A* case, the persistent-only/bundled suffix-time ratio was 0.575, and persistent-only was faster in all twenty pairs.

The median paired suffix difference was −4.836 ms. The median completion-time ratio was 1.002, whereas the ratio for the remaining request work was 0.257. The identical completion operation was therefore similarly priced under this controlled schedule, and the subsequent attachment schedules had different costs. This follow-up establishes the post-completion comparison at checkpoints reconstructed from the archived request prefixes; the original full-sequence totals continue to measure whether persistent completion recovered its earlier investment relative to permanent laziness.

9 Discussion

At the forty-query horizon, delayed completion improved on both upfront completion and permanent laziness in a minority of workloads under the operation-cost model. The ordered-service criterion explains when this additional saving can occur. Every policy that ultimately completes pays for the same persistent checks. Waiting changes which requests are served on the incomplete graph and which are served after completion. Only the difference in terminal validation and search work can give one finite completion date an advantage over another.

9.1 Persistent validation cost relative to lazy continuation

The count visible to the controller, U_{k-1}, measures the checks it would have to perform immediately. Some of those candidates would be tested later even under continued lazy planning. The remainder, U_H, measures the extra persistent checks that completion adds over the whole sequence (Theorem 1). Their cost must be recovered through cheaper query service. A controller can observe the first count as it acts. The second can be measured once the lazy reference reaches the horizon.

The difference was substantial at the recorded triggers. Of the checks bought at completion, the median fraction that lazy continuation left untested at the horizon was 90.5% for A* and 72.3% for query-local LPA*. Hartmann and colleagues show how a planner can exploit selected validated paths and retain query-independent collision predicates [9, 29]. Full completion also acquires results outside the set reached by lazy planning. The size of that set difference determines the persistent-check cost that subsequent service savings must cover.

The spatial regimes give this cost a geometric interpretation. On localized requests, a planner could repeatedly reach its goals using a small fraction of the persistent candidates. Under the dispersed regime, lazy planning checked a larger fraction of the persistent candidates on the same maps. Every dispersed workload admitted a positive price at which completion paid, and all zero-frontier cases occurred in localized or shifted workloads. The regimes differed in both their sampling windows and their minimum start–goal separation. Their observed coverage determines the excess persistent checks on each sequence. The price frontier also depends on the differences in terminal checks and adjacency or repair visits.

9.2 Why the order of requests matters for delay

To improve on both endpoint policies, a delayed completion needs two service patterns on the same sequence. Early requests must collectively be cheaper to serve lazily than on the completed graph (Corollary 2). After the switch, the completed graph must save enough service work on the remaining requests to cover the extra persistent checks. A total for the entire sequence can settle the upfront comparison. Locating a profitable delay requires knowing when the savings occur.

The exact intervals identify sequences with both patterns. At horizon 40, a delayed date beat both endpoint policies at some positive price on one A* workload and 40 query-local LPA* workloads. Even on those workloads, its gain over the better endpoint policy was small. For most workloads at any given positive price, one endpoint already achieved the lowest modeled work. Where both endpoints were more expensive than a delayed date, the ordered service counts determine how much further a controller could gain through timing.

The engines can produce different ordered savings on the same requests. Query-local LPA* repairs distance labels within each request, so resolving a collision during a search can produce a different sequence of neighbor visits than beginning that request with known persistent costs. Mandalika and colleagues analyze how selective edge evaluation trades against search effort [27]. In the

present comparison those costs enter each request’s service difference, whose cumulative value determines the advantage of postponing completion.

9.3 Diagnosing a controller decision

At $\rho = 16$, nineteen triggered A* workloads offered no profitable completion date. The hindsight rule therefore chose permanent laziness on those workloads. On seventeen other A* workloads the chosen date lost, although a different date would have beaten permanent laziness. Only the latter losses could be reversed by rescheduling the same full-graph purchase. Applying the comparison also to workloads on which the rule stayed lazy identifies profitable purchases the controller missed (Figure 11).

The attachment decision calls for a different comparison. Benhamouche and colleagues distinguish static obstacle visibility from the changing connections to the start and goal [8]. Holding persistent completion fixed isolates the recurring checks that eager terminal attachment adds in Eq. (12). In the original timings lazy attachment was faster end to end on 55 of 60 triggered A* workloads. When both policies began at an identical checkpoint, lazy attachment had the lower median suffix time on all 60. Comparing persistent-only completion with permanent laziness instead gave eight original end-to-end wins on those same workloads. Here the comparison charges the policy for buying the persistent checks as well as serving subsequent requests.

