ATLAS: Scaffold-Free Algorithm Synthesis by LLMs via Embedding-Guided Quality-Diversity Search

Danial Yazdani, Mohammad Nabi Omidvar, Yuan Sun, Maksud Ibrahimov, and Xiaodong Li

Abstract

Most LLM-based automated algorithm design methods optimize a designated component within a human-specified scaffold, fixing overall organization and component interactions. We present ATLAS, an embedding-guided quality-diversity framework for scaffold-free full-algorithm synthesis in combinatorial optimization. The problem specification supplies objectives and constraints; a minimal I/O interface fixes only instance and solution formats; the LLM chooses and restructures components, interactions, and control flow. This freedom enlarges the search space, risking invalid candidates and premature convergence to one design region. ATLAS independently detects execution, interface, and feasibility failures, recomputes objectives, and applies error-conditioned repair; similarity-based archive management preserves algorithms across embedding-space regions to counter premature convergence. Its three-layer search refines the best design, gives other regions dedicated refinement opportunities, and performs cross-region synthesis to recombine components and their interactions. Across four NP-hard problems, ATLAS outperforms several state-of-the-art component-synthesis methods and a matched full-synthesis baseline while remaining competitive with strong human-designed algorithms. One ATLAS run retains several algorithms with comparable performance from distinct embedding-space regions rather than a single design. Code inspection finds that these multi-component designs differ in their primary construction or global-search backbone. Our results suggest that embedding-guided quality-diversity search can make the enlarged full-algorithm design space practically searchable. Source code and exact executable prompts are available at https://github.com/Danial-Yazdani/ATLAS.

Index Terms

Automated algorithm design, combinatorial optimization, large language models, multimodal optimization, quality-diversity optimization.

I. Introduction

Optimization algorithms are fundamental tools for applications in logistics, scheduling, and resource allocation [47, 35]. Designing such algorithms typically requires extensive expert knowledge and problem-specific engineering effort. Recent progress in automated algorithm design has shown that large language models (LLMs) can synthesize heuristic components through iterative search [54, 28]. Methods such as FunSearch (Searching in the Function Space) [36], Evolution of Heuristics (EoH) [25], and Reflective Evolution (ReEvo) [57] evolve algorithmic functions (priority selectors, construction heuristics, scoring functions) that operate as components within predefined scaffolds. For these methods, the synthesized component occupies a designated role inside user-defined control flow; the surrounding framework supplies the remaining end-to-end algorithmic machinery and may perform solution-construction or feasibility-preserving operations. Although this scaffolding makes synthesis and evaluation manageable, it imposes much of the algorithmic architecture in advance and restricts the LLM from going beyond the designated interface to add, remove, reorder, or jointly redesign components and their interactions.

Full-algorithm synthesis removes this restriction by placing complete algorithms, rather than only designated components, under LLM-guided search. This relaxation enables exploration of different component inventories and algorithmic organizations, including construction heuristics, local search, metaheuristics, and hybrid pipelines. Existing approaches instantiate it with different boundaries: LLaMEA [48] generates complete metaheuristics under a box-constrained black-box interface, whereas AlphaEvolve [33] evolves user-marked regions of supplied programs. This greater design freedom expands the search space into a larger, more heterogeneous landscape that is harder to navigate and exposes generated algorithms to potential failures in execution, interface compliance, solution construction, and constraint handling. This freedom can also increase synthesis cost because it admits more complex, multi-stage algorithms whose execution and iterative refinement require greater computational effort. However, the main advantage is the resulting design freedom: without a user-prescribed internal architecture, the LLM can explore alternative component inventories, component interactions, and control flows that a fixed scaffold would otherwise exclude.

The resulting algorithm landscape is larger and can also be multimodal: competitive algorithms with different internal organizations may occupy distinct regions of the selected representation. Niching methods in multimodal optimization seek to locate and preserve multiple optima or modes rather than collapsing to a single solution [23]. This motivates full-algorithm

Danial Yazdani and Xiaodong Li are with the School of Computing Technologies, RMIT University, Melbourne, VIC, Australia (e-mails: danial.yazdani@gmail.com; xiaodong.li@rmit.edu.au). Mohammad Nabi Omidvar is with the School of Computing and Leeds University Business School, University of Leeds, Leeds, UK (e-mail: M.N.Omidvar@leeds.ac.uk). Yuan Sun is with La Trobe Business School, La Trobe University, Melbourne, VIC, Australia (e-mail: Yuan.Sun@latrobe.edu.au). Maksud Ibrahimov is with Effective AI, Melbourne, VIC, Australia (e-mail: maksud@effective-ai.com.au).

This work was supported by RMIT RACE through the provision of API access.

Corresponding author: Xiaodong Li.

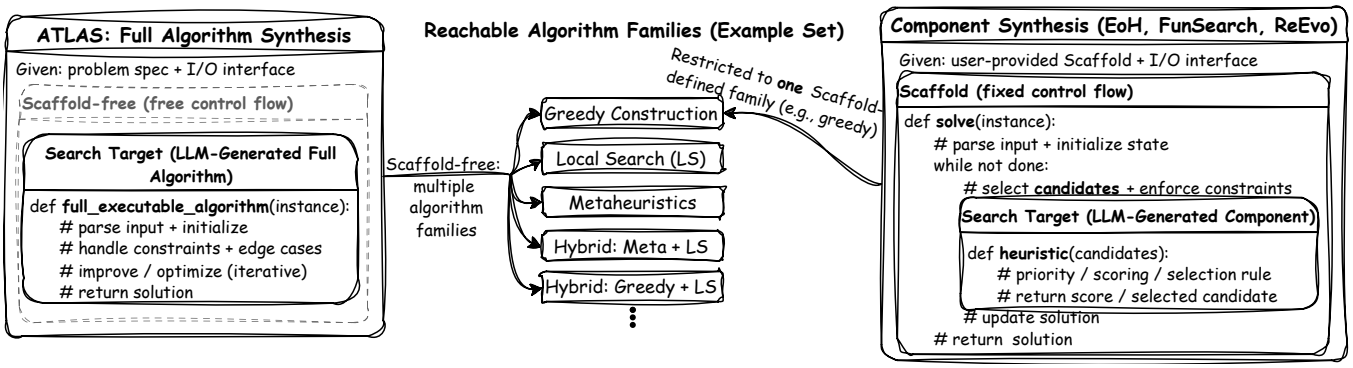


Fig. 1: *Component-based vs. scaffold-free full-algorithm synthesis.* ATLAS synthesizes a complete optimization algorithm that receives problem instances and returns solutions through a fixed external I/O interface, without a user-provided internal algorithmic scaffold; component methods synthesize a designated function within user-provided control flow.

search that preserves several competitive regions while refining promising designs rather than committing too early to one incumbent.

In LLM-driven synthesis, archive diversity has an additional operational role. Reference-conditioned operators can receive the complete source of algorithms from different regions and reason about their components and interactions when constructing a new algorithm. Coverage therefore supplies varied source material for cross-region synthesis, not only insurance against premature convergence: the LLM can select, discard, adapt, or combine mechanisms from different inputs into a new end-to-end organization.

To navigate this landscape, we introduce ATLAS (Archive-based Three-Layer Algorithm Synthesis), an embedding-guided quality-diversity framework for full-algorithm synthesis in combinatorial optimization. ATLAS maintains a coverage-preserving archive, or *semantic repertoire*,[1] of executable algorithms organized in a pretrained embedding space [7, 3, 58]. Embedding distance organizes retrieval, clustering, redundancy control, and reference selection without requiring hand-specified behavior descriptors. The induced regions are recomputed as the archive develops rather than fixed in advance. A three-layer strategy concentrates refinement around the current best algorithm and gives representatives of other regions dedicated refinement opportunities. Cross-region synthesis supplies the LLM with components and functions from multiple input algorithms. The LLM can then reason about their interactions and reorganize them into new hybrid algorithms.

At its core, ATLAS is scaffold-free. The problem definition supplies the objective and constraints as requirements common to any optimization algorithm, while ATLAS fixes only a minimal problem-specific I/O interface and prescribes no internal component inventory, function-role partition, or control flow. The LLM therefore controls the complete algorithmic artifact and is responsible for end-to-end solution construction and constraint handling. ATLAS's external evaluator does not supply or complete this logic, and it does not make returned solutions feasible; it executes candidates under the common resource protocol, verifies returned solutions, and independently recomputes their objective values. Because this responsibility widens the failure surface to malformed and infeasible outputs as well as execution errors, error-conditioned REPAIR is treated as a search operator that regenerates the candidate algorithm from classified failure evidence.

Figure 1 contrasts scaffold-constrained component synthesis with ATLAS's full-algorithm search under its minimal external I/O interface. ATLAS operationalizes this search through prompts that communicate the problem specification and I/O requirements, together with LLM-driven full-algorithm operators that receive complete source code for refinement (IMPROVE, TUNE, SIMPLIFY), synthesis (COMBINE, DIVERGE), and error-conditioned recovery (REPAIR) [41]. Similarity-based retrieval and redundancy-aware pruning preserve coverage across embedding-separated algorithm regions, providing alternatives for targeted refinement and recombination. The supporting execution system evaluates candidates under runtime and memory controls and records code, lineage, outcomes, resource use, and failure diagnostics.

Our contributions are as follows.

- **Scaffold-Free Full-Algorithm Synthesis:** We introduce ATLAS, a framework for scaffold-free full-algorithm synthesis in combinatorial optimization. We provide the problem definition, including its objective and constraints, and specify the callable entry point and instance/solution I/O formats. Beyond these necessary problem and interface requirements, the LLM is free to design the complete algorithm, choose its components, coordinate their interactions, and organize its end-to-end workflow without any user-prescribed restriction on its internal structure.
- **Embedding-Guided Quality-Diversity and Multimodal Search:** We propose a coverage-preserving semantic repertoire organized in pretrained embedding space. Embedding-based retrieval, clustering, and redundancy control mitigate premature

[1]Here and throughout, *semantic* denotes similarity in the pretrained embedding of an algorithm's name, description, and preprocessed source code, not program semantics defined by execution behavior. Embedding-space *regions* or *clusters* denote groups induced by this representation. Mechanism-oriented family labels are used only for the author-annotated embedding benchmark or for illustrative representatives examined separately through code inspection.

convergence and support multimodal search in the selected representation, retaining multiple near-best algorithms while search continues to improve the strongest designs. The preserved alternatives also provide semantically varied references for cross-region synthesis, so coverage contributes to search progress as well as final repertoire quality.

- **Hierarchical Three-Layer Search:** We develop a three-layer resource-allocation strategy over the clustered repertoire: Layer 1 targets the current best region, Layer 2 gives non-elite region representatives dedicated refinement opportunities, and Layer 3 recombines components and their interactions across regions to discover new organizations and hybrids.
- **Independent Evaluation and Failure-Conditioned Recovery:** Because generated algorithms are responsible for solution construction and constraint handling, evaluation must detect a wider failure surface that includes execution, interface, malformed-output, and feasibility failures. ATLAS independently verifies returned solutions and recomputes objectives, applies an all-or-nothing validity rule that records failures rather than converting them into finite penalties, and routes classified evidence to error-conditioned REPAIR. The accompanying implementation provides problem- and interface-aware prompts, worker-process parallel evaluation with runtime and memory controls, and detailed lineage, resource, and failure records.
- **Empirical Design Insights for LLM-Based Full Synthesis:** Controlled studies examine artifact scope, representation, initialization, grouping, and search allocation. They show that moving from component synthesis to full-algorithm synthesis through the ATLAS framework improves performance in most tested settings and provide practical guidance for the design of LLM-based full-algorithm synthesis systems.

We validate these contributions across four NP-hard combinatorial optimization benchmarks. ATLAS outperforms several state-of-the-art component-synthesis methods and the matched full-synthesis baseline, remains competitive with strong human-designed, domain-specific algorithms, and retains multiple competitive algorithms in separate archive regions within a run, consistent with the intended multimodal search behavior.

The remainder of this paper is organized as follows. Section II reviews LLM-based component and full-algorithm synthesis together with relevant quality-diversity research; Section III presents the ATLAS formulation and search framework; Section IV describes the experimental protocol and results; and Section V concludes the paper. The supplementary material provides extended related work, implementation and evaluation details, problem definitions and instance generation, pseudocode and evaluation protocols, configurations and baseline specifications, complete protocols and analyses, additional experiments, and descriptions of representative synthesized algorithms.

II. RELATED WORK

This section reviews work related to the LLM-based automated algorithm design aspects of ATLAS. We focus on component and full-algorithm synthesis, together with optimization methods that act as meta-optimizers over the generated-algorithm space. The concise discussion here highlights the distinctions needed for the present work; a comprehensive taxonomic and method-by-method treatment is provided in Appendices A-A and A-B.

a) LLM-Driven Component Synthesis: Recent methods use LLMs to synthesize *algorithmic components* that occupy predefined roles within user-specified algorithmic control flow. Representative examples include FUNSEARCH [36], EOH [25], REEVO [57], HSEvo [9], MCTS-AHD [61], CALM [18], HEURAGENIX [55], MEOH [56], and EOH-S [27]. These methods differ in search, reflection, and diversity mechanisms, but their generated functions, including priority rules, construction heuristics, and operator-selection policies, do not replace the surrounding algorithmic scaffold. A callable interface alone does not imply component synthesis; the distinction is whether the generated artifact owns the end-to-end algorithmic logic or fills a role within predefined control flow.

b) Full Algorithm Synthesis: LLAMEA [48] generates complete Python metaheuristics for box-constrained continuous black-box optimization and refines one complete algorithm at a time using execution feedback. ALPHAEVOLVE [33] performs patch-based evolution over user-provided codebases with annotated editable regions. A2DEPT [4], developed concurrently with ATLAS, evolves complete combinatorial-optimization algorithms using a program-lineage tree, an explicit function-level representation, hierarchical micro/macro operators, and dependency-aware repair. These methods all move beyond single-component synthesis but impose different domains, representations, and structural priors. ATLAS combines a problem specification and minimal combinatorial I/O interface with scaffold-free full-algorithm operators that receive complete source code: it prescribes neither a component inventory, function-role partition, nor user-provided algorithmic control flow, and it organizes search to preserve coverage across embedding-space regions.

c) Meta-Optimization and Quality-Diversity Search: The optimization method that searches over generated algorithms acts as a meta-optimizer and must balance exploitation of strong designs against exploration and preservation of useful alternatives. Prior LLM-based methods retain alternatives through mechanisms such as island models [36], tree search guided by Upper Confidence Bounds applied to Trees (UCT), which balances revisiting strong nodes with exploring under-sampled branches [61], program-lineage trees [4], and explicit objective- or instance-space diversity criteria [9, 56, 27]. Quality-diversity methods preserve collections of high-performing candidates across a chosen descriptor space [7, 3]. ATLAS applies this principle through an embedding-based representation: distance organizes retrieval, clustering, redundancy-aware pruning, and the selection of inputs for cross-region synthesis. The resulting repertoire reduces premature convergence and supplies distinct regions for continued refinement. It also allows the LLM to draw on components and their interactions from algorithms in different regions when constructing hybrids.

III. ATLAS

This section presents ATLAS and explains how it navigates the enlarged search space created by scaffold-free full-algorithm synthesis. We first clarify the responsibilities of the generated algorithm and external evaluator, then describe the embedding-guided archive, semantic operators, initialization, and three-layer search that jointly preserve and refine multiple algorithmic regions.

A. Overview

1) ATLAS Setting: ATLAS synthesizes complete, executable optimization algorithms for a specified combinatorial optimization problem. Each generated algorithm receives an instance through a fixed input interface and must return a structured solution. Within these external boundaries, it is responsible for solution construction, constraint handling, internal state management, and its own stopping logic within the resource limits. ATLAS supplies no user-written algorithm skeleton, prescribed component inventory, function-role partition, or internal control flow. Consequently, candidate designs may vary their component organization, interactions, and control flow instead of varying only a designated component within a fixed skeleton.

The synthesis setting fixes three external elements, none of which prescribes internal algorithm structure:

- The *problem specification* states the instance semantics, objective, constraints, and valid-solution requirements. These are properties of the optimization problem and apply to every algorithm regardless of how it was designed.
- A minimal algorithm-facing *I/O interface* specifies the callable entry point and how problem instances and returned solutions are represented.
- The *experimental protocol* specifies the execution environment and resource limits used uniformly within each comparison.

After executing a generated algorithm under this protocol, the external evaluator verifies the feasibility of the returned solution and independently recomputes its objective; it does not complete, repair, or make the solution feasible on behalf of the generated algorithm. Scaffold freedom therefore concerns the absence of a prescribed algorithm decomposition, not the absence of a problem specification, I/O interface, or common evaluation protocol. We refer to this setting as *scaffold-free full-algorithm synthesis under a fixed I/O interface*. It differs from component synthesis [36, 25, 57], which optimizes functions inside fixed surrounding control flow, and from scaffolded full-algorithm evolution as instantiated by AlphaEvolve [33], where evolution targets user-marked blocks within an externally supplied program.

An explicit function-role decomposition is not required to represent or revise multi-component designs in ATLAS. Each reference-based operator receives the complete source of every selected algorithm, including helper definitions and the calls linking them. ATLAS evaluates and archives complete algorithms, while the operator suite spans different edit scopes: Tune preserves structure, Improve preserves the core approach, and Combine and Diverge can coordinate changes across functions or restructure the end-to-end workflow.

Accordingly, *full* refers to the scope of algorithm-design responsibility rather than to source-code length or to treating an algorithm as a monolithic component. Code inspection of four released illustrative algorithms shows coordinated multi-function pipelines. In each inspected algorithm, construction or seeding feeds problem-specific improvement and diversification mechanisms, together with an explicit policy for accepting non-improving candidates. These examples confirm that multi-component designs can arise in the displayed runs without a prescribed function-role decomposition; they do not quantify how frequently such structures occur across the complete archives.

This evidence does not imply that explicit structural guidance can never improve reliability or search efficiency. ATLAS can also use operator instructions that focus on a selected component or interaction without imposing a global component decomposition; such policies are not compared here.

2) Problem Formulation: Given a problem $\mathcal{P}$ and training set $\mathcal{I}_{\text{train}} = \{x_1, \ldots, x_m\} \subset \mathcal{I}$, the objective is to find an algorithm minimizing empirical cost:

$$a_{\text{opt}} = \arg\min_{a \in \mathcal{V}} \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(a(x_i)), \tag{1}$$

where $\mathcal{V}$ is the set of valid executable algorithms and $a(x_i) \in \mathcal{S}$ is the solution produced for instance x_i. Generalization is evaluated on separate test instances.

3) ATLAS Approach: ATLAS maintains an archive $\mathcal{A} \subseteq \mathcal{V}$ of valid algorithms and iteratively refines it to approximate a_{opt}. At each iteration, the current best algorithm is $a^* = \arg\min_{a \in \mathcal{A}} J(a)$, where $J(a) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(a(x_i))$ is the average training cost. ATLAS is designed to address two challenges: (i) the risk that fitness pressure concentrates search around a small number of early designs, and (ii) inefficient resource allocation, which under-refines promising but currently weaker regions. To address these challenges, ATLAS organizes the archive in embedding space, clusters it into operational semantic regions, and allocates search effort through a three-layer strategy for local refinement, representative-level maturation, and cross-cluster synthesis.

As illustrated in Figure 2, ATLAS iterates over the following cycle: evaluate and insert valid algorithms into $\mathcal{A}$, organize and cluster the archive in embedding space, and generate new candidates through three-layer search. Algorithm 1 summarizes this orchestration. Appendix B-D provides the connected module-level procedures, detailed in Algorithms B.1–B.5.