9.4 Information lifetime determines what two traces can recover

A persistent collision result records a property of a fixed geometric segment. A distance label records a route cost from a particular start under the edge costs used in that search. LPA* repairs labels when these costs change [16], and lifelong lazy methods couple repair with selective validation [30]. Restarted A* and query-local LPA* both begin the next request with fresh labels. After completion, two executions can therefore reach that request through different histories and perform the same recorded service work. Retained LPA* carries earlier labels into the new query, so its service also depends on the state brought forward.

The comparisons of search implementation and retained state hold validation on demand throughout. On generated maps, query-local LPA* processed fewer queue vertices than restarted A* while making more adjacency/repair visits and taking longer. Updating a single distance label can require examining several neighbors in Eq. (2). Dellin and Srinivasa distinguish graph operations, edge evaluation, and data-structure maintenance [6]. The multi-trailer and dual-arm studies likewise illustrate how work surrounding an individual collision or path calculation contributes to planning time [37, 38]. Recording adjacency/repair visits separately from collision tests in Eq. (8) preserves their different costs.

9.5 Evaluating forecasts of completion value

A prospective controller has to estimate which checks future requests would demand and how much service work full validation would save. The two reference executions turn both quantities into observable outcomes after a workload ends. On held-out sequences, its decisions could then be scored against the best date or permanent laziness at the same operation prices, with separate counts of missed purchases, unprofitable purchases, and timing losses.

Selective completion would pose a different question about which persistent candidates to buy. Two policies could then finish the sequence with different sets of tested edges. Comparing such policies would require accounting for the acquired sets as well as the order of query-service savings.

10 Limitations

The geometric evaluation uses static planar maps with point translation or a fixed-orientation square represented through inflated rectangles. Boundary contact is allowed. Rotation, nonholonomic constraints, localization uncertainty, execution error, moving obstacles, and physical-robot performance are outside the evaluated setting. The independent reference establishes shortest-path agreement within the stored geometric representation and its coordinate precision.

The generated corpus uses eight and sixteen obstacles in four families. The twenty external maps come from one released planning project,

and the twelve square-robot settings derive from a subset of the generated maps. Separate groups and source-map clustering retain those relationships. The three seeded query regimes cover forty requests per workload. The reported frontier and delay frequencies describe those sequences. Testing all completion dates on the frozen workloads adds interventions on the same maps. Their operation counts can change with geometry, the query distribution, or the horizon.

The theorem covers a single irreversible completion of a fixed persistent candidate set. It assumes durable collision results, at most one test per persistent candidate, and completed-query service independent of preceding requests. Restarted A* and query-local LPA* satisfy the service condition here. Retained LPA* was evaluated through measured executions. Extending the reconstruction to retained search labels or a changing candidate graph requires a model of the state carried across queries. The hindsight comparison uses both traces through the horizon. Choosing online would require estimates of the remaining demand and service savings.

The modeled work assigns constant prices to persistent checks, terminal checks, and adjacency/repair visits. The main price analysis assigns both kinds of collision checks the same price. Their separate stored coefficients also permit different prices. Queue operations, memory allocation, variation in geometric test cost, and timer overhead contribute to elapsed time. The continuous frontiers and gain suprema therefore describe weighted operation counts, with the suprema approached as $\rho \to 0^+$. Runtime comparisons use the separately measured elapsed times.

Runtime results concern the deposited Python implementations, geometric routine, fixed adjacency representation, and recorded hosts. The original metadata identifies Python 3.13.5 and the operating platform; the processor model and NumPy version were not captured contemporaneously. The executed source and input hashes match the archived files. A later regression-driver revision, the checkpoint measurement environment, and the present operation-count environment are identified separately. The search engines are matched implementations, and performance can change with optimized queue, dictionary, predicate, or graph-storage choices. Three original timing repetitions reveal common-prefix variability; the twenty-pair checkpoint study controls the attachment comparison on one host and the original triggered states.

The count coefficient is inherited from the preceding experiment, and the measured-time signal uses observed mean check duration. Neither was selected using the two-trace outcome. The study follows an implementation diagnosis and supplies controlled comparisons rather than an independently held-out controller-selection trial. Triggered-only statistics describe the intervention condition; the all-workload and all-position results retain the complete population.

11 Conclusion

On a fixed roadmap with durable collision results and history-independent completed-query service, two executions of the same ordered requests determine the modeled work of every irreversible full-completion decision. A permanently lazy execution identifies the persistent checks it would leave untested at the horizon. An execution that completes upfront supplies the terminal and search work of every post-completion query. A single pass finds the date with the greatest remaining service saving and tests whether that saving pays for the extra checks.

Direct execution verified this reconstruction for all 17,280 completion schedules across 216 forty-query workloads and two search engines. At horizon 40 every finite completion incurred more collision checks than permanent laziness, placing a finite upper bound on the price at which those checks could pay. With positive prices evaluated separately for each workload, some completion paid on 151 of 216 A* workloads and 183 of 216 query-local LPA* workloads. Delay strictly improved on both upfront completion and permanent laziness on one and 40 of those 216 workloads per engine, respectively. Its largest limiting gains over the better endpoint policy were 0.258% and 3.361%, respectively. On these sequences the extra value available from choosing an interior date was therefore small.

The measured timings distinguish search implementation, label retention, and terminal attachment. Under permanent lazy validation, query-local LPA* took longer than restarted A* at the group medians, and retaining its labels across requests further increased median paired time in all three corpus groups. Once both policies had

committed to completing the graph, lazy terminal attachment gave a lower median suffix time on all 60 matched triggered A* workloads. When continuing lazily was the alternative, persistent-only completion gave a lower original end-to-end median on eight of the same workloads. The two-trace reference extends the acquisition comparison across all dates under declared operation prices, so a controller's loss can be traced to the purchase itself or to its timing.

References

[1] Siegwart, R., Nourbakhsh, I.R.: Introduction to Autonomous Mobile Robots. MIT Press, Cambridge, MA (2004)

[2] Macenski, S., Martín, F., White, R., Ginés Clavero, J.: The marathon 2: A navigation system. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2718–2725 (2020). https://doi.org/10.1109/IROS45743.2020.9341207

[3] Kavraki, L.E., Švestka, P., Latombe, J.-C., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation **12**(4), 566–580 (1996) https://doi.org/10.1109/70.508439

[4] LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006). https://doi.org/10.1017/CBO9780511546877 . https://lavalle.pl/planning/

[5] Bohlin, R., Kavraki, L.E.: Path planning using lazy PRM. In: Proceedings of the IEEE International Conference on Robotics and Automation, vol. 1, pp. 521–528 (2000). https://doi.org/10.1109/ROBOT.2000.844107

[6] Dellin, C., Srinivasa, S.: A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 26, pp. 459–467 (2016). https://doi.org/10.1609/icaps.v26i1.13788

[7] Song, G., Thomas, S., Amato, N.M.: A general framework for PRM motion planning. In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation, vol. 3, pp. 4445–4450 (2003). https://doi.org/10.1109/ROBOT.2003.1242289

[8] Benhamouche, O., Asani, Z., Cotorruelo, A., Wauthion, B., Vanderborght, B., Garone, E.: On an almost closed-form fast generation of Euclidean shortest path trajectories in cluttered environments. In: 2025 11th International Conference on Automation, Robotics, and Applications (ICARA), pp. 399–403 (2025). https://doi.org/10.1109/ICARA64554.2025.10977725

[9] Hartmann, V.N., Strub, M.P., Toussaint, M., Gammell, J.D.: Effort informed roadmaps (EIRM*): Efficient asymptotically optimal multiquery planning by actively reusing validation effort. In: Billard, A., Asfour, T., Khatib, O. (eds.) Robotics Research, pp. 555–571. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-25555-7_37

[10] Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: 2007 IEEE 23rd International Conference on Data Engineering, pp. 826–835 (2007). https://doi.org/10.1109/ICDE.2007.367928

[11] Lozano-Pérez, T.: Spatial planning: A configuration space approach. IEEE Transactions on Computers **C-32**(2), 108–120 (1983) https://doi.org/10.1109/TC.1983.1676196

[12] Lozano-Pérez, T., Wesley, M.A.: An algorithm for planning collision-free paths among polyhedral obstacles. Commun. ACM **22**(10), 560–570 (1979) https://doi.org/10.1145/359156.359164

[13] Liu, Y.-H., Arimoto, S.: Path planning using a tangent graph for mobile robots among polygonal and curved obstacles. Int. J. Robot. Res. **11**(4), 376–382 (1992) https://doi.org/10.1177/027836499201100409

[14] Rohnert, H.: Shortest paths in the plane with convex polygonal obstacles. Inf. Process. Lett. **23**, 71–76 (1986) https://doi.org/