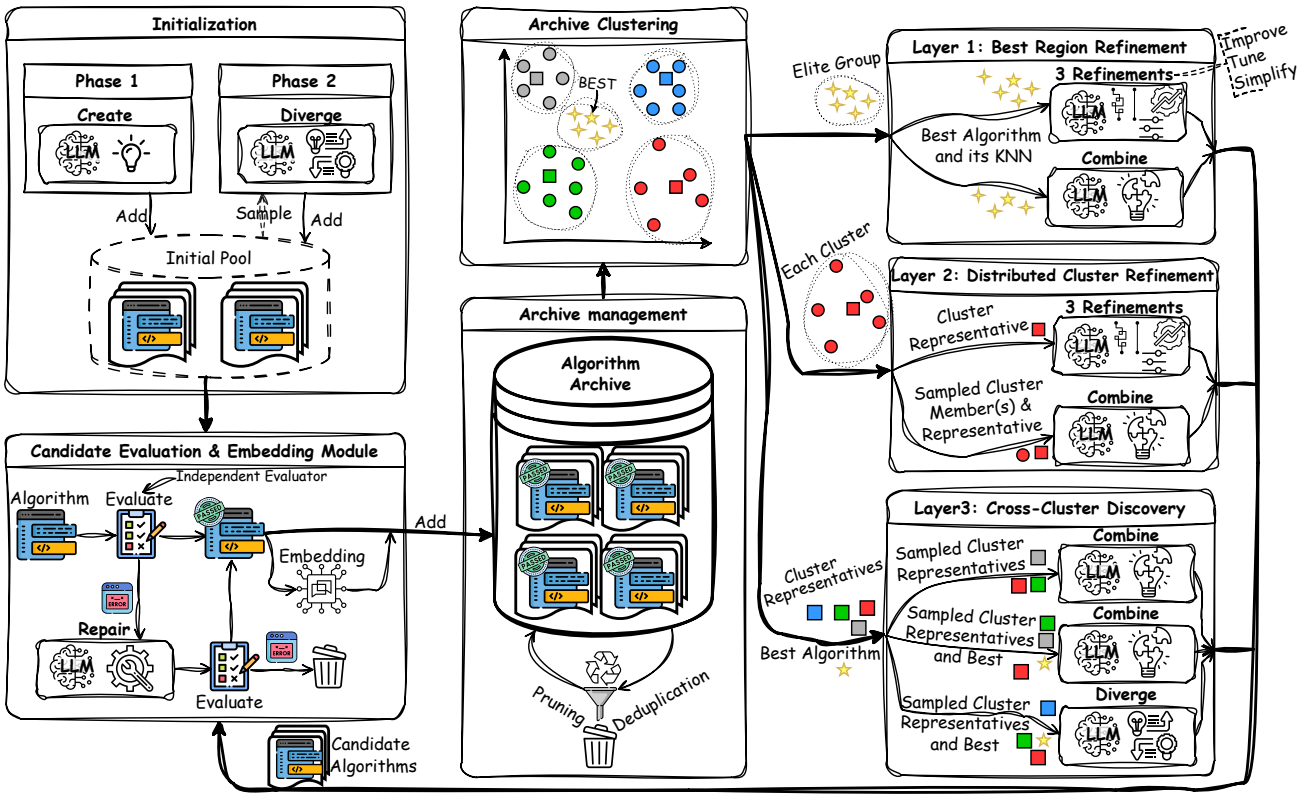


Fig. 2: *Overview of ATLAS.* **(Top-Left)** *Initialization* generates diverse candidates via Create and DIVERGE operators. **(Bottom-Left)** *Candidate Processing* evaluates, repairs, and embeds generated algorithms. **(Bottom-Center)** *Archive Management* performs deduplication and capacity-constrained pruning. **(Top-Center)** *Archive Clustering* organizes algorithms by similarity under the selected embedding into the elite region $\mathcal{G}^*$ (gold stars) and semantic clusters (colored circles); cluster representatives are shown as squares. **(Right)** *Three-Layer Search* allocates resources hierarchically: Layer 1 refines the best algorithm (a^*) and its neighborhood; Layer 2 refines cluster representatives in isolation; Layer 3 performs cross-cluster discovery. Candidates cycle through processing and archive integration to enable systematic refinement across multiple represented regions.

The framework comprises five components: embedding-based similarity (§III-B) for reference selection and structure discovery, a coverage-preserving archive (§III-C) to retain diverse regions, semantic operators (§III-D) to generate and repair candidate algorithms, initialization (§III-E) to establish broad initial coverage, and three-layer search (§III-F) to allocate search effort across regions.

B. Embedding-Based Similarity

ATLAS structures the algorithm search space using pretrained embeddings, providing a semantic coordinate system for reference algorithm selection, region discovery, and archive management.

1) Algorithm Representation: Each algorithm is represented as a tuple $a = (n, d, c)$ comprising: (i) *Name* n, (ii) *Description* d, and (iii) *Source Code* c (a standalone executable program). These components are generated jointly by the LLM during synthesis (§III-D).

2) Embedding Strategy: We represent algorithms using mGTE-large-en-v1.5 [58], concatenating the name, description, and preprocessed source code into a single text input. This configuration was selected through systematic evaluation of multiple embedding strategies (Appendix C-A).

3) Similarity Infrastructure: Let $\mathbf{e}_i$ denote the embedding of algorithm a_i. We maintain an incremental distance matrix $\mathbf{D} \in \mathbb{R}^{|\mathcal{A}|\times|\mathcal{A}|}$ with entries $D_{ij} = 1 - \cos(\mathbf{e}_i, \mathbf{e}_j)$, updated when new algorithms enter $\mathcal{A}$. $\mathbf{D}$ supports: (i) k-nearest-neighbor retrieval for targeted sampling, (ii) clustering to define operational embedding-space regions (§III-F), and (iii) similarity-based deduplication and pruning (§III-C).

C. Coverage-Preserving Archive

ATLAS maintains a bounded archive $\mathcal{A}$ of valid algorithms that serves as a semantic repertoire of the explored algorithm space. Each validated candidate is added to $\mathcal{A}$, after which similarity-based management removes redundancy while preserving coverage of distinct regions in embedding space. Here, multimodality is operationalized in the selected representation: retaining competitive algorithms across separate embedding-space regions, without treating each region as a verified mechanistic family.

Algorithm 1: High-Level ATLAS Pseudocode

Input: Problem specification and I/O interface; LLM; synthesis budget
Output: Training-best algorithm a^*, final archive $\mathcal{A}$, and cluster representatives $\mathcal{T}$

Procedure `ProcessCandidate(`c`)`**:**
  Execute c on every training instance under the resource limits
  Verify its output and feasibility, and independently recompute its objectives
  if *c fails execution, interface, resource, or feasibility checks* **then**
    Regenerate c with `Repair` using the aggregated failure evidence
    Fully re-evaluate the repaired algorithm on all training instances
  if *c succeeds on every training instance* **then**
    Mark c valid and retain its training objective and lineage
  else
    Discard c

$\mathcal{A} \leftarrow \emptyset$
`// Two-phase initialization`
Use `Create` to synthesize algorithms; call `ProcessCandidate(`c`)`
Iteratively apply `Diverge` to references sampled from the growing valid archive; call `ProcessCandidate(`c`)`
Embed the retained algorithms and remove embedding-near duplicates
while *the synthesis budget is not exhausted* **do**
  `// Reorganize the evolving archive`
  $a^* \leftarrow$ the training-best member of $\mathcal{A}$
  $\mathcal{G}^* \leftarrow a^*$ and its embedding-nearest archive members
  Cluster the full archive in embedding space
  Exclude the cluster containing a^* from the non-elite region set
  Remove other $\mathcal{G}^*$ members, discard empty regions, and select each region's training-best surviving representative
  Freeze $\mathcal{G}^*$, the non-elite regions, and their representatives for this iteration
  `// Layer 1: intensify the current best region`
  Schedule repeated `Improve`, `Tune`, and `Simplify` proposals from a^*
  Schedule probabilistically selected refinements of the other members of $\mathcal{G}^*$
  Schedule local hybrids by applying `Combine` to a^* and its neighbors
  `// Layer 2: mature each represented non-elite region`
  foreach *non-elite region and its representative* **do**
    Schedule probabilistically selected `Improve`, `Tune`, and `Simplify` proposals from the representative
    Schedule a within-region `Combine` using the representative and other members of that region
  `// Layer 3: search across regions`
  Schedule `Combine` between representatives from different regions to form hybrids
  Schedule quality-anchored hybrids by applying `Combine` to a^* and non-elite representatives
  Schedule `Diverge` from a^* and non-elite representatives to seek approaches separated from the referenced regions
  $\mathcal{V} \leftarrow \emptyset$
  foreach *scheduled operator call while budget remains* **do**
    Ask the LLM operator to synthesize c from the complete sources of its references
    Call `ProcessCandidate(`c`)` and add it to $\mathcal{V}$ only if valid
  Embed all algorithms in $\mathcal{V}$ and set $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{V}$
  Remove embedding-near redundancy
  If capacity is exceeded, prune the most similar pairs first while protecting the updated best neighborhood
After budget exhaustion, set a^* to the training-best member of the finalized archive
Cluster the finalized archive and set $\mathcal{T}$ to each cluster's training-best member
return $a^*, \mathcal{A}, \mathcal{T}$

1) Similarity-Based Deduplication: Using the distance matrix $\mathbf{D}$ from §III-B, ATLAS removes near-duplicate algorithms at each iteration via two tiers (Algorithm B.3): (i) *strict deduplication* removes one algorithm from each pair where $D_{ij} < 1 - \tau_{\text{strict}}$, and (ii) *performance-aware deduplication* removes one algorithm from each pair where $D_{ij} < 1 - \tau_{\text{soft}}$ and their performance difference is below a small absolute tie tolerance ϵ. In both tiers, ties are broken uniformly at random.

2) Capacity-Constrained Pruning: If $|\mathcal{A}| > N_{\max}$, ATLAS iteratively removes the lower-performing algorithm from the most similar pair (a_i, a_j), where $(i, j) = \arg\min_{i \neq j} D_{ij}$ (Algorithm B.4). To prevent loss of refinement anchors, the elite region $\mathcal{G}^* = \{a^*\} \cup \text{k-NN}(a^*, k)$ is excluded.

3) Effect: Unlike selection-driven evolutionary methods that discard alternatives via global fitness ranking, ATLAS removes algorithms that contribute the least additional coverage (i.e., those redundant under $\mathbf{D}$). This similarity-first management retains representation across multiple embedding-space regions even when their candidates are not currently optimal, while avoiding unbounded archive growth.

D. Semantic Search Operators

ATLAS uses LLM-driven semantic operators, implemented as prompts, to create and transform complete algorithms. We organize the suite by the number of reference algorithms supplied to the LLM and by whether an operator is triggered by evaluation failure.[2] CREATE receives no input algorithm (zero-reference) and generates initial candidates directly from the

[2]In standard evolutionary computation (EC) terminology, each reference algorithm is a *parent*. Zero-, single-, and multi-reference operators therefore correspond to zero-, single-, and multi-parent operators, respectively. We retain *reference* to emphasize that the complete algorithm source is supplied to the LLM.

problem specification and I/O requirements. IMPROVE, TUNE, and SIMPLIFY refine one archived reference, whereas COMBINE and DIVERGE synthesize a new candidate from multiple archived references. REPAIR instead receives a rejected candidate together with classified evaluation evidence and regenerates it for full re-evaluation.

Creation, variation, and crossover operators are common in LLM-based algorithm design [25]; ATLAS adapts these roles to full-algorithm synthesis. Each operator prompt combines its search objective with the problem specification (the objective, constraints, and solution semantics) and the external entry-point and I/O requirements needed to produce an executable algorithm. Permitted dependencies and the structured response format are implementation requirements rather than a prescribed algorithmic scaffold. Every operator requests one structured response containing the complete Python source code, an algorithm name, and a self-contained description. When an operator requires archived references, ATLAS selects them according to the current initialization or search-layer policy, such as uniform archive sampling, current-best selection, embedding-neighborhood selection, or cluster representatives. The exact executable system message, operator instructions, problem-specific prompt components, repair guidance, and response requirements are provided in the public source-code repository.[3]

1) Zero-Reference Generation: **CREATE** generates complete algorithms from the problem specification and I/O requirements without a reference algorithm, and is used only for initialization. Its prompt also provides numerical example instances paired with valid solutions. These solution-grounding examples clarify the required output structure and constraint satisfaction without exposing implementation logic or favoring any algorithmic paradigm.

2) Single-Reference Refinement: Given one reference algorithm, these operators refine it while preserving its paradigm: (i) **IMPROVE**: improves logic and efficiency without changing the fundamental approach; (ii) **TUNE**: adjusts numerical parameters while keeping structure fixed; (iii) **SIMPLIFY**: reduces redundant code and complexity, acting as parsimony pressure against code bloat.

3) Multi-Reference Synthesis: Given multiple references, these operators synthesize new algorithms: (i) **COMBINE**: integrates complementary ideas from the references; nearby references provide a local context, while embedding-distant references provide cross-region context for potential hybridization; (ii) **DIVERGE**: generates algorithms explicitly different from the references, enabling large jumps across regions.

4) Repair: In component synthesis, the surrounding code generally retains responsibility for solution construction and constraint handling, so the generated component need not produce and validate a complete feasible solution. Full-algorithm synthesis transfers this responsibility to the candidate. Its output may therefore violate capacity, time-window, permutation, uniqueness, or completeness requirements even when the code executes without an exception. Full-algorithm candidates also expose entry-point, output-format, runtime, and memory failures.

ATLAS records an infeasible or malformed returned solution as an evaluation failure and rejects the candidate rather than ranking it using a finite penalty. The external evaluator checks feasibility and independently recomputes the objective, but it never completes, modifies, or makes a returned solution feasible. A failure on any evaluated instance invalidates the candidate. For a candidate that reaches evaluation but fails, REPAIR routes the aggregate failure evidence among six prompt modes: memory limit, partial failure, timeout, constraint violation, runtime error, and framework or interface violation. The prompt asks the LLM to regenerate the complete algorithm source with targeted corrections, after which the candidate is fully re-evaluated. Appendix B-F6 details the deterministic failure routing, identified failure classes, correction objectives, and full re-evaluation used by REPAIR.

E. Initialization

ATLAS uses a two-phase bootstrap to establish diverse initial coverage (Algorithm B.2). (i) **Bootstrap generation:** We sample candidates with CREATE (§III-D) at temperature $T{=}1.0$; despite stochasticity, the outputs often cluster around well-known textbook approaches. (ii) **Reference-conditioned diversification:** To expand coverage, we iteratively apply DIVERGE (§III-D), sampling references uniformly from the current archive and requesting approaches that differ from the referenced strategies. This dual-phase initialization substantially improves initial coverage compared to sampling solely with CREATE (Appendix C-B).

F. Three-Layer Search Strategy

ATLAS allocates operator applications across three complementary layers (Algorithm B.5): concentrated refinement around the current best, distributed refinement across other represented regions, and cross-region synthesis using components and interactions from multiple inputs. This is implemented by clustering the archive in embedding space and operating either locally (within a region) or globally (across regions).

1) Archive Clustering: All clustering is based on the distance matrix $\mathbf{D}$ (§III-B). At each iteration, the elite region $\mathcal{G}^*$ is formed from the current best algorithm a^* and its k-nearest neighbors, and Layer 1 operates on this region. We then apply k-medoids with automatic K selection to the full archive (details in Appendix B-E2), yielding operational semantic regions for search.[4] The cluster containing a^* is removed in full, and any other members of $\mathcal{G}^*$ are removed from the remaining clusters.

[3] https://github.com/Danial-Yazdani/ATLAS

[4] Clusters are regions induced by the selected embedding; mechanistic family identity is not inferred from clustering alone.

Empty clusters are discarded, and a representative removed by this process is replaced by the best surviving member according to the training objective. Let Ω_t denote the resulting non-elite clusters and $R_t = |\Omega_t|$. Layer 2 operates only on Ω_t. Layer 3 draws its non-elite representatives from Ω_t, while its quality-anchored COMBINE and DIVERGE streams additionally include a^* as a reference.

2) Layer Roles: **Layer 1 (elite-region refinement)** focuses on a^* and its neighborhood through repeated single-reference refinement and local COMBINE. **Layer 2 (within-region maturation)** applies single-reference refinement to each non-elite representative and, when enough members remain, local COMBINE within its surviving cluster. This gives a representative from a currently inferior region a dedicated proposal opportunity without requiring global-fitness parent selection. The search-layer ablation evaluates this Layer 2 contribution conditional on Layer 3 (Appendix C-D). **Layer 3 (cross-region discovery)** applies multi-reference operators to representatives through three synthesis streams with different reference contexts: (i) COMBINE between representatives without an a^* anchor, (ii) COMBINE between a^* and representatives for quality-anchored cross-region synthesis, and (iii) DIVERGE using representatives together with a^* as references (instructions intended to move proposals away from the referenced archive regions; see Appendix B-D4b). The complete three-layer configuration performs best among the four layer variants; the scope of the supported comparisons is detailed in Appendix C-D.

3) Archive-Conditional Allocation: The operator policy and probabilities are fixed in advance. Layer 1 uses the configured elite-region and intensification settings, whereas the nominal work assigned to Layers 2 and 3 scales with the surviving cluster count R_t; in particular, each Layer 3 stream schedules R_t candidates, subject to the remaining global budget. Because clustering, elite membership, and representatives are recomputed at every iteration, the scale and reference context of the schedule change with the archive, and Layer 1 recenters whenever the incumbent best changes. The detailed operator schedule and settings are given in Algorithm B.5 and Appendix B-E3.

This structure is related in motivation to island-model search and classic MAP-Elites, which continue sampling alternatives rather than selecting references solely by global fitness [36, 31]. The mechanism is different. Unlike an island model, ATLAS has no persistent demes connected by migration; unlike classic MAP-Elites, it has no fixed task-specific behavior-descriptor grid with cell-wise replacement. Instead, it maintains one global archive and recomputes transient embedding-induced clusters for search allocation.

IV. EXPERIMENTS

This section first describes the benchmark problems, comparison methods, evaluation protocol, and implementation settings used to assess ATLAS. It then reports the main comparisons, generalization beyond the default benchmark settings, and the training-selected archive representatives. Finally, component ablations and design analyses examine the principal choices underlying the search.

A. Experimental Setup

1) Benchmark Problems: We evaluate on four NP-hard combinatorial optimization problems: Capacitated Vehicle Routing Problem **(CVRP)** [8], which minimizes travel distance under vehicle capacity constraints; CVRP with Time Windows **(CVRPTW)** [42], which adds customer time-window constraints to CVRP; Flow Shop Scheduling **(FSS)** [15], which minimizes makespan in a fixed machine order; and Quadratic Assignment Problem **(QAP)** [22], which minimizes flow-distance assignment cost between facilities and locations. Together, they cover routing, scheduling, and assignment, spanning different constraint structures and objective types. Problem formulations and instance-generation details are provided in Appendices B-A and B-B. The exact problem descriptions and I/O requirements presented to the LLM are provided in the public repository.

2) Baselines: We compare against three groups of baselines (implementation details in Appendix B-G):

a) Human-designed, domain-specific methods: For CVRP and CVRPTW, we use PyVRP [52, 53], Google OR-Tools [14], and VROOM [6, 50]. For FSS, we use NEH [32], Taillard-accelerated Iterated Greedy with idle-time tie-breaking (IG-TB) [38, 45, 12], and Iterative Beam Search [24]. For QAP, we use Robust Tabu Search (RoTS) [46], Simulated Annealing [5], Breakout Local Search (BLS) [1], and Memetic Search (BMA) [2]. Runtime-sensitive methods are evaluated under the corresponding benchmark-setting-specific per-instance runtime caps; implementation provenance and configurations are provided in Appendix B-G1.

b) Component-based LLM methods: We include REEVO [57], EOH [25], and MCTS-AHD [61]. Because their official repositories target different problems and use non-uniform scaffolds, prompts, and evaluators, we evaluate them under a shared greedy-construction scaffold and common benchmark interface based on LLM4AD [26], while aligning method-specific search settings to the official repositories. All LLM-based methods use the same LLM backend and matched token-budget-based termination criteria; see Appendix B-G2.

*c) Full-synthesis baseline (*EOH-FULL*):* We construct EOH-FULL by adapting EOH's evolutionary framework to evolve full algorithms using ATLAS's problem- and interface-aware operators, repair, and evaluation framework while retaining EOH's fitness-driven selection. Comparing component-based EOH with EOH-FULL evaluates the combined practical effect of moving from a fixed component scaffold to scaffold-free full-algorithm synthesis with the machinery needed to support it. Comparing EOH-FULL with ATLAS then isolates the contribution of ATLAS's archive-based embedding-guided quality-diversity search from standard fitness-driven evolution under matched full-synthesis conditions; see Appendix B-G3.

3) Evaluation Protocol: We evaluate methods using the following protocol:

a) Benchmark settings and data splits: We consider four *default* benchmark settings: FSS with 50 jobs and 10 machines, CVRP with 50 customers, CVRPTW with 50 customers, and QAP with 50 facilities. For each setting, the reported protocol uses 31 training instances (split seed 2024) and 31 independently generated test instances (split seed 42). Search and archive construction access only the training split. The algorithm selected using training performance is then evaluated once on the test set. Split-local random-number generators and immutable instance manifests keep both sets disjoint and reproducible (Appendix B-B).

b) Independent runs: LLM-based methods (ATLAS, EoH, ReEvo, MCTS-AHD, and EoH-Full) perform $R = 5$ independent synthesis runs with different random seeds, each yielding one training-selected optimization algorithm; the algorithms may differ across runs. Five runs per method is set by the substantial LLM API cost of generation and the computational cost of repeatedly evaluating candidate algorithms throughout each search. Human-designed, domain-specific baselines are evaluated directly on the test instances under the same benchmark-setting-specific runtime caps.

c) LLM configuration and budgets: All LLM-based synthesis methods use OpenAI GPT-5-mini with reasoning effort set to *low* and temperature $T = 1.0$. For end-to-end ATLAS component ablations and search-configuration sensitivity analyses, we fix the search budget to $B = 500$ evaluated operator executions. The embedding and initialization analyses instead use the study-specific sample sizes reported in Appendix C. For cross-method comparisons, we match total token consumption per synthesis run, since per-iteration token usage differs substantially between component synthesis and full-algorithm synthesis. The resulting benchmark-specific token budgets, calibrated from the average token usage of 500 evaluated operator executions under full synthesis, are 5M for FSS, 6.5M for CVRP, 7M for CVRPTW, and 6M for QAP.

d) Full synthesis evaluation: Each candidate algorithm is executed on each instance to produce a solution. That solution is then passed to a problem-specific external evaluator that checks output format, detects constraint violations, and computes the objective value (Appendix B-F).

e) Runtime caps and execution model: Each benchmark setting uses a common per-instance runtime cap for all compared methods: 210 s for FSS, 30 s for CVRP, 120 s for CVRPTW, and 240 s for QAP. All methods are executed under single-core, single-threaded settings whenever applicable, making these runtime caps more comparable across synthesized and human-designed methods. For the routing comparisons, performance-critical components of PyVRP and the OR-Tools routing engine execute as compiled C++ code through Python interfaces, whereas the algorithms synthesized by the LLM-based methods, including ATLAS, execute as Python programs. Equal wall-clock limits therefore control the available time but do not remove implementation-language overhead: for comparable low-level operations, the compiled baselines can generally perform more search iterations or neighborhood evaluations within the cap. Thus, this aspect of the runtime protocol favors the human-designed CVRP and CVRPTW baselines over the LLM-based methods. The full cap is applied unchanged during training and test execution.

f) Reporting: At the end of each ATLAS run, the algorithm with the best training performance is selected and evaluated on the 31 test instances. Test outcomes cannot trigger fallback, re-ranking, repair, archive mutation, or a repeat test evaluation. Any timeout, memory violation, exception, invalid output, non-finite objective, or infeasible solution on any test instance causes the entire run to be classified as failed. The outcome ledger retains the status and failure reason for every failed instance. This failure rule was not triggered in any reported LLM-based synthesis run.

g) Reader-facing objective display: For every problem and benchmark setting, paired tests on the raw per-instance objectives identify the statistically best-performing group of human-designed, domain-specific methods. Within that group, we designate the method with the lowest reported mean objective as the reference. If $\overline{J}_{m,s}$ is method m's reported mean objective in setting s and $\overline{J}_{\mathrm{ref},s}$ is that reference mean, we report the relative mean gap $100(\overline{J}_{m,s} - \overline{J}_{\mathrm{ref},s})/\overline{J}_{\mathrm{ref},s}$. A value of zero matches the reference mean, a positive value is worse, and a negative value is better. This descriptive quantity is a gap to the best evaluated domain-specific reference, not an optimality gap. The same transformation is used for the archive-representative tables, where test gaps are computed from the reported representative mean objectives only after the representatives have been fixed using training information. The corresponding reference means are reported in Table B.1.

h) Statistical testing: For each test instance, we average the raw objective values from the five independent runs of each LLM-based method; human-designed methods contribute their direct per-instance raw objectives. This averaging marginalizes over synthesis runs and yields one raw objective value per method and test instance. Pairwise comparisons between all methods use a two-sided Wilcoxon signed-rank test at significance level 0.05 on the resulting 31 pairs of per-instance raw objective values. Percentage gaps are descriptive only and are not inputs to the test. In the result tables, boldface identifies the statistically best-performing group across all methods: the method with the lowest mean and any method not significantly different from it. Every textual claim that one method achieves better test-set objective performance than another is based on this paired test (Appendix B-C5).

4) ATLAS Configuration: Table I summarizes the key hyperparameters of ATLAS.

5) ATLAS Implementation: Implementation details, including pseudocode, hyperparameters, and the evaluation pipeline, are provided in Appendix B. The complete executable prompt definitions are provided in the public repository.

a) Hardware and environment: All experiments were run on WSL2 (Ubuntu 22.04 LTS) using Python 3.12, hosted on a workstation with an AMD Ryzen 9 7950X (32 threads), 64 GB physical RAM (30 GB allocated to WSL2, 8 GB swap), and

TABLE I: *ATLAS configuration.* Complete details and rationale in Appendix B-E3.

Parameter	Value	Parameter	Value
Archive capacity	$N_{\max} = 100$	Layer 1 size	$\lvert\mathcal{G}^*\rvert = 5$
Initialization	50 algorithms	Clustering	K-Medoids, adaptive K
Multi-ref operators (refs)		*Single-ref operator probs*	
COMBINE	$m_{\text{comb}} = 2$	IMPROVE, TUNE	$p = 0.5$
DIVERGE	$m_{\text{div}} = 3$	SIMPLIFY	$p = 0.2$
Embedding model	MGTE-LARGE-EN-V1.5 [58]	LLM	GPT-5-MINI (T=1.0, LOW)
Budget	500 ops or benchmark-setting-specific tokens	Timeout	benchmark-setting-specific
Train/test	31/31	Train/test split seeds	2024/42

an NVIDIA GeForce RTX 5090 (32 GB VRAM). Dependency constraints and installation instructions are provided in the `requirements.txt` file in the ATLAS repository. All compared methods were executed under single-core, single-threaded settings whenever applicable: each candidate–instance invocation executes within one worker process and one thread, while different instances are evaluated in parallel; restricting human-designed, domain-specific baselines to the same per-instance execution model makes the runtime caps more comparable across methods.

b) Per-run resource usage and cost: Under the default ATLAS settings in Table I (500 operator calls and approximately 5M–7M total tokens per synthesis run, with roughly two-thirds input and one-third output, including reasoning tokens), the estimated OpenAI API cost is approximately $4.2–$5.8 per run using GPT-5-MINI (LOW) at current standard pricing. The wall-clock time of a synthesis run depends on the benchmark setting, API latency, and the execution time of the candidate algorithms being evaluated; under the default budget of 500 evaluated operator executions, a typical synthesis run required approximately 10–15 hours in our setup. In practical settings, this automated synthesis cost is modest relative to the engineering effort required to manually design and tune a competitive problem-specific optimization algorithm.

B. Main Experimental Results

1) Comparison with Baselines: The paired Wilcoxon analysis shows that ATLAS statistically outperforms every LLM-based baseline across all four default benchmark settings (Table II). The component-synthesis implementations of EOH, REEVO, and MCTS-AHD share the same greedy-construction scaffold and therefore search only within that controlled artifact interface. Among the component-synthesis methods, EOH achieves the best results in three of the four default settings, while MCTS-AHD performs best on FSS.

The full-synthesis baseline EOH-FULL substantially improves over the component-based implementations. This comparison demonstrates the benefit of the combined change from their shared greedy component interface to ATLAS's full-algorithm artifact and corresponding problem- and interface-aware operators; it should not be interpreted as isolating scaffold freedom alone. Because EOH-FULL and ATLAS share the full-synthesis operators, repair, evaluator, LLM, and budgets, ATLAS's remaining advantage over EOH-FULL more directly supports the contribution of its archive-based embedding-guided quality-diversity search. Fitness-driven selection used in EOH-FULL can concentrate effort around early promising regions, whereas ATLAS preserves coverage across embedding-separated regions, refines them in parallel, and performs cross-cluster synthesis.

ATLAS is also competitive with the human-designed, domain-specific baselines. Its relative mean gaps to the best such reference are 0.347% on FSS, 0.094% on CVRP, 0.000% on CVRPTW, and 0.460% on QAP. On CVRP, the paired Wilcoxon test finds no statistically significant difference between ATLAS and PyVRP, the selected human-designed reference. ATLAS also matches the lowest reported CVRPTW mean at the displayed precision and remains within 0.46% of the reference mean on all four default settings, despite synthesizing an optimization algorithm automatically for each problem. The reported gaps and displayed-precision comparisons are descriptive; statistical comparisons use paired raw objectives. Examples of best-found algorithms generated by ATLAS, one for each problem, are available with their descriptions in the source-code repository.

2) Generalization Beyond Default Benchmark Settings: We further evaluate ATLAS on FSS and CVRP at half and double the default problem scale, with runtime caps scaled accordingly. For each scaled setting, synthesis is re-run from scratch on newly generated train/test instances at that scale.

ATLAS achieves the best results among the LLM-based methods in all four additional settings. On FSS, ATLAS has relative mean gaps of 0.029% and 1.651% at the smaller and larger scales, respectively. At $n25m5$, the paired Wilcoxon test finds no statistically significant difference between ATLAS and IG-TB. On CVRP, it matches the best reported domain-specific mean at $n25$ and has a 1.820% gap at $n100$.

Interestingly, on the smaller FSS setting, EOH outperforms EOH-FULL, suggesting that the shared greedy-construction scaffold used by EOH remains particularly effective in this easier setting.

Our cross-scale investigation suggests that the performance of the synthesized algorithms becomes more sensitive to numerical hyperparameter settings as problem scale increases. This increased sensitivity may contribute to the wider gaps observed at larger scales. The current TUNE operator proposes numerical changes based on the candidate source code. Each proposal is then assessed through the standard ATLAS evaluation and selection process, but the operator does not perform dedicated

TABLE II: *Comparison with baselines on the default benchmark settings using relative mean gaps.* Values are relative mean gaps (%) to the statistically selected human-designed reference in each setting; lower is better. The reference in each column is the human-designed, domain-specific method with the lowest mean within the statistically best-performing human-designed group and therefore has gap 0.000%. Statistical tests use the paired raw objectives, not these gaps. Boldface identifies the statistically best-performing group across all methods under the paired Wilcoxon signed-rank test: the method with the lowest mean and any method not significantly different from it. Light-gray cell shading identifies the best LLM-based method in each setting.

Group	**Method**	**FSS** ($n50m10$)	**CVRP** ($n50$)	**CVRPTW** ($n50$)	**QAP** ($n50$)
Domain-specific	NEH	2.809%	–	–	–
	IG-TB (Taillard acc.)	**0.000%**	–	–	–
	Iterative Beam Search	1.787%	–	–	–
	OR-Tools (GLS)	–	1.430%	0.121%	–
	PyVRP (ILS)	–	**0.000%**	**0.000%**	–
	VROOM	–	**0.095%**	**0.060%**	–
	Robust Tabu Search	–	–	–	0.455%
	BLS	–	–	–	**0.015%**
	BMA	–	–	–	**0.000%**
	Simulated Annealing	–	–	–	1.405%
LLM-based	EoH	3.909%	21.106%	20.711%	1.520%
	ReEvo	3.912%	29.071%	35.161%	1.986%
	MCTS-AHD	2.837%	24.393%	21.505%	2.012%
	EoH-Full	0.782%	3.815%	4.520%	1.324%
	ATLAS (ours)	0.347%	**0.094%**	**0.000%**	0.460%

TABLE III: *Generalization beyond the default benchmark settings using relative mean gaps (re-run from scratch).* Values are relative mean gaps (%) to the statistically selected human-designed reference for every problem size. Lower is better. Boldface identifies the statistically best-performing group across all methods under the paired Wilcoxon signed-rank test: the method with the lowest mean and any method not significantly different from it. Light-gray cell shading identifies the best LLM-based method in each setting.

Group	**Method**	**FSS** ($n25m5$)	**FSS** ($n100m20$)	**CVRP** ($n25$)	**CVRP** ($n100$)
Domain-specific	NEH	1.143%	4.133%	–	–
	IG-TB (Taillard acc.)	**0.000%**	**0.000%**	–	–
	Iterative Beam Search	2.321%	2.865%	–	–
	OR-Tools (GLS)	–	–	**0.162%**	4.930%
	PyVRP (ILS)	–	–	**0.000%**	**0.000%**
	VROOM	–	–	**0.000%**	0.975%
LLM-based	EoH	0.310%	6.810%	23.662%	16.371%
	EoH-Full	0.515%	2.529%	1.919%	5.801%
	ATLAS (ours)	**0.029%**	1.651%	**0.000%**	1.820%

feedback-driven hyperparameter optimization. Accurately calibrating sensitive hyperparameters normally requires repeated empirical evaluations. Consequently, proposals based only on source code are unlikely to identify robust settings reliably.

3) Training-Selected Archive Representatives: To illustrate ATLAS's multimodal retention across embedding-space regions and the structures it discovers, Table IV reports the complete selected representative set retained in the final archive of a single synthesis run, the run among the $R{=}5$ synthesis runs whose deployed algorithm achieved the best training objective, for each problem. Each representative is selected using training information as the training-best member of a final embedding-space cluster, so representative identities are fixed before test evaluation. The descriptive test results show that multiple representatives for every problem lie within 1% of the selected human-designed reference mean for that problem. The fixed representatives are subsequently ordered by their test relative mean gaps solely to make their quality and structural differences interpretable. The listed names and descriptions were manually checked to ensure that they are broadly consistent with the corresponding synthesized algorithms.

TABLE IV: *Training-selected archive representatives with descriptive test performance.* All representatives are drawn from the final archive of a single synthesis run: the run, among the $R{=}5$ synthesis runs, whose deployed algorithm achieved the best training objective. Representative identities are fixed from training objectives as the training-best members of final embedding-space clusters. Values in parentheses are test-set relative mean gaps (%) to the statistically selected human-designed, domain-specific reference; lower is better. Ordering by these gaps is descriptive only and affects neither selection nor statistical tests.

Problem	Training-selected representatives, ordered by descriptive test gap
FSS ($n50m10$)	Priority-Seeded Prefix-Guided LNS with Lightweight Anneal (0.331%); Priority-seeded NEH with Guided Neighborhood Search (0.458%); Priority-Spectral Hybrid with Tabu-Repair Local Search (0.472%); Prefix-Aware Hybrid Genetic with Annealed Polish (0.478%); Sampled Pairwise Greedy with Bounded Cache and Adaptive Iterated Refinement (0.527%); Hybrid NEH-Regret Genetic Search with Aggressive Complexity Reduction (0.714%); Guided Greedy Adaptive Cache with Pheromone and Block Relocation (0.965%); Bottleneck-Guided Greedy with ILS and Simulated Annealing (1.239%)
CVRP ($n50$)	Hybrid Multi-Start Clustered-Repair Large Neighborhood Search (0.000%); Multi-Start Regret Insertion with Focused Large Neighborhood Search (0.095%); Spectral-Seeding Regret-LNS with Focused Savings (0.191%); Hybrid Demand-Seeding with Localized Destroy-and-Repair (3.622%); Clustered Demand-Aware Insertion with Focused LNS (3.622%); Adaptive Clustered Chain-Savings Solver (3.908%)
CVRPTW ($n50$)	Large-Neighborhood Search with Time-Aware Regret Repair (0.000%); Ruin-and-Recreate Vehicle Routing with Time Windows (0.000%); Hybrid Label-Constructive and Ruin-Repair Routing (0.000%); Hybrid Adaptive Ruin-and-Repair with Temporal Beam Seeding (0.060%); Hybrid Guided Repair with Selective Savings Merge (0.302%); Permutation-Guided Large Neighborhood Search (2.171%); Radial and Time-Priority Insertion with Neighbor Guidance (3.317%)
QAP ($n50$)	Hierarchical Spectral-Memetic Solver with Block Neighborhoods and Cycle Diversification (0.440%); Soft-Probabilistic Beam with Block Reassignment and Tabu Polishing (0.461%); Adaptive Spectral-Pheromone Memetic Solver (0.496%); Spectral-Guided Adaptive Large-Neighborhood Search with Local Intensification (0.525%); Spectral-Softassign Hybrid with Mixed Local Search (0.813%)

a) Mechanism diversity and hybridization: Code inspection of the displayed algorithms in Table IV identifies differences in core construction or search components together with some shared refinement mechanisms. For example, in CVRP and CVRPTW, several representatives differ in their primary construction or global-search logic, including regret insertion, savings-based construction, clustering, ruin-and-recreate, and label- or beam-inspired seeding, while often converging on strong repair or large-neighborhood refinement components. In FSS, the representatives span different core search patterns, including NEH-style construction, greedy or pairwise search, genetic search, tabu-guided local search, and large-neighborhood refinement. In QAP, the representatives likewise cover different core search engines, including memetic search, pheromone-guided search, beam-style search, tabu-guided local improvement, and large-neighborhood refinement.

This pattern of structural diversity, component reuse, and hybridization is consistent with ATLAS's design: the archive preserves representatives from multiple embedding-space regions, while Layer 3 can draw components and their interactions from different input algorithms and reorganize them into new hybrids. The displayed repertoire therefore demonstrates that scaffold-free synthesis can retain several competitive, structurally different, multi-component algorithms rather than only one best design. It does not by itself establish archive-wide functional validity of the embedding clusters or per-instance portfolio complementarity.

C. Component Ablations and Design Analyses

We report the search-layer ablation and summarize the principal design analyses here; complete setups, results, and discussion for both the component ablations and design analyses are provided in Appendix C.

As a controlled comparative experiment rather than a component ablation, Table II shows that EOH-FULL's full-algorithm artifact and problem- and interface-aware operators improve over the shared-scaffold component implementations. ATLAS then improves over EOH-FULL under shared full-algorithm operators, repair, evaluator, LLM, and budgets, providing evidence for the framework-level value of its archive-based semantic quality-diversity search organization beyond the shared full-synthesis machinery, without isolating every internal search mechanism.