10.1016/0020-0190(86)90045-1

[15] Oh, S., Leong, H.W.: Edge N-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths. In: Proceedings of the Tenth International Symposium on Combinatorial Search, pp. 64–72 (2017). https://doi.org/10.1609/socs.v8i1.18427

[16] Koenig, S., Likhachev, M., Furcy, D.: Lifelong planning A*. Artificial Intelligence **155**(1–2), 93–146 (2004) https://doi.org/10.1016/j.artint.2003.12.001

[17] Harabor, D., Grastien, A., Öz, D., Aksakalli, V.: Optimal any-angle pathfinding in practice. J. Artif. Intell. Res. **56**, 89–118 (2016) https://doi.org/10.1613/jair.5007

[18] Cui, M.L., Harabor, D.D., Grastien, A.: Compromise-free pathfinding on a navigation mesh. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, pp. 496–502 (2017). https://doi.org/10.24963/ijcai.2017/70

[19] Shen, B., Cheema, M.A., Harabor, D.D., Stuckey, P.J.: Fast optimal and bounded suboptimal euclidean pathfinding. Artif. Intell. **302**, 103624 (2022) https://doi.org/10.1016/j.artint.2021.103624

[20] Jiang, K., Seneviratne, L.D., Earles, S.W.E.: Time-optimal smooth-path motion planning for a mobile robot with kinematic constraints. Robotica **15**(5), 547–553 (1997) https://doi.org/10.1017/S0263574797000635

[21] Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. **18**(3), 305–363 (1997) https://doi.org/10.1007/PL00009321

[22] Li, Y., Li, D., Maple, C., Yue, Y., Oyekan, J.: K-order surrounding roadmaps path planner for robot path planning. Journal of Intelligent & Robotic Systems **75**(3–4), 493–516 (2014) https://doi.org/10.1007/s10846-013-9861-3

[23] Zhang, D., Ye, Z., Yu, J.: Asymptotically-optimal multi-query path planning for a polygonal robot. In: 2025 IEEE International Conference on Robotics and Automation, pp. 1780–1786 (2025). https://doi.org/10.1109/ICRA55743.2025.11127599

[24] Sánchez, G., Latombe, J.-C.: On delaying collision checking in prm planning: Application to multi-robot coordination. The International Journal of Robotics Research **21**(1), 5–26 (2002) https://doi.org/10.1177/027836402320556458

[25] Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research **30**(7), 846–894 (2011) https://doi.org/10.1177/0278364911406761

[26] Hauser, K.: Lazy collision checking in asymptotically-optimal motion planning. In: 2015 IEEE International Conference on Robotics and Automation, pp. 2951–2957 (2015). https://doi.org/10.1109/ICRA.2015.7139603

[27] Mandalika, A., Choudhury, S., Salzman, O., Srinivasa, S.: Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 29, pp. 745–753 (2019). https://doi.org/10.1609/icaps.v29i1.3543

[28] Haghtalab, N., Mackenzie, S., Procaccia, A.D., Salzman, O., Srinivasa, S.S.: The provable virtue of laziness in motion planning. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 28, pp. 106–113 (2018). https://doi.org/10.1609/icaps.v28i1.13879

[29] Hartmann, V.N., Ortiz-Haro, J., Toussaint, M.: Efficient path planning in manipulation planning problems by actively reusing validation effort. In: 2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2822–2829 (2023). https://doi.org/10.1109/IROS55552.2023.10341264

[30] Lim, J., Ghanei, M., Lawson, R.C., Srinivasa, S., Tsiotras, P.: Lazy incremental search for

efficient replanning with bounded suboptimality guarantees. The International Journal of Robotics Research **43**(8), 1175–1207 (2024) https://doi.org/10.1177/02783649241227869

[31] Díaz Ramos, G., Arikan, T., Baraniuk, R.G.: Deep learning-based framework for efficient multigoal shortest path planning in indoor environments. Robotica **44**, 2203–2233 (2026) https://doi.org/10.1017/S0263574726103932

[32] Guarda, F., Palli, G.: ROTOTILLER: Rolling optimization of tasks and online trajectories for integrated multi-robot task allocation and fleet navigation. Robotics **15**(10), 189 (2026) https://doi.org/10.3390/robotics15100189

[33] Pasricha, A., Roncone, A.: The virtues of laziness: Multi-query kinodynamic motion planning with lazy methods. In: 2024 IEEE International Conference on Robotics and Automation, pp. 14286–14292 (2024). https://doi.org/10.1109/ICRA57147.2024.10611326