The layer ablation in Table V shows that the complete three-layer configuration performs best among the four layer configurations. Comparing Variant A with Variant C isolates the addition of Layer 1 to Layers 2+3 and shows the value of intensive refinement in the elite region. Comparing Variant C with Variant D isolates the addition of Layer 2 when Layer 3 remains active and supports representative-level refinement. Variant D (Layer 3 only) also outperforms Variant B (Layer 1 only), showing that the Layer 3-only configuration is stronger than the Layer 1-only configuration in these settings. Full details are provided in Appendix C-D.

The appendix reports one further search-guidance ablation and four supporting analyses: (i) **Embedding representation analysis:** MGTE-LARGE-EN-V1.5 with concatenated name, description, and preprocessed code is the strongest single-encoder embedding strategy (Appendix C-A); (ii) **Initialization-strategy analysis:** dual-phase initialization with DIVERGE yields substantially greater initial embedding-space dispersion than prompt variation alone while maintaining high validity (Appendix C-B); (iii) **Semantic search-guidance ablation:** semantic neighbor selection and grouping improve search guidance over fitness-based and random alternatives (Appendix C-C); (iv) **Reasoning-effort sensitivity and cost analysis:** for GPT-5-MINI, Low provides the best quality–cost trade-off, with Medium and High substantially increasing token usage for only marginal

TABLE V: *Relative mean gaps for the layer ablation.* Percent gap to the statistically selected human-designed reference in each setting (PyVRP for CVRP and IG-TB for FSS); lower is better. Each gap is computed from the configuration's mean raw objective and the corresponding reference mean. Boldface identifies the statistically best-performing group among the ablation configurations from paired tests on raw per-instance objectives; the displayed gaps are descriptive and are not test inputs. Each run is represented by its training-selected algorithm; the test set is used once for evaluation without fallback, and failed runs are retained. Detailed setup and analysis are provided in Appendix C-D.

Configuration	**CVRP** ($n50$) gap (%)↓	**FSS** ($n50m10$) gap (%)↓
Variant A: All layers	**0.094**	**0.347**
Variant B: Layer 1 only	3.620	0.814
Variant C: Layers 2+3	0.668	0.474
Variant D: Layer 3 only	2.095	0.672

gains (Appendix C-E); (v) **Underlying-LLM comparison:** ATLAS remains effective across several LLM backends, with stronger models performing better but the differences remaining moderate (Appendix C-F).

V. Conclusion

We have presented ATLAS, a framework for scaffold-free full-algorithm synthesis in combinatorial optimization. The problem specification supplies the objective and constraints, while a minimal external I/O interface defines how instances are provided and solutions are returned without prescribing internal algorithmic structure. Each generated algorithm is responsible for satisfying the problem requirements, and full-algorithm operators leave its internal component inventory and control flow to synthesis. By organizing complete algorithms in an embedding-guided coverage archive and allocating search across three complementary layers, ATLAS supports multimodal search, refinement, and recombination across multiple embedding-space regions. Across four combinatorial optimization benchmarks, it outperforms several state-of-the-art component-synthesis methods and the matched full-synthesis baseline, while remaining highly competitive with strong human-designed, domain-specific methods. These results support the combination of full-algorithm synthesis under a fixed I/O interface with embedding-guided quality-diversity search for multimodal retention and hybrid synthesis.

Future work will build directly on our empirical analysis. Synthesized full algorithms, particularly hybrid designs, can contain several algorithm-specific hyperparameters. Our inspection suggests that more effective numerical configuration could narrow part of the remaining performance gap between promising ATLAS-generated algorithms and the strongest human-designed methods. The current TUNE operator proposes changes from source code without conducting dedicated empirical hyperparameter optimization. LLaMEA-HPO addresses this challenge by coupling LLM-driven structural generation with in-loop hyperparameter optimization (HPO), reporting competitive or improved performance with fewer LLM queries [49]. Integrating training-only HPO into ATLAS is therefore a promising direction for improving the generated algorithms. A complementary direction concerns how ATLAS distributes search effort across its archive. ATLAS preserves and refines representatives across embedding-space regions to maintain diverse search opportunities. A more adaptive resource-allocation mechanism could use accumulated training evidence to concentrate refinement on promising regions while retaining sufficient coverage for continued exploration and cross-region synthesis. This could improve search efficiency without sacrificing the diversity supported by the archive.

References

[1] U. Benlic and J.-K. Hao. Breakout local search for the quadratic assignment problem. *Applied Mathematics and Computation*, 219(9): 4800–4815, 2013. doi: 10.1016/j.amc.2012.10.106.

[2] U. Benlic and J.-K. Hao. Memetic search for the quadratic assignment problem. *Expert Systems with Applications*, 42(1):584–595, 2015. doi: 10.1016/j.eswa.2014.08.011.

[3] K. Chatzilygeroudis, A. Cully, V. Vassiliades, and J.-B. Mouret. Quality-diversity optimization: a novel branch of stochastic optimization. In *Black box optimization, machine learning, and no-free lunch theorems*, pages 109–135. Springer, 2021.

[4] B. Chen, S. Zhu, B. Liu, Y. Zhao, T. Pu, H. Li, and Z. Zhu. A2DEPT: Large language model-driven automated algorithm design via evolutionary program trees. *arXiv preprint arXiv:2604.24043*, 2026. doi: 10.48550/arXiv.2604.24043.

[5] D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93–100, 1990.

[6] J. Coupey, J.-M. Nicod, and C. Varnier. *Vroom v1.14, Vehicle Routing Open-source Optimization Machine*. Verso, Besançon, France, 2024. http://vroom-project.org/.

[7] A. Cully. Autonomous skill discovery with quality-diversity and unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 81–89, 2019.

[8] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[9] P. V. T. Dat, L. Doan, and H. T. T. Binh. HSEvo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26931–26938, 2025.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[12] V. Fernandez-Viagas and J. M. Framinan. On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. *Computers & Operations Research*, 45:60–67, 2014. doi: 10.1016/j.cor.2013.12.012.

[13] S. Finck, N. Hansen, R. Ros, and A. Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions. Technical Report RR-6829, INRIA, 2009. URL https://inria.hal.science/inria-00362633v2/document. Updated version as of February 2019.

[14] V. Furnon and L. Perron. OR-Tools Routing Library. https://developers.google.com/optimization/routing/, 2026. Version 9.15.6755.

[15] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1 (2):117–129, 1976.

[16] Google Cloud. Solve harder problems with AlphaEvolve, now available to everyone on Google Cloud, July 2026. URL https://cloud.google.com/blog/products/ai-machine-learning/alphaevolve-is-available-for-everyone. Accessed August 12, 2026.

[17] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021.

[18] Z. Huang, W. Wu, K. Wu, J. Wang, and W.-B. Lee. CALM: Co-evolution of algorithms and language model for automatic heuristic design. *arXiv preprint arXiv:2505.12285*, 2025.

[19] ISA-QAP-Algorithms contributors. ISA-QAP-Algorithms: Implementations for the quadratic assignment problem. https://github.com/jeffrey-chr/ISA-QAP-Algorithms/tree/f0840b25, 2025. Public implementation collection, commit f0840b25.

[20] js-aguiar. Algorithms for the permutation flowshop scheduling problem using python and cython. https://github.com/js-aguiar/permutation-flowshop, 2019. GitHub repository, commit 1a2c357cc09f717153f3a73081fd51f2a54126a1.

[21] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[22] T. C. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica: journal of the Econometric Society*, pages 53–76, 1957.

[23] X. Li, M. G. Epitropakis, K. Deb, and A. Engelbrecht. Seeking multiple solutions: An updated survey on niching methods and their applications. *IEEE Transactions on Evolutionary Computation*, 21(4):518–538, aug 2017. doi: 10.1109/TEVC.2016.2638437.

[24] L. Libralesso, P. A. Focke, A. Secardin, and V. Jost. Iterative beam search algorithms for the permutation flowshop. *European Journal of Operational Research*, 301(1):217–234, 2022. doi: 10.1016/j.ejor.2021.10.015.

[25] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang. Evolution of heuristics: towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, pages 32201–32223, 2024.

[26] F. Liu, R. Zhang, Z. Xie, R. Sun, K. Li, X. Lin, Z. Wang, Z. Lu, and Q. Zhang. LLM4AD: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024.

[27] F. Liu, Y. Liu, Q. Zhang, X. Tong, and M. Yuan. EoH-S: Evolution of heuristic set using LLMs for automated heuristic design. In *40th Annual AAAI Conference on Artificial Intelligence (AAAI-26)*. AAAI Press, 2026.

[28] F. Liu, Y. Yao, P. Guo, Z. Yang, X. Lin, Z. Zhao, X. Tong, K. Mao, Z. Lu, Z. Wang, et al. A systematic survey on large language models for algorithm design. *ACM Computing Surveys*, 2026.

[29] F. Liu, R. Zhang, S. Yao, Q. Hu, K. Zheng, Z. Lu, and Q. Zhang. Hierarchical representations for cross-task automated heuristic design using LLMs. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=yLytuQFgeK.

[30] Z. Ma, H. Guo, Y.-J. Gong, J. Zhang, and K. C. Tan. Toward automated algorithm design: A survey and practical guide to meta-black-box-optimization. *IEEE Transactions on Evolutionary Computation*, 2025.

[31] J.-B. Mouret and J. Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[32] M. Nawaz, E. E. Enscore Jr, and I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1): 91–95, 1983.

[33] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[34] OpenAI. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/, 2025. Accessed: 2025-01-24.

[35] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[36] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[37] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[38] R. Ruiz and T. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007. doi: 10.1016/j.ejor.2005.12.009.

[39] E. Schubert and P. J. Rousseeuw. Fast and eager k-medoids clustering: O(k) runtime improvement of the PAM, CLARA, and CLARANS algorithms. *Information Systems*, 101:101804, 2021.

[40] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv*, abs/2402.03300, 2024.

[41] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2024.

[42] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2): 254–265, 1987.

[43] W. Sun, S. Feng, S. Li, and Y. Yang. CO-Bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 33126–33134, 2026.

[44] T. Suresh, R. G. Reddy, Y. Xu, Z. Nussbaum, A. Mulyar, B. Duderstadt, and H. Ji. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*, 2025.

[45] É. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1): 65–74, 1990. doi: 10.1016/0377-2217(90)90090-X.

[46] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(4–5):443–455, 1991.

[47] P. Toth and D. Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

[48] N. van Stein and T. Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2):331–345, Apr. 2025. doi: 10.1109/TEVC.2024.3497793.

[49] N. van Stein, D. Vermetten, and T. Bäck. In-the-loop hyper-parameter optimization for LLM-based automated design of heuristics. *ACM Trans. Evol. Learn. Optim.*, Apr. 2025. doi: 10.1145/3731567.

[50] VROOM Project. VROOM v1.16.0-dev. https://github.com/VROOM-Project/vroom/tree/07be776fc20b8d6c85d9f78797e38a9f4e44ec44, 2026. Source snapshot used in the experiments, commit 07be776fc20b8d6c85d9f78797e38a9f4e44ec44.

[51] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics.

[52] N. A. Wouda, L. Lan, and W. Kool. PyVRP: A high-performance vrp solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024. doi: 10.1287/ijoc.2023.0055.

[53] N. A. Wouda, L. Lan, and W. Kool. PyVRP v0.13.3. Zenodo software release, Feb. 2026. URL https://doi.org/10.5281/zenodo.18657571.

[54] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024.

[55] X. Yang, L. Zhang, H. Qian, L. Song, and J. Bian. HeurAgenix: Leveraging LLMs for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025.

[56] S. Yao, F. Liu, X. Lin, Z. Lu, Z. Wang, and Q. Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 27144–27152, 2025.

[57] H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

[58] X. Zhang, Y. Zhang, D. Long, W. Xie, Z. Dai, J. Tang, H. Lin, B. Yang, P. Xie, F. Huang, et al. mGTE: Generalized long-context text representation and reranking models for multilingual text retrieval. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1393–1412, 2024.

[59] Y. Zhang, R. Cheng, G. Yi, and K. C. Tan. A systematic survey on large language models for evolutionary optimization: From modeling to solving. *arXiv preprint arXiv:2509.08269*, 2025.

[60] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

[61] Z. Zheng, Z. Xie, Z. Wang, and B. Hooi. Monte Carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Forty-second International Conference on Machine Learning*, 2025.

APPENDIX TABLE OF CONTENTS

APPENDIX A
EXTENDED RELATED WORK AND COMPARATIVE ANALYSIS

This section contextualizes ATLAS within the broader landscape of LLM-based automated algorithm design from two complementary perspectives. First, we position ATLAS within recent taxonomic frameworks proposed in comprehensive surveys (Appendix A-A), highlighting its formulation of scaffold-free full-algorithm synthesis under a fixed I/O interface. Second, we provide a technical overview of recent LLM-based methods (Appendix A-B) and concentrate direct comparisons on designed-multiplicity and full-synthesis approaches, for which the relationship to ATLAS requires closer examination.

A. Positioning ATLAS in Existing Taxonomies

Recent systematic reviews have formalized the landscape of LLM-based optimization, providing useful taxonomic foundations for contextualizing ATLAS. We position ATLAS within three complementary frameworks, each offering a different perspective on automated algorithm design.

1) LLM-as-Designer (Role-Based View): Liu et al. [28] categorize LLMs for Algorithm Design (LLM4AD) into four paradigms based on the LLM's role: LLM-as-Optimizer, LLM-as-Predictor, LLM-as-Extractor, and LLM-as-Designer (LLMaD). ATLAS belongs to LLMaD, where the model generates algorithmic code rather than directly proposing solutions. Critically, Liu et al. [28] observe that most LLMaD methods struggle with "synthesizing complete, state-of-the-art algorithms" and instead target "specific components (e.g., heuristics, reward functions, code snippets) rather than complete algorithms." ATLAS addresses this limitation in combinatorial optimization by synthesizing complete algorithms from a problem specification under a fixed I/O interface.

2) High-Level Algorithm Generation (Intervention-Level View): Complementing this role-based taxonomy, Zhang et al. [59] organize the field by separating *Optimization Modeling* from *Optimization Solving*, with the latter further divided into three paradigms: (i) LLMs as stand-alone optimizers, (ii) low-level LLM-assisted methods (embedded components), and (iii) high-level LLM-assisted methods (algorithm selection and generation). ATLAS operates at the high-level generation tier. While algorithm selection methods "choose the most suitable algorithm from a portfolio for each problem instance" [59], ATLAS performs *open-ended synthesis*, maintaining a search repertoire of generated algorithms across multiple embedding regions rather than selecting from a fixed library.

3) Generative MetaBBO (Mechanism $\times$ Task View): At a finer level of granularity, Ma et al. [30] unify automated design under *Meta-Black-Box Optimization (MetaBBO)*, categorizing methods along two dimensions: learning mechanism (Reinforcement Learning, Supervised Learning, Neuroevolution, or In-Context Learning) and meta-level task (Algorithm Selection, Configuration, Solution Manipulation, or Generation). ATLAS is a MetaBBO method for Algorithm Generation via In-Context Learning. The key distinction is that while many MetaBBO approaches perform *parametric* meta-optimization, such as tuning hyperparameters or selecting operators within fixed templates, ATLAS performs *structural* meta-design by synthesizing the complete algorithmic logic under a fixed external I/O interface.

Synthesis: Scaffold-Free Full-Algorithm Synthesis under a Fixed I/O Interface: Across these three taxonomic lenses, ATLAS lies at the intersection of LLMaD (role), high-level generation (intervention), and structural MetaBBO (mechanism). Appendix A-B reviews selected recent methods through these taxonomic and technical dimensions.

B. Technical Overview of Selected Recent Methods

Appendix A-A positioned ATLAS within the LLM-as-Designer paradigm for high-level algorithm generation. The methods reviewed here differ in synthesis scope (component-based vs. full-algorithm), internal structural prior, search organization, and the rule by which candidates persist. Table A.1 summarizes these distinctions, and the following subsections provide technical descriptions of the selected methods.

- **Single-Algorithm Component Methods** (Appendix A-B1) iteratively optimize a designated component within a predefined or hierarchical scaffold.
- **Designed-Multiplicity Component Methods** (Appendix A-B2) explicitly aim to produce multiple heuristics, prescribing diversity through multi-objective formulations or instance-coverage criteria.
- **Full-Synthesis Methods** (Appendix A-B3) generate and optimize complete algorithms rather than individual components or functions.

ATLAS Distinction: ATLAS combines full-algorithm synthesis from a problem specification under a minimal I/O interface with no prescribed internal decomposition, then applies embedding-distance-aware archive maintenance to preserve coverage across search regions. The following subsections describe representative methods in each category. Direct comparisons with ATLAS are concentrated on designed-multiplicity and full-synthesis approaches, where differences in output purpose, synthesis scope, and structural assumptions require closer examination.

TABLE A.1: *Comparative analysis of LLM-based algorithm synthesis methods.* **Scope** distinguishes component optimization from full-algorithm synthesis. **Organizing space** indicates where diversity/selection pressure is applied. **Preservation logic** states the operational rule by which candidates persist in the search state; these mechanisms are reported descriptively without imposing a hierarchy among them.

Method	Scope	Primary Mechanism	Organizing Space	Preservation Logic
Component-Based Single-Algorithm Synthesis (Appendix A-B1)				
FUNSEARCH [36]	Component	Islands + best-shot prompting	Output signatures (eval scores)	Cluster-biased sampling + island resets
EOH [25]	Component	Operator-based evolution	Fitness (scalar performance)	Fitness-based selection; no explicit diversity objective
REEVO [57]	Component	Verbal gradients + reflection	Fitness + meta-objective constraints	Successful candidates under meta-objective constraints
HSEVO [9]	Component	Harmony Search + reflection	Embeddings (diversity indices) + fitness	Embedding diversity indices (SWDI/CDI) modulate search
MCTS-AHD [61]	Component	MCTS with LLM actions	Genealogy (tree) + UCT scores	Tree retention with UCT-based revisiting
CALM [18]	Component	RL fine-tuning (GRPO) + pool evolution	Fitness rewards (policy updates)	Pool with stagnation-triggered collapse; diversity not explicit
HEURAGENIX [55]	Component[1]	Contrastive heuristic evolution + online selection	Fixed state–operator interface + heuristic pool	Interchangeable heuristic pool under a predefined $H : \mathcal{Z} \to \mathcal{O}$ interface
MTHS [29]	Hierarchical Component[2]	Two-level multi-task evolution + cross-task transfer	Task-score vectors + task-specific program populations	Task champions + Pareto fronts; per-task best-program retention
Component-Based Designed-Multiplicity Synthesis (Appendix A-B2)				
MEOH [56]	Component	NSGA-II + dominance-based prompting	Objective space (quality vs. runtime)	Non-dominated sorting + crowding distance
EOH-S [27]	Component	Portfolio optimization (CPI)	Instance-wise performance vectors	Greedy marginal CPI + instance-space niche coverage
Full-Synthesis (Appendix A-B3)				
LLAMEA [48]	Full Metaheuristic[3]	$(1{+}1)/(1{,}1)$ full-algorithm evolution	Scalar BBOB performance [13] + execution feedback	Best/latest-parent retention
ALPHAEVOLVE [33]	Scaffolded Full Algorithm[4]	Patch-based evolution over annotated blocks	Program variants (MAP-Elites-inspired archive)	Archive retention with evolutionary sampling
A2DEPT [4]	Structured Full Algorithm[5]	Program-lineage tree + hierarchical edits	Genealogy, scalar fitness, and operator feedback	SA acceptance + diversity-aware history/partner sampling
ATLAS (Ours)	**Scaffold-Free Full Algorithm[6]**	**Full-algorithm operators + semantic quality-diversity archiving**	**Embedding space (semantic structure) + fitness**	**Embedding-distance-aware pruning preserves coverage across archive regions**

[1] HEURAGENIX evolves heuristics within a fixed hyper-heuristic scaffold with predefined state-operator interface.
[2] MTHS instantiates a template-structured, task-agnostic metaheuristic as complete task-specific programs, then identifies and evolves one key function per task; complete programs support execution and transfer, but task-specific refinement is component-level.
[3] The published LLAMEA instantiation generates complete optimizer classes under a fixed callable interface for box-constrained continuous BBOB [13].
[4] ALPHAEVOLVE evolves programs by applying patches within user-marked evolution blocks; code outside blocks constrains interfaces and execution flow.
[5] A2DEPT represents a complete algorithm as a fixed preface plus a role-partitioned function registry; its macro operator can rewrite the entry-point workflow and introduce helper functions.
[6] ATLAS receives the objective, constraints, and valid-solution requirements as the problem specification; its algorithm-facing interface fixes the callable entry point and instance/solution I/O formats without prescribing the algorithm's internal decomposition or control flow.

1) Component-Based Single-Algorithm Synthesis: This category encompasses methods whose iterative search targets a designated heuristic component within a predefined algorithmic framework or structured hierarchy. Prominent examples include FUNSEARCH [36], EOH [25], REEVO [57], MCTS-AHD [61], CALM [18], HEURAGENIX [55], and MTHS [29]. While these approaches use different search mechanisms, they retain a prescribed scaffold or component-level refinement target, such as a priority function, scoring rule, operator, or identified key function.

a) FUNSEARCH*:* FUNSEARCH [36] is an evaluator-guided program discovery framework that uses a pretrained code LLM to iteratively improve a target function within a user-provided program. The user provides an `evaluate` routine (the scoring oracle) together with an initial implementation; in practice, FUNSEARCH is most naturally applied when the program serves as a *scaffold* and the search is focused on a critical function (e.g., a priority rule in a greedy algorithm), allowing the LLM to optimize the key design choice within a fixed algorithmic structure. *Best-shot prompting* samples k programs from a population,

inserts them as versioned functions (e.g., `priority_v0`, `priority_v1`), and appends an empty `priority_v2` header for the LLM to complete, encouraging reuse and recombination of ideas across sampled candidates.

To sustain exploration, FUNSEARCH maintains a programs database evolved with an *islands model*, in which sub-populations evolve independently and are periodically reset by discarding the worst-performing half of islands and reseeding them from the best programs of surviving islands. Within each island, candidates are clustered by a *functional signature*, defined as the tuple of scores over the evaluation inputs, and sampling proceeds by first choosing a signature cluster (biased toward higher score) and then selecting shorter programs within that cluster. This yields a form of behavioral diversity, but the induced partition remains tied to observed outputs on a finite evaluation set.

*b) Evolution of Heuristics (*EOH*):* EOH [25] introduces an LLM-guided evolutionary framework for heuristic design in which each candidate is represented by both a concise natural-language *thought*, describing the core algorithmic idea, and an executable code implementation (typically a Python function) that realizes it. Candidates are evaluated by executing their code on training instances to obtain a scalar fitness, and new candidates are generated through five LLM-driven operators. Three operators refine individual heuristics through different mutation-style transformations, while two exploration operators either combine information from multiple parents or generate more novel variants. Population updates are then driven by fitness-based selection, such as tournament or rank-based selection, which retains stronger candidates and discards weaker ones.

*c) Reflective Evolution (*REEVO*):* REEVO [57] shifts evolutionary search from purely stochastic mutation toward reasoning-guided optimization by introducing *verbal gradients*, namely natural-language critiques that explain why one heuristic outperforms another. The system employs dual LLM roles: a *generator* produces candidate heuristics as executable code, and a *reflector* performs comparative analysis through short-term reflection (pairwise comparison guiding crossover) and long-term reflection (persistent memory of design patterns guiding mutation). To maintain exploration, REEVO selects parent pairs randomly from successful candidates with an explicit constraint to avoid identical meta-objective values, encouraging behavioral diversity and reducing premature convergence toward near-duplicate heuristics. REEVO's reflection mechanism provides explicit directional guidance for code improvement: the LLM receives textual critiques of why particular design choices succeed or fail, which can accelerate local iterative refinement within a fixed scaffold.

*d) Harmony Search Evolution (*HSEVO*):* HSEVO [9] addresses the exploration-exploitation trade-off in LLM-based heuristic synthesis by treating diversity as an explicit optimization concern. Motivated by the empirical observation that earlier methods may preserve higher diversity at the expense of objective performance (e.g., EOH) or achieve stronger objective performance with lower diversity (e.g., REEVO, FUNSEARCH), HSEVO proposes two embedding-based diversity measures: the Shannon-Wiener Diversity Index (SWDI) and Cumulative Diversity Index (CDI). It adapts Harmony Search by maintaining a Harmony Memory updated via *Memory Consideration* (reuse of existing patterns), *Pitch Adjustment* (LLM-guided local variation), and *Random Selection* (novel proposals). HSEVO further augments the reflection-based pipeline with *Flash Reflection*, a batch ranking and distillation procedure, and an HS-based individual tuning stage that extracts tunable parameters (e.g., thresholds, weights) from top individuals, optimizes them via Harmony Search, and reinserts the improved heuristics into the population.

e) Monte Carlo Tree Search for Automatic Heuristic Design (MCTS-AHD): MCTS-AHD [61] replaces population-based evolution with Monte Carlo Tree Search to avoid prematurely discarding temporarily underperforming heuristics. Motivated by the concern that population-based retention rules may eliminate candidates that do not immediately outperform the current worst individual, thereby blocking worse-before-better search trajectories, MCTS-AHD preserves LLM-generated heuristic functions as nodes in an MCTS tree, each associated with executable code and a linguistic description, and iteratively applies selection, expansion, simulation, and backpropagation. The core premise is that weak individuals may still serve as useful stepping stones toward higher-quality solutions. Expansion employs LLM-driven actions analogous to evolutionary operators (initialization, mutation, crossover), together with a distinctive *Tree-Path Reasoning* action in which the LLM analyzes the full evolutionary lineage from the root to the current leaf and synthesizes insights from that history of design modifications. To reduce code-description mismatch, the method uses *thought-alignment*, generating descriptions after code generation rather than before. To balance exploration and convergence, it employs *Progressive Widening* to control the branching factor dynamically and an *Exploration-Decay* mechanism that linearly reduces the exploration weight over time.

*f) Co-evolution of Algorithms and Language Model (*CALM*):* CALM [18] extends LLM-based automatic heuristic design beyond prompt-level evolution by jointly evolving the heuristic pool and the LLM via reinforcement learning. Specifically, CALM integrates *verbal guidance* (operator-driven evolution over a pool of heuristics) with *numerical guidance* by fine-tuning the LLM using Group Relative Policy Optimization (GRPO) [40] from heuristic-performance rewards. The framework maintains a pool of heuristics, each storing an idea, executable code, and performance; at each round it samples an evolutionary operator (injection, replacement, crossover, simplification, or initialization), generates multiple candidate responses, evaluates them, and uses the resulting prompt-response-performance tuples to update the LLM. For most operators, parent heuristics are sampled by performance rank with probability inversely proportional to rank (and heuristics beyond the pool cutoff receive zero probability), and CALM includes a stagnation-triggered *collapse mechanism* that resets the pool by discarding all but the original seed and the current best heuristic to re-initiate exploration.

g) HEURAGENIX*:* HEURAGENIX [55] proposes an LLM-based hyper-heuristic framework with two coupled stages: (i) a contrastive, data-driven heuristic evolution phase that discovers reusable evolution strategies from contrastive solution trajectories,

and (ii) an adaptive problem-solving phase that selects among the evolved heuristic pool online, using either a frontier LLM or a lightweight fine-tuned selector model. The framework is positioned as evolving and deploying heuristics without relying on an external framework, while enabling instance-level adaptation through dynamic heuristic switching.

Despite this end-to-end framing, HEURAGENIX remains component-based from the perspective of synthesis scope because the generated artifacts are constrained to a fixed hyper-heuristic scaffold: a heuristic is explicitly defined as a function mapping a structured *problem state* to an allowable *operation* ($H : \mathcal{Z} \to \mathcal{O}$), which is then executed by a predefined transition function ($T : \mathcal{Z} \times \mathcal{O} \to \mathcal{Z}$). All heuristics in the pool must satisfy a common callable interface and produce operators from a predefined operator family to support runtime switching.

*h) Multi-Task Hierarchical Search (*MTHS*):* MTHS [29] addresses cross-task automated heuristic design through an explicitly scaffolded two-level representation. Each high-level individual contains a task-agnostic metaheuristic expressed through a structured template, with components such as initialization, the main search loop, and post-processing. High-level evolution revises this structured metaheuristic description rather than an unrestricted task-specific source artifact. For each task, an LLM converts that representation into a complete executable implementation under a supplied program template, identifies one performance-critical key function, and conducts low-level evolution by replacing that function while preserving its interface and the surrounding program.

A cross-task transfer stage supplies a successful source-task program together with the target task's template to generate an adapted target program. At the high level, each individual is evaluated by a vector of task-specific scores; population management retains task champions and then applies Pareto-front survival, while the low-level search retains the best task-specific program associated with each metaheuristic. Complete programs are therefore generated for evaluation and transfer, but the task-specific evolutionary refinement remains component-level and the high-level representation prescribes an algorithmic structure. We consequently classify MTHS as scaffolded hierarchical synthesis with key-function refinement, rather than full-algorithm synthesis.

2) Component-Based Multi-Algorithm Synthesis:

*a) Multi-objective Evolution of Heuristics (*MEOH*):* MEOH [56] reformulates automatic heuristic design as a multi-objective optimization problem to address the practical trade-off between solution quality and computational complexity. Arguing that standard methods (e.g., EOH) may neglect runtime efficiency in pursuit of marginal performance gains, MEOH embeds an LLM within an NSGA-II-based evolutionary framework [10] to approximate the Pareto front of heuristic designs. The framework employs a specialized *dominance-based prompting* strategy: the LLM is presented with parent heuristics together with their objective vectors (e.g., optimality gap and runtime) and is instructed to generate offspring that improve the trade-off between the competing objectives. By applying non-dominated sorting and crowding-distance selection, MEOH maintains a diverse population of heuristics, ultimately producing a spectrum ranging from fast, efficient variants to more computationally intensive, high-performance alternatives.

*b) Evolution of Heuristic Set (*EOH-S*):* Observing that a single heuristic may generalize poorly across instance distributions and scales, EOH-S [27] reformulates LLM-based design as Automated Heuristic Set Design: learning a small portfolio that minimizes the Complementary Performance Index (CPI), defined as the average across instances of the best-performing heuristic in the set. Leveraging CPI's monotone-supermodular structure, EOH-S introduces Complementary Population Management, which greedily selects heuristics by marginal CPI gains, together with a diversity-aware memetic search that chooses parents with maximal Manhattan distance in instance-wise performance space so as to synthesize heuristics covering different niches.

MEOH and EOH-S both target multiplicity as an explicit output objective: respectively, a Pareto front over quality and runtime, or a complementary portfolio over instance regimes. ATLAS does not optimize either of those output criteria. Instead, it explicitly prescribes embedding-space coverage as a search-state preservation rule. The identities and algorithmic mechanisms of the retained candidates emerge during synthesis, while similarity-based pruning, neighborhood retrieval, and cross-region selection determine how that discovered repertoire is maintained and revisited.

3) Full-Synthesis: Full-synthesis approaches remain relatively underexplored and instantiate the setting with different domains, external interfaces, and internal structural priors. LLAMEA [48] evolves complete metaheuristics under a fixed continuous black-box optimizer interface; ALPHAEVOLVE [33] edits marked regions of user-provided programs; and the concurrent A2DEPT [4] evolves complete algorithms through an explicit function-level representation and hierarchical edit operators. ATLAS combines a problem-specific combinatorial I/O interface with internally undecomposed full-algorithm operators and embedding-space coverage preservation.

a) LLAMEA*:* LLAMEA [48] is a generate–evaluate–refine framework whose published instantiation synthesizes complete Python metaheuristics for box-constrained continuous black-box optimization. Its outer search uses either elitist $(1{+}1)$ selection, which returns the best-so-far algorithm to the LLM, or non-elitist $(1{,}1)$ selection, which returns the latest algorithm. In both cases the refinement prompt contains the selected algorithm's complete code, aggregate performance, variability, and execution errors, and asks the LLM to refine or redesign it. The prompt also lists the names and mean performance scores of previously generated algorithms, providing a compact summary of the explored search history; however, only the selected algorithm is supplied in full as the parent for refinement. The LLM therefore chooses the effective edit scale and can change parameters, operators, and higher-level interactions inside the complete algorithm.

The framework is generic, but the published study focuses on the BBOB functions [13] in a homogeneous real-vector search space with box constraints, a fixed function-evaluation budget, and a prescribed callable optimizer interface. Its $(1+1)$ and $(1,1)$ search schemes maintain a single active evolutionary trajectory, with either the best-so-far or the most recent algorithm serving as the complete parent design. Although the names and mean scores of earlier algorithms provide lightweight historical context, their complete designs are not retained as simultaneous parents for continued refinement or recombination. This organization is simple, but it does not simultaneously preserve and refine multiple competing algorithm families. That limitation can become important in broad full-algorithm design spaces containing alternative component inventories, multiple interacting components, and opportunities for hybridization, because a promising but temporarily inferior design may be discarded before it can be further refined or combined with other designs. LLaMEA therefore establishes the feasibility of full-algorithm LLM evolution in its demonstrated setting, while leaving open how population- or archive-based search could improve coverage and refinement in more heterogeneous design spaces.

b) AlphaEvolve*:* AlphaEvolve [33] is an evolutionary coding agent for scientific and algorithmic discovery that operates over complete code artifacts, potentially spanning multiple functions or components, using one or more automated evaluators that return scalar feedback signals. The user provides an initial codebase and explicitly marks editable regions with evolution-block annotations, while code outside those regions remains fixed and provides the surrounding execution structure. Candidate updates are generated by an ensemble of Gemini models and are applied as structured code edits before evaluation. The public description also highlights support for expensive objectives through staged evaluation cascades, optional LLM-based auxiliary judging, and large-scale parallel evaluation.

A key positioning claim of AlphaEvolve is that it extends beyond single-function program search by enabling coordinated evolution over larger code regions and richer code contexts. At the same time, its notion of full synthesis is most accurately understood as *block-wise evolution of a supplied code artifact*: the search space is still determined by the user-provided codebase, the selected editable regions, and the surrounding fixed interfaces and execution flow. In this sense, AlphaEvolve goes beyond component-level function evolution, but it does not operate in a scaffold-free setting.

AlphaEvolve was initially introduced through a public white paper and is now offered as a Google Cloud product [16]; its underlying implementation remains closed source.

c) A2DEPT*:* A2DEPT [4] was developed concurrently with ATLAS and is therefore treated here as parallel work rather than prior work. It targets the same broad setting of complete executable algorithm synthesis for combinatorial optimization. Its "program tree" is a search genealogy whose nodes are complete algorithm variants, rather than a syntax tree of components. Within each node, however, A2DEPT imposes an explicit function-level representation: a fixed preface is paired with a function registry that an LLM partitions into immutable definitions and mutable strategies. Micro-tuning edits one mutable strategy while preserving its interface; macro-mutation rewrites the entry-point workflow and may introduce new helper functions; semantic crossover combines parent strategies; and call-graph analysis repairs missing dependencies and prunes unreachable code.

Thus, A2DEPT is not component synthesis, nor does it fix the number of components: it is full-algorithm synthesis with a strong function-level inductive bias and explicit edit granularities. In this respect it shares with AlphaEvolve the use of additional structure to guide full-algorithm editing, although the structures differ: AlphaEvolve relies on user-marked editable blocks in a supplied codebase, whereas A2DEPT generates complete initial algorithms and can rewrite their entry-point workflows. ATLAS takes the complementary design choice of supplying complete algorithm source directly to general operators, without a mutable/immutable role partition or a prescribed function hierarchy, and delegates identification and coordinated revision of internal mechanisms to the LLM.

ATLAS's released illustrative algorithms provide qualitative evidence that this undecomposed full-algorithm representation can yield coordinated multi-function algorithms: for example, the CVRP algorithm connects alternative construction procedures, savings merging, regret repair, relocation, 2-opt, large-neighborhood search, and simulated-annealing acceptance in one executable workflow. This demonstrates that explicit component labels are not required to realize multi-component designs in the studied setting; it does not establish that explicit structural guidance cannot improve sample efficiency or reliability elsewhere. Similarly, ATLAS could add a prompt policy that focuses an edit on a selected function or on cross-function interactions without changing its archive, evaluator, or three-layer search, but the current work does not evaluate that specialized policy. No public A2DEPT implementation was available at the time of writing, so we restrict this comparison to the methodological level and do not include an empirical comparison.

Appendix B
Implementation Details

The exact prompts used by ATLAS are maintained with the executable implementation in the public source-code repository. Each LLM request combines a system instruction with a user message assembled in `framework/prompt_system.py` from the operator objective, problem specification, external I/O requirements, structured response requirements, and each selected reference's complete name, description, and source code when applicable. The problem-specific components are defined in `framework/problem_definitions.py`. The implementation also records the numerical examples supplied to Create and the failure-conditioned guidance supplied to Repair.

A. Formal Problem Definitions

This section provides formal mathematical formulations for the four benchmark problems.

1) Permutation Flow Shop Scheduling (FSS): **Verbal definition:** In the permutation flow shop scheduling problem, n jobs must be processed on m machines in a fixed sequential order. All jobs visit machines in the same sequence: Machine $1 \rightarrow$ Machine $2 \rightarrow \ldots \rightarrow$ Machine m. The goal is to find a job permutation that minimizes the makespan (total completion time).

Notation: Let $\mathcal{J} = \{1, \ldots, n\}$ be the set of jobs and $\mathcal{M} = \{1, \ldots, m\}$ be the set of machines in processing order. Let $p_{j,k} \geq 0$ denote the processing time of job $j \in \mathcal{J}$ on machine $k \in \mathcal{M}$.

Decision variable: A permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of jobs, where $\pi_\ell \in \mathcal{J}$ denotes the job scheduled in position ℓ. This ordering applies to all machines.

Completion times: Let $C_{\ell,k}$ denote the completion time of the job in position ℓ on machine k. With boundary conditions $C_{0,k} = 0$ for all k and $C_{\ell,0} = 0$ for all ℓ, completion times are computed recursively:

$$C_{\ell,k} = \max(C_{\ell-1,k}, C_{\ell,k-1}) + p_{\pi_\ell,k} \quad \text{for } \ell = 1, \ldots, n,\ k = 1, \ldots, m. \tag{B.1}$$

Objective function: Minimize the makespan:

$$\min_{\pi}\ C_{\max}(\pi) = C_{n,m}. \tag{B.2}$$

Constraints:

- **Permutation:** π is a valid permutation of $\mathcal{J}$.
- **Machine sequence:** All jobs visit machines in order $1 \rightarrow 2 \rightarrow \cdots \rightarrow m$.
- **Machine capacity:** Each machine processes at most one job at a time (implicitly enforced by the sequential structure of π and Eq. (B.1)).
- **Non-preemption:** Operations cannot be interrupted once started.
- **Precedence:** Job π_ℓ cannot start on machine $k+1$ until it completes on machine k (enforced by Eq. (B.1)).

Implementation note: In the released prompt implementation, jobs and machines are 0-indexed (0 to $n-1$ and 0 to $m-1$), matching Python conventions. The mathematical formulation above uses 1-indexing for notational clarity; the two representations are equivalent.