[34] Zhang, L., Guo, X., Cai, K., Wu, F., Miao, Z., Bing, Z., Knoll, A.: Efficient motion planning in narrow passages via long-short edge coupling and stitching sampling. Journal of Intelligent & Robotic Systems (2026) https://doi.org/10.1007/s10846-026-02484-9 . Published online 13 November 2026, article in press

[35] Zhao, D., Huang, G., Liu, M., Ren, T., Tang, Q.: Path planning for robotic arm in catenary maintenance: An improved RRT algorithm based on obstacle node. Journal of Intelligent & Robotic Systems **112**(4) (2026) https://doi.org/10.1007/s10846-026-02425-6

[36] Huang, L., Yu, J., Wang, J., Jing, X.: Asymptotically optimal lifelong planning with lazy edge evaluation under expensive collision checks. IEEE Transactions on Automation Science and Engineering **23**, 401–416 (2026) https://doi.org/10.1109/TASE.2025.3636173 . Published online 24 November 2025

[37] Dahlmann, J., Völz, A., Graichen, K.: Global motion planning for multi-trailer vehicles in partially structured environments. Journal of Intelligent & Robotic Systems **111**(4) (2025) https://doi.org/10.1007/s10846-025-02330-4

[38] Wittmann, J., Ochsenfarth, F., Sonneville, V., Rixen, D.: Centralized vs. decoupled dual-arm planning taking into account path quality. Journal of Intelligent & Robotic Systems **110**(4) (2024) https://doi.org/10.1007/s10846-024-02175-3

[39] Moll, M., Şucan, I.A., Kavraki, L.E.: Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. IEEE Robot. Autom. Mag. **22**(3), 96–102 (2015) https://doi.org/10.1109/MRA.2015.2448276

[40] Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968) https://doi.org/10.1109/TSSC.1968.300136

[41] Mouawad, M.: Reduced and Improved Visibility Graph for Robot Path Planning. Zenodo. Technical note, version 1.1.0 (2026). https://doi.org/10.5281/zenodo.20682235 . https://zenodo.org/records/20682235

Statements and Declarations

Funding

No funding was received for this research.

Competing interests

The author declares no competing interests.

Author contribution

Meshaal Mouawad: conceptualization, methodology, investigation, software, formal analysis, visualization, and manuscript preparation.

Data availability

Online Resource 1 preserves the frozen `ROBOTICA-PQV-v10` study: map and query inputs, original measurements and routes, independent geometric references, timing-phase calculations, and matched-checkpoint records. Online Resource

2 contains the JIRS-TWO-TRACE-v1.1 extension: reference traces, directly checked completion coefficients, exact cost intervals, continuous gain calculations, and verification and publication outputs. Both resources include file manifests and source and input hashes. The extension retains the original measurements unchanged.

Code availability and project provenance

Both resources include implementation, analysis, and reproduction instructions. Recorded environments distinguish the original timing run, checkpoint follow-up, and current operation-count study. The extension uses the unchanged archived planner and geometry routines.

The earlier MATLAB project is available in the public project repository, established on 17 September 2021. It credits Aditya Jain's code and Meshaal Mouawad's subsequent edits; the historical attribution and license remain with those materials. The associated technical note documents that project [41]. The present Python study and its extension retain separate execution records.

AI declaration

ChatGPT assisted with implementation and analysis scripting, source discovery and synthesis, manuscript revision, pseudocode, and vector-artwork preparation during 4–5 and 18 November 2026. The current revision used GPT-6 Astra Pro and GPT-6.2 agent models for mathematical and source checks, direct-completion scripting, analysis, and production. The deposited records distinguish source checks, independent analytical checks, and computational comparisons.

All figures are programmatically rendered from declared coordinates, editable specifications, or recorded data using ChatGPT-assisted Python scripting, Matplotlib, and vector-rendering tools. Captions identify the assistance and distinguish constructed illustrations from empirical observations. The supplied PDF and SVG files retain the explicit geometry and numerical inputs.

Supplementary information

Online Resource 1 (ESM_1.zip): frozen fourteen-mode study, independent geometric validation, timing-phase analysis, and matched-checkpoint follow-up, release ROBOTICA-PQV-v10. Online Resource 2 (ESM_2.zip): protocol, two-trace computations, direct completion-position verification, exact cost frontiers, continuous delay-gain analysis, and reproducible publication outputs, release JIRS-TWO-TRACE-v1.1.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- ESM2.zip
- ESM1.zip