2) Capacitated Vehicle Routing Problem (CVRP): **Verbal definition:** A fleet of homogeneous vehicles with capacity Q must serve n customers from a central depot. Each customer has a demand and geographic coordinates. All vehicles start and end at the depot. The goal is to construct routes that visit all customers while respecting vehicle capacity constraints and minimizing total distance traveled.

Notation: Let $V = \{1, \ldots, n\}$ be the set of customers and node 0 denote the depot. Let $p_i = (x_i, y_i) \in \mathbb{R}^2$ be the coordinates of node $i \in \{0\} \cup V$. Define Euclidean distance:

$$c_{ij} = \|p_i - p_j\|_2 = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{B.3}$$

Each customer $i \in V$ has demand $d_i > 0$, and vehicle capacity is Q.

Decision variables: A solution is a set of routes $\mathcal{R} = \{r^{(1)}, \ldots, r^{(K)}\}$ where each route $r^{(k)} = (v_1^{(k)}, \ldots, v_{m_k}^{(k)})$ is an ordered sequence of distinct customers with $v_\ell^{(k)} \in V$. The number of routes K is not fixed a priori (unlimited fleet).

Objective function: The cost of route $r = (v_1, \ldots, v_m)$ is:

$$\text{cost}(r) = c_{0,v_1} + \sum_{\ell=1}^{m-1} c_{v_\ell, v_{\ell+1}} + c_{v_m,0}. \tag{B.4}$$

Minimize total distance:

$$\min_{\mathcal{R}} \sum_{r \in \mathcal{R}} \text{cost}(r). \tag{B.5}$$

Constraints: For a route r, let $\text{cust}(r)$ denote the set of customers appearing in r.

$$\textbf{(Coverage)} \qquad \bigcup_{r \in \mathcal{R}} \text{cust}(r) = V, \quad \text{cust}(r) \cap \text{cust}(r') = \emptyset\ \forall r \neq r' \tag{B.6}$$

$$\textbf{(Capacity)} \qquad \sum_{v \in \text{cust}(r)} d_v \leq Q \quad \forall r \in \mathcal{R} \tag{B.7}$$

Implementation note: In the released prompt implementation, customers are 0-indexed (0 to $n-1$), and the depot is not included in route representations; it is implicit that all routes start and end at the depot. This formulation instead uses customer indices 1 to n together with an explicit depot node 0 for mathematical clarity.

3) CVRP with Time Windows (CVRPTW): **Verbal definition:** CVRPTW extends CVRP with temporal constraints. Each customer has a time window $[e_i, l_i]$ during which service must begin and a service time s_i. The depot has operating hours $[e_0, l_0]$. Vehicles travel at unit speed (travel time equals Euclidean distance). Vehicles may wait if arriving early, but starting service after a time window closes renders the route infeasible. All routes must return to the depot before it closes.

Notation: In addition to CVRP notation, each customer $i \in V$ has service time $s_i \geq 0$ and time window $[e_i, l_i]$. The depot (indexed as node 0) has time window $[e_0, l_0]$ and zero service time ($s_0 = 0$), so depot feasibility is checked on return arrival time. Travel time equals distance: $t_{ij} = c_{ij}$.

Decision variables: Same as CVRP: a set of routes $\mathcal{R} = \{r^{(1)}, \ldots, r^{(K)}\}$, where each route $r^{(k)} = (v_1^{(k)}, \ldots, v_{m_k}^{(k)})$ is an ordered sequence of customer nodes $v_\ell^{(k)} \in V = \{1, \ldots, n\}$.

Timing dynamics: For route $r = (v_1, \ldots, v_m)$, let τ_i denote the arrival time at node i and σ_i denote the service start time. Vehicles depart the depot at time $\tau_0^{\text{dep}} = 0$ (since $e_0 = 0$ in our instances). The dynamics are:

$$\tau_{v_1} = \tau_0^{\text{dep}} + t_{0,v_1}, \quad \sigma_{v_1} = \max(\tau_{v_1}, e_{v_1}) \tag{B.8}$$

$$\tau_{v_\ell} = \sigma_{v_{\ell-1}} + s_{v_{\ell-1}} + t_{v_{\ell-1},v_\ell}, \quad \sigma_{v_\ell} = \max(\tau_{v_\ell}, e_{v_\ell}) \text{ for } \ell = 2, \ldots, m \tag{B.9}$$

Depot return time:

$$\tau_0^{\text{return}} = \sigma_{v_m} + s_{v_m} + t_{v_m,0}. \tag{B.10}$$

Objective function: Minimize total distance (same as Eq. (B.5)):

$$\min_{\mathcal{R}} \sum_{r \in \mathcal{R}} \text{cost}(r). \tag{B.11}$$

Constraints: For a route r, let $\text{cust}(r)$ denote the set of customers appearing in r.

(Coverage)	Same as Eq. (B.6)	
(Capacity)	Same as Eq. (B.7)	
(Customer time windows)	$e_i \leq \sigma_i \leq l_i \quad \forall r \in \mathcal{R},\ \forall i \in \text{cust}(r)$	(B.12)
(Depot return)	$\tau_0^{\text{return}} \leq l_0 \quad \forall r \in \mathcal{R}$	(B.13)

Important notes:

- Time windows are **inclusive**: $[e_i, l_i]$ means service start at exactly e_i or l_i is acceptable.
- The time-window constraint applies to **service start** σ_i. Service may extend beyond l_i as long as $\sigma_i \leq l_i$.
- Waiting is allowed: if $\tau_i < e_i$, then service starts at $\sigma_i = e_i$.
- Infeasibility occurs when $\sigma_i > l_i$.
- **Index mapping:** Here the depot is node 0 and customers are nodes $\{1, \ldots, n\}$. In the released prompt implementation, customers are indexed in `customer_coords` as $\{0, \ldots, n-1\}$ and routes contain only these customer-list indices; the depot is represented separately via `depot_coords` and `depot_time_window`.

4) Quadratic Assignment Problem (QAP): **Verbal definition:** Assign n facilities to n locations to minimize total interaction cost. The cost depends quadratically on the assignment: if facility i is assigned to location p and facility j is assigned to location q, their contribution to the total cost is $f_{ij} \cdot d_{pq}$, where f_{ij} is the flow between facilities and d_{pq} is the distance between locations.

Notation: Let $\mathcal{F} = \{1, \ldots, n\}$ be the set of facilities and $\mathcal{L} = \{1, \ldots, n\}$ be the set of locations. Let $F = [f_{ij}]$ be the $n \times n$ flow matrix (symmetric with $f_{ii} = 0$) and $D = [d_{pq}]$ be the $n \times n$ distance matrix (symmetric with $d_{pp} = 0$).

Decision variable: A permutation $\pi : \mathcal{F} \to \mathcal{L}$, where $\pi(i)$ denotes the location assigned to facility i.

Objective function: Minimize the total quadratic cost:

$$\min_{\pi} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i),\pi(j)}. \tag{B.14}$$

Equivalently, exploiting symmetry:

$$\min_{\pi} \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2 \cdot f_{ij} \cdot d_{\pi(i),\pi(j)}. \tag{B.15}$$

Constraints:

$$\pi \text{ is a bijection from } \mathcal{F} \text{ to } \mathcal{L}. \tag{B.16}$$

Implementation note: In the released prompt implementation, facilities and locations are 0-indexed (0 to $n-1$). The assignment $\pi[i] = p$ means facility i is placed at location p. The evaluator computes cost by summing over all pairs as in Eq. (B.14). Since $f_{ii} = 0$, diagonal terms ($i = j$) contribute zero to the sum.

B. Benchmark Instance Generation

We generate synthetic instances for each problem using benchmark generators based on the LLM4AD framework [26]. For CVRP, CVRPTW, and QAP, we follow the corresponding LLM4AD generators. For FSS, however, LLM4AD uses COBench predefined instances [43], whereas we use a synthetic generator for FSS for consistency with the other benchmark problems; we therefore replace the FSS benchmark source with the generator described in Appendix B-B3.

For each benchmark setting, the protocol generates $N_{\text{train}} = 31$ training instances and $N_{\text{test}} = 31$ independently generated test instances. The corresponding split seeds are 2024 and 42. The training split is the only split accessible to synthesis and archive construction; the test split is first accessed after the selected algorithm has been frozen.

Interpretation of train–test objective values: Differences between mean training and test objective values in our setting should not be interpreted in the same way as train–test loss gaps in supervised learning. The splits contain different sampled optimization instances, and any split may have a higher or lower mean raw objective simply because its instances are intrinsically harder or have higher optimal objective values. Accordingly, generalization is assessed by performance on unseen test instances, including comparison against human-designed, domain-specific baselines on those same test instances, rather than by the raw gap between mean train and mean test objective values. Even a method that always finds the global optimum on every instance could exhibit either higher or lower average objective values on the train and test splits if the underlying instances differ in their optimal objective scales.

1) CVRP: Each instance contains a depot and n customers. Coordinates are sampled i.i.d. uniformly in $[0, 1]^2$ for all $n + 1$ nodes (depot + customers). Customer demands are sampled independently as integers in $\{1, \ldots, 9\}$, and the depot demand is set to 0. The vehicle capacity is fixed to $Q = 40$. Distances are Euclidean.

Parameters (default): $n = 50$, $Q = 40$, coordinate range $[0, 1]$, demand range $\{1, \ldots, 9\}$.

2) CVRPTW: We sample one depot coordinate and n customer coordinates i.i.d. uniformly in $[0, 1]^2$. Customer demands are sampled independently as integers in $\{1, \ldots, 9\}$, and capacity is fixed to $Q = 40$. Travel time equals Euclidean distance.

Service times are sampled for customers i.i.d. from $\mathcal{U}(0.15, 0.20)$, with depot service time set to 0. The depot time window is fixed to $[0, T_{\max}]$ with $T_{\max} = 4.6$. Customer time windows follow the LLM4AD construction: for each customer i, a window length $L_i \sim \mathcal{U}(0.15, 0.20)$ is sampled, and the early time is set as $e_i = \alpha_i \, d_{0i}$, where d_{0i} is the distance from the depot to customer i, and

$$\alpha_i = 1 + U_i \cdot \left(\frac{T_{\max} - s_i - L_i}{d_{0i}} - 2 \right), \quad U_i \sim \mathcal{U}(0, 1),$$

with s_i being the service time of customer i. The late time is $l_i = e_i + L_i$.

Parameters (default): $n = 50$, $Q = 40$, coordinate range $[0, 1]$, demand range $\{1, \ldots, 9\}$, $T_{\max} = 4.6$, service time $\mathcal{U}(0.15, 0.20)$, window length $\mathcal{U}(0.15, 0.20)$.

3) Permutation Flow Shop Scheduling (FSS): For each instance, processing times are sampled i.i.d. as integers in $\{10, \ldots, 99\}$ for each job–machine pair, then cast to float for evaluation. We consider permutation flow shop: a solution is a permutation of the n jobs applied across all m machines.

Parameters (default): $n = 50$ jobs, $m = 10$ machines, processing times in $\{10, \ldots, 99\}$.

4) Quadratic Assignment Problem (QAP): We generate a symmetric flow matrix $F = [f_{ij}] \in \mathbb{Z}_+^{n \times n}$ and a symmetric distance matrix $D = [d_{pq}] \in \mathbb{Z}_+^{n \times n}$. To do so, we first sample integer matrices $\tilde{F}$ and $\tilde{D}$ with entries in $\{1, \ldots, 100\}$, then symmetrize them via elementwise floor division:

$$F = \left\lfloor \frac{\tilde{F} + \tilde{F}^\top}{2} \right\rfloor, \qquad D = \left\lfloor \frac{\tilde{D} + \tilde{D}^\top}{2} \right\rfloor,$$

and set their diagonals to zero. The objective is computed by summing over all ordered pairs as in Eq. (B.14).

Parameters (default): $n = 50$, matrix entries in $\{1, \ldots, 100\}$, symmetric with zero diagonal.

C. Evaluation Protocol

This subsection specifies the evaluation protocol, including evaluator design, synthesis budgets, runtime caps, selection of the deployable algorithm, test execution, reporting, and statistical testing. Unless otherwise stated, experiments use the default benchmark settings defined in the main experimental setup and the train/test generation procedure in Appendix B-B.

1) Evaluators: All methods are evaluated on the same benchmark instances for each problem, but the evaluation interface depends on the synthesis setting. For component-based LLM baselines (EoH, ReEvo, and MCTS-AHD), evaluation follows the shared scaffolded setup aligned with LLM4AD [26], where the synthesized artifact is a heuristic component executed inside a fixed external framework. In this setting, parts of the solution-construction and constraint-handling logic are supplied by the scaffold. The underlying constraints remain properties of the problem; the scaffold supplies algorithmic machinery for handling them.

For full-synthesis methods (ATLAS and EoH-Full), the synthesized artifact is a complete executable algorithm under the stated I/O interface and execution environment. We therefore use evaluators designed for full algorithms, which execute the synthesized program end-to-end on each benchmark instance and verify both objective quality and problem-specific feasibility

conditions, including constraint satisfaction and output-format validity. ATLAS and EoH-Full use identical evaluators, which are provided in the ATLAS codebase. Although the evaluator interfaces differ between component-based and full-synthesis settings, they produce identical objective values for identical feasible solutions; the distinction lies in how candidate solutions are generated and validated, not in the underlying cost computation.

Thus, all methods use the same benchmark instances, but component-based and full-synthesis methods are evaluated through interfaces appropriate to their respective synthesis settings.

2) Synthesis Runs and LLM Configuration: All LLM-based synthesis methods use OpenAI GPT-5-mini with reasoning effort set to *low* and temperature $T = 1.0$.

LLM-based methods (ATLAS, EoH, ReEvo, MCTS-AHD, and EoH-Full) perform $R = 5$ independent synthesis runs with different random seeds, each yielding one final optimization algorithm; the algorithms may differ across runs. Since the human-designed, domain-specific baselines do not synthesize algorithms, they are evaluated once on the test instances. Each ATLAS synthesis run yields exactly one deployable algorithm, chosen on training data alone as specified below. Every comparison method likewise contributes one training-selected algorithm per run, subject to the same test protocol.

3) Budgets and Runtime Caps: We use two budget notions in our experiments.

- *Synthesis budgets for LLM-based methods:* For end-to-end ATLAS component ablations and search-configuration sensitivity analyses, we fix the search budget to $B = 500$ evaluated operator executions. The embedding and initialization analyses use their separately reported sample sizes. For cross-method comparisons with other LLM-based baselines, we instead match total token consumption per synthesis run, since per-iteration token usage differs substantially between component synthesis and full-algorithm synthesis. The resulting benchmark-specific token budgets are calibrated from the average token usage of 500 evaluated operator executions under full synthesis: 5M tokens for FSS, 6.5M for CVRP, 7M for CVRPTW, and 6M for QAP.
- *Per-instance runtime caps:* For each default benchmark setting, we use a common *per-instance* runtime cap across all compared methods: 210 s for FSS, 30 s for CVRP, 120 s for CVRPTW, and 240 s for QAP. For LLM-based methods, this cap is enforced during synthesis-time evaluation: exceeding the cap produces a failed evaluation rather than a problem-infeasibility judgment. Runtime-sensitive human-designed baselines are evaluated under the same benchmark-setting-specific caps. The same full cap is enforced on training and test executions.

4) Execution Model and Human-Designed Baselines: All compared methods are executed under single-core, single-threaded settings whenever applicable. Each candidate–instance invocation executes within one worker process and one thread, while different instances are evaluated in parallel. Restricting human-designed baselines to the same per-instance execution model makes runtime caps more comparable across methods. Human-designed baselines are evaluated directly on the test instances. NEH runs deterministically to completion. OR-Tools is deterministic in our single-threaded configuration and is evaluated once per test instance. The runtime-sensitive methods including PyVRP, VROOM, the IG variants, Iterative Beam Search, RoTS, SA, BLS, and BMA are each executed once under the same benchmark-setting-specific per-instance runtime cap.

5) Reporting and Statistical Testing:

a) Training-defined ATLAS selection: At the end of each ATLAS run, the algorithm with the best training performance is selected as the output and evaluated on the test instances for reporting the results.

b) Test-set eligibility: The best selected algorithm on train set is executed on all $N_{\text{test}} = 31$ test instances. This stage is evaluation-only: it cannot invoke Repair, add or remove archive members, change embeddings or clusters, or otherwise resume search. A run is successful only if all 31 executions finish within the ordinary problem-specific cap and pass memory, exception, output-format, finite-objective, and problem-feasibility checks.

c) Paired statistical comparison: For each test instance, we average the raw objective values from the five independent runs of each LLM-based method; human-designed methods contribute their direct per-instance raw objectives. This averaging marginalizes over synthesis runs and yields one raw objective value per method and test instance. Pairwise comparisons between all methods use a two-sided Wilcoxon signed-rank test at significance level 0.05 on the resulting 31 pairs of per-instance raw objective values. Within each result table, boldface identifies the statistically best-performing group: the method with the lowest mean and any method not significantly different from it. Every textual claim that one method achieves better test-set objective performance than another is supported by this paired test.

d) Percentage-gap display: For each setting, let $\overline{J}_{m,s}$ be method m's mean objective over the 31 test instances. Among the statistically best-performing human-designed methods, the one with the lowest mean is used as the reference with mean $\overline{J}_{\text{ref},s}$. We report

$$\text{Gap}_{m,s} = 100\,\frac{\overline{J}_{m,s} - \overline{J}_{\text{ref},s}}{\overline{J}_{\text{ref},s}}.$$

This percentage is a descriptive gap to the selected domain-specific reference, not an optimality gap. Statistical testing uses the per-instance raw objective values, not the percentage gaps. Table B.1 lists the resulting reference method and mean $\overline{J}_{\text{ref},s}$ for every benchmark setting; the implementations and configurations of these methods are described in Appendix B-G1.

TABLE B.1: *Raw reference means used for percentage-gap reporting.* For each problem and size, paired tests on raw per-instance objectives identify the statistically best-performing group of human-designed, domain-specific methods; the member of that group with the lowest mean supplies the reported reference. These are empirical reference means on the study instances, not optima or best-known-solution claims. The CVRP $n25$ methods tie at the reported precision.

Problem setting	Reference method	Raw mean objective
FSS $(n50m10)$	IG-TB (Taillard acc.) [38, 45, 12]	3 343.00
CVRP $(n50)$	PyVRP (ILS) [52, 53]	10.49
CVRPTW $(n50)$	PyVRP (ILS) [52, 53]	16.58
QAP $(n50)$	BMA [2]	5 704 446
FSS $(n25m5)$	IG-TB (Taillard acc.) [38, 45, 12]	1 599.84
FSS $(n100m20)$	IG-TB (Taillard acc.) [38, 45, 12]	6 808.45
CVRP $(n25)$	PyVRP (ILS) [52, 53] / VROOM [6, 50]	6.19
CVRP $(n100)$	PyVRP (ILS) [52, 53]	18.46

D. ATLAS: Algorithms and Pseudocode

This section provides the detailed procedural modules implementing the high-level ATLAS workflow in Algorithm 1. The algorithms below specify the control flow, hyperparameter usage, operator-application strategies, and archive-management logic.

1) Main Execution Loop: Algorithm B.1 expands the high-level workflow into the detailed execution loop from initialization through final archive output.

2) Initialization: Algorithm B.2 implements the two-phase bootstrap that establishes initial archive coverage: diverse generation via CREATE (Phase 1), followed by paradigm diversification via DIVERGE (Phase 2).

3) Archive Maintenance: Algorithms B.3 and B.4 manage archive quality and capacity.

a) Deduplication (Algorithm B.3): The two-tier mechanism removes semantic duplicates (strict threshold τ_{strict}) and redundant near-duplicates that offer negligible performance gains (soft threshold τ_{soft} with absolute tie tolerance ϵ).

b) Capacity-Constrained Pruning (Algorithm B.4): When the archive exceeds $N_{\max}$, pruning must remove algorithms to maintain computational feasibility. Fitness-ranked survival can reduce representation of lower-fitness alternatives. ATLAS instead prioritizes *embedding-space coverage* through a similarity-first strategy: it iteratively identifies the most similar pair (a_i, a_j) via distance matrix $\mathbf{D}$ and removes the worse performer. This forces the archive to spread across the semantic space rather than concentrating on high-performing regions. Note that performance is used only to decide *which member* of the closest pair to remove, not to determine *which pair* to prune.

Elite Protection to Resolve Conflict: A pure similarity-first strategy would aggressively prune $\mathcal{G}^*$, as this region around a^* is naturally the densest in the archive. However, these algorithms have survived the configured deduplication thresholds and serve as the refinement anchor for Layer 1's intensive search. Removing them would weaken exploitation capability precisely where it matters most. Therefore, Algorithm B.4 explicitly protects $\mathcal{G}^*$, maintaining both global diversity and local refinement power under capacity constraints. Crucially, this protection is *dynamic*: when a superior algorithm is discovered in a different region, $\mathcal{G}^*$ shifts immediately to the new neighborhood, and the previous elite region (now stripped of protection and often highly dense) becomes a primary target for pruning. This allows ATLAS to reclaim capacity from regions that are no longer the current best while securing refinement power around a newly leading region, without unbounded archive growth.

Note on Algorithm B.4: Algorithm B.4 is presented at a high level. In the configurations considered in this paper, the archive size is always substantially larger than the protected elite region $|\mathcal{G}^*|$, so capacity-constrained pruning always has many removable non-elite pairs available. Consequently, the case in which all candidate pairs are protected does not arise under the intended operating regime; handling such degenerate configurations is an implementation detail omitted from the pseudocode.

4) Three-Layer Search Strategy: Algorithm B.5 specifies the hierarchical search across intensive best-region refinement (Layer 1), distributed regional refinement (Layer 2), and cross-cluster discovery (Layer 3).

a) Note on Intensive Refinement in Layer 1: In many evolutionary algorithms, the selection process yields multiple near-clones of the best individual (a^*), which are then refined independently through variation operators [25] in each generation. This effectively intensifies refinement on a^*. ATLAS enforces strict deduplication, so a^* exists as a unique instance. To compensate, Layer 1 explicitly iterates refinement operators on a^* for $n_{\text{intensify}}$ times.

b) Note on Layer 3 Operator Strategies: Layer 3 applies multi-reference operators to cluster representatives to enable global exploration:

- **Cross-Cluster COMBINE:** Combines representatives from different embedding clusters and instructs the LLM to integrate useful mechanisms from them. The operation is intended to generate hybrids or intermediate designs between represented regions.
- **Best-Distant COMBINE:** Combines the current best algorithm a^* with representatives from other embedding clusters. Unlike Layer 1's local synthesis with nearby candidates, this operation anchors the prompt with a^* while supplying more distant reference context and asking the LLM to integrate complementary ideas.

Algorithm B.1: ATLAS: Main Execution Loop

Input: Problem & Model: $\mathcal{P}$ (problem), $\mathcal{M}$ (LLM)
Input: Search Budget: B (iterations)
Input: Archive: $N_{\max}$ (capacity), τ_{strict} and τ_{soft} (similarity thresholds), ϵ (performance tolerance)
Input: Layer 1 Neighborhood: k (kNN size)
Input: Initialization: $n_{\text{init}}^{\text{crt}}$ (Phase 1 count), $n_{\text{init}}^{\text{div}}$ (Phase 2 count)
Input: Operator Counts: $n_{\text{intensify}}$ (refine repeats on a^*)
Input: Reference Sizes: m_{comb} (`Combine` refs), m_{div} (`Diverge` refs)
Input: Operator Probabilities: p_{imp}, p_{tune}, p_{simp} (refinement ops in Layers 1 & 2)
Output: Best Algorithm a^*, Archive $\mathcal{A}$, and Cluster Representatives $\mathcal{T}$

`// 1. Initialization (See Alg. B.2)`
$\mathcal{A} \leftarrow \texttt{InitializeArchive}(\mathcal{P}, \mathcal{M}, n_{\text{init}}^{\text{crt}}, n_{\text{init}}^{\text{div}}, m_{\text{div}})$
`// Compute initial distance matrix`
Compute pairwise distance matrix $\mathbf{D}$
$\mathcal{A}, \mathbf{D} \leftarrow \texttt{ArchiveDeduplication}(\mathcal{A}, \mathbf{D}, \tau_{\text{strict}}, \tau_{\text{soft}}, \epsilon)$ `// Deduplicate initialization archive`
while *Budget* B *not exhausted* **do** `// Main Search Loop`
  `// 2. Layer Configuration from the Maintained Archive`
  $a^* \leftarrow \arg\min_{a \in \mathcal{A}} J(a)$ `// where` $J(a) = a.J$
  $\mathcal{G}^* \leftarrow \{a^*\} \cup \text{kNN}(a^*, \mathcal{A}, k, \mathbf{D})$ `// Elite group (Layer 1)`
  $\{C_1, \ldots, C_K\} \leftarrow \texttt{Cluster}(\mathcal{A}, \mathbf{D})$ `// Cluster entire archive`
  $C_{\text{best}} \leftarrow \{C_k : a^* \in C_k\}$ `// Cluster containing` a^*
  Remove $\mathcal{G}^*$ from the remaining clusters and discard empty clusters
  $\Omega \leftarrow \{(C_i, r_i) : C_i \neq C_{\text{best}},\ C_i \neq \emptyset,\ r_i = \arg\min_{a \in C_i} J(a)\}$ `// Re-select representatives after elite removal`
  `// 3. Three-Layer Candidate Generation (See Alg. B.5)`
  $\mathcal{C} \leftarrow \texttt{GenerateCandidates}(\mathcal{G}^*, \Omega, a^*, m_{\text{comb}}, m_{\text{div}}, n_{\text{intensify}}, p_{\text{imp}}, p_{\text{tune}}, p_{\text{simp}})$
  `// 4. Candidate Evaluation & Archive Update`
  foreach $c \in \mathcal{C}$ **do**
    $c.J, c.\text{error} \leftarrow \texttt{Evaluate}(c, \mathcal{I}_{\text{train}})$
    if $c.error \neq null$ **then**
      $c \leftarrow \texttt{Repair}(c, c.\text{error}, \mathcal{P}, \mathcal{M})$
      $c.J, c.\text{error} \leftarrow \texttt{Evaluate}(c, \mathcal{I}_{\text{train}})$
    if $c.error = null$ **then**
      $c.\text{embed} \leftarrow \texttt{ComputeEmbedding}(c)$
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{c\}$
  `// Update distances for newly added algorithms`
  Update distance matrix $\mathbf{D}$ for new algorithms
  $\mathcal{A}, \mathbf{D} \leftarrow \texttt{ArchiveDeduplication}(\mathcal{A}, \mathbf{D}, \tau_{\text{strict}}, \tau_{\text{soft}}, \epsilon)$ `// See Alg. B.3`
  $a^* \leftarrow \arg\min_{a \in \mathcal{A}} J(a)$
  $\mathcal{G}^* \leftarrow \{a^*\} \cup \text{kNN}(a^*, \mathcal{A}, k, \mathbf{D})$ `// Protect the updated best neighborhood`
  if $|\mathcal{A}| > N_{\max}$ **then**
    $\mathcal{A}, \mathbf{D} \leftarrow \texttt{ArchiveSizeManagement}(\mathcal{A}, \mathbf{D}, N_{\max}, \mathcal{G}^*)$ `// See Alg. B.4`
`// Final cluster representatives`
$\{C_1, \ldots, C_K\} \leftarrow \texttt{Cluster}(\mathcal{A}, \mathbf{D})$ `// Cluster the final archive`
$\mathcal{T} \leftarrow \{\arg\min_{a \in C_k} J(a) : k = 1, \ldots, K\}$
return $a^*, \mathcal{A}, \mathcal{T}$

- **DIVERGE Includes a^*:** The DIVERGE operator explicitly includes a^* in the reference set $\mathcal{R}$. This complements Layer 1, which already concentrates refinement effort around the current best region. Including a^* as a reference instructs the LLM to avoid variants too close to that already sampled region and to propose an alternative separated from the referenced approaches.

E. ATLAS: Configuration and Hyperparameters

1) Embedding Strategy: **Chosen representation (early fusion + MGTE):** Based on the embedding representation analysis in Appendix C-A, we represent each algorithm a using a *single* embedding produced by MGTE-LARGE-EN-V1.5 [58] from the concatenation of its *name* n, *description* d, and *preprocessed code* c, where comments, docstrings, and blank lines are removed. Similarity is computed in this embedding space using cosine similarity under the early-fusion formulation in Eq. (C.3).

In Appendix C-A, the *early-fusion* + MGTE configuration achieved the highest Class Cohesion Index (Eq. (C.8)) and the second-best accuracy. Compared with the top-accuracy variant, this representation is simpler, more practical, and based on an

Algorithm B.2: Module: Two-Phase Initialization

Input: $\mathcal{P}$ (problem), $\mathcal{M}$ (LLM), $n_{\text{init}}^{\text{crt}}$ (Phase 1 count), $n_{\text{init}}^{\text{div}}$ (Phase 2 count), m_{div} (Phase 2 refs)
Output: Initial Archive $\mathcal{A}$

$\mathcal{A} \leftarrow \emptyset$
// Phase 1: Bootstrapping via Create
for $i = 1$ **to** n_{init}^{crt} **do**
  $a \leftarrow \texttt{Create}(\mathcal{P}, \mathcal{M})$
  $a.J, a.\text{error} \leftarrow \texttt{Evaluate}(a, \mathcal{I}_{\text{train}})$
  if $a.error \neq null$ **then**
    $a \leftarrow \texttt{Repair}(a, a.\text{error}, \mathcal{P}, \mathcal{M})$
    $a.J, a.\text{error} \leftarrow \texttt{Evaluate}(a, \mathcal{I}_{\text{train}})$
  if $a.error = null$ **then**
    $a.\text{embed} \leftarrow \texttt{ComputeEmbedding}(a)$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{a\}$
// Phase 2: Paradigm Diversification via DIVERGE
for $i = 1$ **to** n_{init}^{div} **do**
  Randomly sample references $\mathcal{R} \subset \mathcal{A}$ of size m_{div}
  $a \leftarrow \texttt{Diverge}(\mathcal{R}, \mathcal{P}, \mathcal{M})$
  $a.J, a.\text{error} \leftarrow \texttt{Evaluate}(a, \mathcal{I}_{\text{train}})$
  if $a.error \neq null$ **then**
    $a \leftarrow \texttt{Repair}(a, a.\text{error}, \mathcal{P}, \mathcal{M})$
    $a.J, a.\text{error} \leftarrow \texttt{Evaluate}(a, \mathcal{I}_{\text{train}})$
  if $a.error = null$ **then**
    $a.\text{embed} \leftarrow \texttt{ComputeEmbedding}(a)$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{a\}$
return $\mathcal{A}$

Algorithm B.3: Module: Archive Deduplication

Input: Archive $\mathcal{A}$, distance matrix $\mathbf{D}$, τ_{strict} (strict similarity threshold), τ_{soft} (soft similarity threshold), ϵ (performance tolerance)
Output: Deduplicated Archive $\mathcal{A}$ and updated $\mathbf{D}$

// Two-Tier Deduplication Strategy
foreach *pair* $(a_i, a_j) \in \mathcal{A} \times \mathcal{A}$ *with* $i < j$ **do**
  if $D_{ij} < 1 - \tau_{strict}$ **then**
    // Strict Tier: Remove near-duplicate regardless of performance
    if $a_i.J \neq a_j.J$ **then**
      Remove $\arg\max_{a \in \{a_i, a_j\}} a.J$ from $\mathcal{A}$ // Remove worse
    else
      Remove random $a \in \{a_i, a_j\}$ from $\mathcal{A}$ // Identical performance
    Remove corresponding row/column from $\mathbf{D}$
  else if $D_{ij} < 1 - \tau_{soft}$ ***and*** $|a_i.J - a_j.J| < \epsilon$ **then**
    // Soft Tier: Remove similar with close performance
    if $a_i.J \neq a_j.J$ **then**
      Remove $\arg\max_{a \in \{a_i, a_j\}} a.J$ from $\mathcal{A}$ // Remove worse
    else
      Remove random $a \in \{a_i, a_j\}$ from $\mathcal{A}$ // Identical performance
    Remove corresponding row/column from $\mathbf{D}$
return $\mathcal{A}, \mathbf{D}$

open-source model, which motivates its use as the default representation in ATLAS.

Practical details: All embeddings are ℓ_2-normalized before similarity computation, so cosine similarity is equivalent to the dot product between normalized vectors.

2) Clustering Details: **Clustering objective and algorithm:** We cluster candidate algorithms using *k-medoids* on a precomputed distance matrix $\mathbf{D}$, solved with FasterPAM [39], an optimized variant of the classical PAM (Partitioning Around Medoids) procedure [21]. K-medoids is used because it operates directly on arbitrary pairwise distances and yields medoids that are *actual* candidate algorithms in the archive, which is useful for downstream representative-based search.

Distance matrix: For the archive $\mathcal{A}$, pairwise distances are computed from cosine similarity in the early-fusion embedding

Algorithm B.4: Module: Archive Size Management

Input: Archive $\mathcal{A}$, distance matrix $\mathbf{D}$, $N_{\max}$ (capacity), $\mathcal{G}^*$ (protected elite region)
Output: Pruned Archive $\mathcal{A}$ and updated $\mathbf{D}$

$\mathcal{E} \leftarrow \emptyset$ `// Set of skipped protected pairs`
`// Iterative Capacity-Constrained Pruning with Elite Protection`
while $|\mathcal{A}| > N_{\max}$ **do**
  $(a_i, a_j) \leftarrow \arg\min_{\substack{x,y \in \mathcal{A} \\ x \neq y,\ (x,y) \notin \mathcal{E}}} D_{xy}$ `// Find closest unexamined pair`
  if $a_i \in \mathcal{G}^*$ ***and*** $a_j \in \mathcal{G}^*$ **then**
    `// Both protected: skip this pair`
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a_i, a_j)\}$
    continue to next closest pair
  else if $a_i \in \mathcal{G}^*$ ***or*** $a_j \in \mathcal{G}^*$ **then**
    `// One protected: remove the unprotected one`
    if $a_i \in \mathcal{G}^*$ **then**
      Remove a_j from $\mathcal{A}$
    else
      Remove a_i from $\mathcal{A}$
    Remove corresponding row/column from $\mathbf{D}$
  else
    `// Neither protected: remove worse performer`
    if $a_i.J \neq a_j.J$ **then**
      Remove $\arg\max_{a \in \{a_i, a_j\}} a.J$ from $\mathcal{A}$
    else
      Remove random $a \in \{a_i, a_j\}$ from $\mathcal{A}$
    Remove corresponding row/column from $\mathbf{D}$
return $\mathcal{A}, \mathbf{D}$

space:

$$D_{ij} = 1 - \mathrm{sim}_{\text{concat}}(a_i, a_j), \qquad a_i, a_j \in \mathcal{A},$$

where $\mathrm{sim}_{\text{concat}}(\cdot,\cdot)$ is defined in Eq. (C.3).

Initialization (PAM BUILD): Each FasterPAM run is initialized with the PAM BUILD procedure [21], which greedily selects an initial set of medoids before refinement.

Multiple restarts: For each candidate K, we perform $n_{\text{init}} = 10$ independent FasterPAM runs with different random seeds and a per-run iteration budget of `max_iter` $= 300$, and retain the solution with the lowest k-medoids objective value. This reduces sensitivity to initialization and helps avoid poor local minima [21].

Automatic K selection (Silhouette): We select K automatically by maximizing the average Silhouette score [37] over a candidate range:

$$K \in \{K_{\min}, \ldots, K_{\max}\}, \qquad K_{\max} = \max\Big(K_{\min}, \Big\lfloor \sqrt{|\mathcal{A}|} \Big\rfloor\Big),$$

where clustering is performed on the full archive $\mathcal{A}$. We use $K_{\min} = 3$ in ATLAS because the cluster containing the current best algorithm a^* is removed after clustering; thus, at least two clusters must remain to support Layer 3 operations. For each candidate K, we cluster $\mathcal{A}$ with FasterPAM and compute the Silhouette score using the precomputed distance matrix. The selected value K^* is the maximizer, with ties broken in favor of smaller K.

Singleton-cluster handling: After clustering, we only post-process *singleton* clusters ($|C| = 1$). Each singleton is merged into the nearest non-singleton cluster, where "nearest" is defined by the minimum average distance from the singleton to members of the target cluster.

Representative selection: For each cluster, we select as representative the best-performing algorithm in that cluster, i.e., the member with the lowest objective value. This representative serves as the cluster's strongest current exemplar for downstream refinement and cross-cluster synthesis.

3) Hyperparameter Configuration: This section documents all hyperparameters used in ATLAS together with their design rationale. Unless otherwise noted, these values are fixed across all experiments.

a) Archive management:

- $\boldsymbol{N_{\max} = 100}$**:** Maximum archive size. ATLAS maintains a pairwise embedding distance matrix and performs clustering over the archive, both of which scale at least quadratically with $|\mathcal{A}|$. A capacity of 100 permits coverage across multiple embedding-space regions while keeping per-iteration overhead manageable. Under the square-root heuristic used for clustering, this yields at most $K_{\max} \approx 10$ candidate clusters.

Algorithm B.5: Module: Three-Layer Candidate Generation

Input: $\mathcal{P}$ (problem), $\mathcal{M}$ (LLM), Layer 1 region $\mathcal{G}^*$, best a^*, semantic clusters $\Omega = \{(C_i, r_i)\}$
Input: Counts: $n_{\text{intensify}}$
Input: Reference sizes: m_{comb}, m_{div}
Input: Probabilities: p_{imp}, p_{tune}, p_{simp}
Output: Candidate Set $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$
// Layer 1: Intensive Refinement of Best Algorithm
for $i = 1$ **to** $n_{intensify}$ **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Improve}(a^*, \mathcal{P}, \mathcal{M})\}$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Tune}(a^*, \mathcal{P}, \mathcal{M})\}$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Simplify}(a^*, \mathcal{P}, \mathcal{M})\}$
// Layer 1: Probabilistic Refinement of Neighbors
foreach $a \in \mathcal{G}^* \setminus \{a^*\}$ **do**
  if *random*() $< p_{imp}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Improve}(a, \mathcal{P}, \mathcal{M})\}$
  if *random*() $< p_{tune}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Tune}(a, \mathcal{P}, \mathcal{M})\}$
  if *random*() $< p_{simp}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Simplify}(a, \mathcal{P}, \mathcal{M})\}$
// Layer 1: Combination within Best Region
for $j = 1$ **to** $|\mathcal{G}^*|$ **do**
  Randomly sample $\mathcal{R}' \subset \mathcal{G}^* \setminus \{a^*\}$ of size $m_{\text{comb}} - 1$
  $\mathcal{R} \leftarrow \{a^*\} \cup \mathcal{R}'$ // Ensure a^* included
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Combine}(\mathcal{R}, \mathcal{P}, \mathcal{M})\}$
// Layer 2: Distributed Regional Refinement
foreach *cluster* $(C_k, r_k) \in \Omega$ **do**
  if *random*() $< p_{imp}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Improve}(r_k, \mathcal{P}, \mathcal{M})\}$
  if *random*() $< p_{tune}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Tune}(r_k, \mathcal{P}, \mathcal{M})\}$
  if *random*() $< p_{simp}$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Simplify}(r_k, \mathcal{P}, \mathcal{M})\}$
  Randomly sample $\mathcal{R}' \subset C_k \setminus \{r_k\}$ of size $m_{\text{comb}} - 1$
  $\mathcal{R} \leftarrow \{r_k\} \cup \mathcal{R}'$ // Ensure r_k included
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Combine}(\mathcal{R}, \mathcal{P}, \mathcal{M})\}$
// Layer 3: Cross-Cluster Operators
for $j = 1$ **to** $|\Omega|$ **do**
  Randomly sample $\mathcal{R} \subset \{\, r_k : (C_k, r_k) \in \Omega \,\}$ of size m_{comb} from different clusters
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Combine}(\mathcal{R}, \mathcal{P}, \mathcal{M})\}$
  Randomly sample $\mathcal{R}' \subset \{\, r_k : (C_k, r_k) \in \Omega \,\}$ of size $m_{\text{comb}} - 1$ from different clusters
  $\mathcal{R} \leftarrow \{a^*\} \cup \mathcal{R}'$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Combine}(\mathcal{R}, \mathcal{P}, \mathcal{M})\}$
  Randomly sample $\mathcal{R}'' \subset \{\, r_k : (C_k, r_k) \in \Omega \,\}$ of size $m_{\text{div}} - 1$
  $\mathcal{R} \leftarrow \{a^*\} \cup \mathcal{R}''$ // Include a^* to stay distinct from the $\mathcal{G}^*$ region
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\texttt{Diverge}(\mathcal{R}, \mathcal{P}, \mathcal{M})\}$
return $\mathcal{C}$

- $\boldsymbol{\tau_{\text{strict}} = 0.95}$**:** Strict similarity threshold for deduplication. This value was chosen based on an exploratory analysis of aggregated archives from 8 pilot runs on FSS and CVRP (800 algorithms per problem), and then fixed for all experiments. Although pairwise cosine-similarity distributions vary across problems, manual inspection showed that pairs with similarity greater than 0.95 predominantly corresponded to clones, functionally equivalent rewrites, trivial edits, or near-identical parameter variants. We therefore use $\tau_{\text{strict}} = 0.95$ to remove near-duplicates regardless of performance difference.
- $\boldsymbol{\tau_{\text{soft}} = 0.90}$**:** Soft similarity threshold for performance-aware deduplication. Pairs above this threshold are highly similar under the selected representation and often differ only in local design choices, while still sometimes exhibiting meaningful performance differences. We therefore treat similarity greater than 0.90 as a candidate near-duplicate region and apply the performance tolerance below before pruning.
- $\boldsymbol{\epsilon = 0.01}$**:** Performance tolerance for soft-tier deduplication. Unlike strict deduplication, which removes extremely similar

pairs regardless of performance difference, soft-tier deduplication applies only to highly similar pairs with nearly identical average training objective values. Specifically, for pairs with similarity greater than τ_{soft}, one algorithm is removed only if the absolute difference in their average training objective values is smaller than ϵ. We set $\epsilon = 0.01$ as a very small tie tolerance so that soft deduplication removes only highly similar algorithms with effectively identical performance. In our benchmark settings, performance differences among competitive algorithms are typically much larger than this threshold, so ϵ functions as a near-equality check rather than a substantive performance margin. This absolute tolerance is appropriate for the objective scales considered here, but may require adjustment for problems with substantially smaller-magnitude objective values.

b) Initialization:

- **Total: 50 initial algorithms (20 Phase 1, 30 Phase 2):** We generate 50 initial candidates, corresponding to 50% of $N_{\max} = 100$, to provide sufficient pre-search coverage for clustering and k-NN retrieval. Before the main loop, deduplication removes near-identical seeds, often from Phase 1, so the retained initial archive size can be smaller than 50. Under the default budget ($B = 500$ operator calls), initialization consumes 10% of synthesis resources, leaving 90% for iterative refinement in the three-layer strategy. This balances initial coverage with efficiency: too small a seed set delays coverage across embedding-space regions, while too large a seed set spends excessive budget on seeding rather than targeted search.
- $\boldsymbol{n_{\text{init}}^{\text{crt}} = 20}$**:** Phase 1 uses CREATE for from-scratch generation. In practice, zero-shot synthesis often concentrates on canonical strategies, such as standard greedy construction and insertion heuristics. We therefore use 20 CREATE calls to obtain a diverse initial seed set without allocating excessive budget to a mode that tends to produce overlapping starting points.
- $\boldsymbol{n_{\text{init}}^{\text{div}} = 30}$**:** Phase 2 is weighted more heavily ($30 > 20$) to explicitly expand beyond the dominant modes produced by CREATE. DIVERGE is applied iteratively to a growing pool: it initially uses Phase 1 seeds as references to avoid, then repeatedly samples from the current pool, including previously diverged algorithms, to drive further structural diversification. This bootstraps multiple clusters in embedding space before the main loop and provides diverse anchors for subsequent clustering and three-layer search.

c) Layer 1 configuration.:

- $\boldsymbol{|\mathcal{G}^*| = 5}$ **($\boldsymbol{a^*}$ + 4 nearest neighbors):** The elite set size balances local refinement around the best-performing region against the budget reserved for Layers 2–3. A small neighborhood provides multiple refinement anchors around a^* for k-NN sampling and local search while keeping Layer 1 overhead bounded. Smaller elite sets weaken local coverage, whereas larger sets increase per-iteration cost because more members undergo refinement.
- $\boldsymbol{n_{\text{intensify}} = 3}$**:** Number of intensification rounds applied to a^*. In Layer 1, neighbors in $\mathcal{G}^* \setminus \{a^*\}$ are refined via single-reference operators sampled according to operator probabilities ($p_{\text{imp}}, p_{\text{tune}}, p_{\text{simp}}$). In contrast, a^* is excluded from probabilistic refinement and receives special treatment: in each intensification round, we always apply IMPROVE, TUNE, and SIMPLIFY to a^*, thereby maintaining strong exploitation pressure. This compensates for strict deduplication: selection-driven evolutionary methods implicitly generate many near-clones of the current best that each undergo refinement, whereas ATLAS removes such clones to preserve diversity and therefore requires explicit intensification.
  Sensitivity: The value $n_{\text{intensify}} = 3$ was selected based solely on training-set performance. The training analysis indicated that $n_{\text{intensify}} = 2$ under-exploited the current-best region, whereas $n_{\text{intensify}} = 4$ provided insufficient additional improvement over $n_{\text{intensify}} = 3$ to justify reducing the budget available to Layers 2–3. For interpretability, Table B.2 reports the corresponding test-set relative mean gaps for the candidate settings. These test results were not used either to select $n_{\text{intensify}}$ or to select the algorithm within any run.

TABLE B.2: *Relative mean gaps for the $n_{\text{intensify}}$ sensitivity analysis.* Percent gap to the strongest human-designed reference in each setting (PyVRP for CVRP and IG-TB for FSS); lower is better. Values are computed from the underlying raw objective means. The setting $n_{\text{intensify}} = 3$ was selected using training-set performance only; the test-set gaps are reported only for interpretability.

	CVRP($n50$) gap (%)↓			**FSS**($n50m10$) gap (%)↓		
$\boldsymbol{n_{\text{intensify}}}$	2	3	4	2	3	4
Relative mean gap (%)	0.572	**0.094**	**0.095**	0.401	**0.347**	**0.351**

d) Clustering: ATLAS clusters the archive using k-medoids (FasterPAM) on the precomputed embedding distance matrix, with K selected automatically by maximizing the average Silhouette score over $[K_{\min}, K_{\max}]$. Each cluster representative is selected as its best-performing member, i.e., the algorithm with the lowest objective value. Complete clustering details, including initialization, restarts, and singleton handling, are provided in Appendix B-E2.

- $\boldsymbol{K_{\min} = 3}$ **(structural minimum):** Minimum cluster count required to sustain the three-layer architecture. Since the cluster containing a^* is excluded from Layer 2 and Layer 3 selection, at least $K_{\min} - 1 = 2$ non-elite clusters must remain available for Layer 2 maturation and Layer 3 cross-cluster search. This ensures that Layer 3 can draw representatives from multiple clusters for COMBINE and provides alternative represented regions for DIVERGE across iterations.

- $K_{\max} = \max(3, \lfloor\sqrt{|\mathcal{A}|}\rfloor)$ **(granularity bound):** Upper bound for automatic K selection, following the square-root heuristic to balance granularity with stability. For $|\mathcal{A}| = 100$, this yields $K_{\max} = 10$. This bound discourages Silhouette optimization from fragmenting the embedding space into very small clusters and keeps representative-based operations supplied with nontrivial groups. The $\max(3, \cdot)$ term ensures $K_{\max} \geq K_{\min}$ when the archive is small.

e) Multi-reference operator reference sizes: These operator-level settings are used consistently wherever COMBINE and DIVERGE are applied.

- $m_{\text{comb}} = 2$ **(recombination depth):** Number of reference algorithms for COMBINE. We set $m_{\text{comb}} = 2$ because each ATLAS COMBINE prompt includes complete source programs, so reference count directly trades integration depth against context length. This choice is motivated by three considerations: (i) **Code complexity:** ATLAS combines complete executable programs, often already containing hybrid logic from earlier iterations, so additional references increase the amount of source that must be integrated. (ii) **Integration depth:** with two references, the LLM more reliably performs substantive integration of both algorithms; with $m_{\text{comb}} \geq 3$, preliminary runs often produced outputs dominated by one reference with only shallow incorporation of the others. (iii) **Token efficiency:** two full algorithms provide sufficient recombination context while keeping prompt length manageable.
- $m_{\text{div}} = 3$ **(negative signal sufficiency):** Number of reference algorithms for DIVERGE. Unlike COMBINE, DIVERGE uses references primarily as strategies to avoid rather than as inputs for deep integration, so it benefits from slightly broader context. We therefore use $m_{\text{div}} = 3$: three references provide a broader negative context and are intended to encourage outputs separated from the referenced archive regions while keeping token cost manageable. Larger values provided little additional benefit in preliminary runs while increasing prompt length.

f) Operator probabilities (Layers 1 & 2): Single-reference refinement operators are applied stochastically to Layer 1 neighbors (excluding a^*, which receives deterministic intensification) and Layer 2 cluster representatives. These probabilities control both the refinement mix and budget allocation: each operator invocation produces a candidate that must be validated and evaluated, so increasing these probabilities increases refinement pressure but consumes more of the fixed synthesis budget B, thereby reducing the number of outer-loop iterations and the budget available for multi-reference search and cross-cluster exploration.

- $p_{\text{imp}} = 0.5,\ p_{\text{tune}} = 0.5$ **(balanced refinement mix):** Probabilities for IMPROVE (structural logic refinement) and TUNE (parameter-level adjustment). We set them equal to allocate comparable pressure to structural refinement and parameter adjustment, which play complementary roles in full-algorithm synthesis. Setting both to 0.5 provides strong refinement pressure in Layers 1 and 2 while preserving sufficient budget for outer-loop progress and multi-reference exploration. Substantially larger values over-allocate budget to local refinement, whereas substantially smaller values weaken refinement coverage.
- $p_{\text{simp}} = 0.2$ **(parsimony under budget constraints):** Probability for SIMPLIFY (code reduction). SIMPLIFY acts as a regularizer against code bloat during iterative synthesis, but it is not the primary performance-improvement operator. We therefore apply it less frequently than IMPROVE and TUNE: a value of 0.2 provides periodic parsimony pressure without diverting too much budget away from performance-driven refinement and broader exploration. Higher values can over-emphasize simplification and remove useful logic prematurely, whereas much lower values weaken bloat control.

4) LLM Configuration:

- **Model: GPT-5-MINI (reasoning effort: LOW):** We use GPT-5-MINI with low reasoning effort because it is capable enough for full-algorithm synthesis while remaining practical in cost and latency for iterative search over hundreds of operator calls.
- $T = 1.0$ **(temperature):** Sampling temperature used for all synthesis operators. We use $T = 1.0$ to balance generation diversity and stability during search. The same value is fixed across all compared LLM-based synthesis methods in our experiments.

5) Embedding configuration: Both the embedding model and input representation were selected based on the representation analysis in Appendix C-A.

- **Model: MGTE-LARGE-EN-V1.5 [58]:** See Appendix B-E1 for details.
- **Input: early fusion (name + description + preprocessed code):** Preprocessing details are provided in Appendix B-E1.

6) Search Budget:

- $B = 500$ **evaluated operator executions:** Default ATLAS search budget used for end-to-end component ablations and search-configuration analyses. This budget is sufficient to demonstrate effective search behavior while keeping overall runtime and API cost tractable.
- **Token-matched budgets for cross-method comparisons:** For comparisons against baselines, we match total token consumption per synthesis run rather than the number of evaluated operator executions, because token usage differs substantially between component synthesis and full-algorithm synthesis. These budgets are calibrated from the average token usage of 500 evaluated operator executions under full synthesis, yielding problem-specific budgets of 5M tokens for FSS, 6.5M for CVRP, 7M for CVRPTW, and 6M for QAP.

a) Evaluation:

- **Per-instance runtime caps (problem dependent):** Maximum execution time allowed for evaluating an algorithm on a single instance. These are upper bounds rather than typical runtimes: many synthesized and baseline algorithms terminate well before the cap. Full-algorithm synthesis explores algorithms with substantially different computational profiles, ranging from fast constructive methods to iterative metaheuristics, making benchmark-setting-specific runtime caps necessary to accommodate this diversity while maintaining fair comparison. The caps serve two purposes: (i) **Safety:** they terminate pathological implementations (e.g., infinite loops or excessively slow code) and keep the overall experiments tractable; and (ii) **comparison consistency:** they ensure that all methods within a benchmark setting are evaluated under the same computational budget.
  The runtime caps are benchmark-setting-specific and correspond to the default problem settings used in this work (Appendix B-B): 210 s for FSS, 30 s for CVRP, 120 s for CVRPTW, and 240 s for QAP. These values are applied uniformly to all compared methods within each benchmark setting. These full caps are applied unchanged during training and test execution.
- $N_{\mathbf{train}} = \mathbf{31},\ N_{\mathbf{test}} = \mathbf{31}$ **(split sizes):** Search uses only the 31 training instances, and the independently generated 31-instance test set provides the primary estimate. Split seeds are 2024 and 42, respectively.
- $m_{\text{limit}} = \mathbf{1}$ **GB additional worker address space:** The released evaluator requires Linux/WSL2 and refuses native Windows execution. In each worker, `RLIMIT_AS` is set to the inherited virtual-memory footprint plus 1 GB, thereby limiting approximately 1 GB of additional virtual-address-space allocation while one instance is executed. This is a resource-containment allowance, not a claim that resident memory is measured exactly; a violation disqualifies the candidate execution.

F. ATLAS: Candidate Validation, Repair, and Execution Pipeline

Each candidate algorithm generated by the LLM is evaluated through a structured pipeline that assesses executability, constraint feasibility, and solution quality.

1) Sequential LLM Calls, Parallel Instance Evaluation: LLM synthesis calls are executed sequentially, while evaluation across problem instances is parallelized. Sequential LLM calls simplify rate-limit handling and long-run orchestration of the iterative search loop. Instance evaluations use a CPU-aware process pool whose worker count is the smallest of the instance count, available CPU cores, and the user-specified limit.

2) Entry-Point Checking and Worker-Process Execution: Each generated program is first checked for the required problem-specific entry-point function; programs failing this compatibility check are marked invalid without execution. Programs that pass are executed with normal Python built-ins and import access in separate worker processes. The process boundary, timeouts, and memory limits provide fault and resource containment, but they are not a security sandbox and do not block filesystem or network access. Generated code must therefore be treated as untrusted and evaluated in a disposable, least-privilege virtual machine or container without sensitive files, network access, or inherited credentials.

3) Resource Limits and Error Handling: A per-instance timeout t_{timeout} is enforced during execution. Forward experiments run under Linux/WSL2 (native Windows execution is refused), where each evaluation worker receives a configured allowance for additional virtual-address-space allocation above its inherited process footprint through `RLIMIT_AS`. These controls bound runaway execution and excessive allocation; the memory setting is a worker resource allowance rather than a measurement of an algorithm's peak resident memory. Timeouts, memory-limit violations, runtime exceptions (e.g., `IndexError`, `TypeError`), or invalid outputs mark that instance as failed. For candidates eligible for repair, the operator receives failure counts, evaluator error summaries, and available traceback or resource flags; it does not receive the failing instance data itself.

4) Independent Constraint and Objective Validation: Objective values and constraint satisfaction are evaluated by the framework's authoritative validator. The generated algorithm returns only the raw solution representation (e.g., a route set or permutation); the evaluator then independently verifies:

- **Structural validity:** correct output format and dimensions
- **Resource constraints:** capacity limits respected (e.g., CVRP)
- **Temporal constraints:** time windows satisfied (e.g., CVRPTW)
- **Problem-specific validity:** sequencing/order and feasibility conditions required by the target problem (e.g., FSS, QAP)

5) All-or-Nothing Validity and Repair: A candidate is considered *valid* only if it succeeds on *all* training instances without timeouts, exceptions, format violations, or constraint violations. If any instance fails, the candidate is tagged as failed and passed once to the REPAIR operator (§III-D) with diagnostic feedback. The repaired candidate is re-evaluated; if it still fails, it is discarded. Only valid algorithms are added to the archive, and their fitness is computed as the mean objective value across instances.

6) Failure-Conditioned Repair: REPAIR is a source-level recovery operator for a generated full algorithm that reaches training evaluation but fails during execution or evaluator validation. Candidates rejected before evaluation, such as malformed structured LLM responses or source that does not pass the entry-point check, do not enter this repair path.

a) Failure evidence and routing: For each evaluated candidate, ATLAS records the aggregate error message, an available traceback, timeout and memory flags, and the numbers of successful and failed instances. Deterministic rules classify this evidence without an additional LLM diagnosis call. The classes are checked in the priority order shown in Table B.3; the first matching class determines the correction guidance.

TABLE B.3: Failure classes and correction objectives used by REPAIR.

Order	Class	Identifying evidence	Correction objective
1	Memory limit	Memory-limit flag or `MemoryError`	Reduce large materialized structures, unbounded containers, redundant copies, or deep recursion
2	Partial failure	Some training instances succeed and others fail	Preserve the core approach while handling edge cases and removing instance-dependent failures
3	Timeout	Timeout marker in the aggregate error	Reduce computational complexity, redundant work, and excessive iteration within the existing approach
4	Constraint violation	A feasibility keyword in the evaluator message (Table B.4)	Correct construction or modification logic so returned solutions satisfy the problem constraints
5	Runtime error	Exception, error, or traceback markers after excluding earlier classes	Locate and correct the source-level crash and add relevant defensive checks
6	Framework violation	Remaining unmatched validation or interface failure	Correct the entry point, signature, return type, output structure, or other interface requirement

Constraint violations are separated from runtime errors because the candidate executes successfully but returns an invalid or infeasible solution. This branch uses case-insensitive keyword matching over the aggregate evaluator message. Table B.4 reproduces the keyword groups used by the implementation.

TABLE B.4: Keyword groups used to identify constraint-violation failures. Matches are case-insensitive.

Category	Keywords
Capacity and resources	exceeds capacity; capacity constraint; over capacity
Completeness	not all customers visited; missing jobs; missing items; missing customers
Uniqueness	visited multiple times; duplicate; appears multiple times
Permutation and sequence	not a valid permutation; job sequence length; invalid permutation; sequence length
General validity	constraint violation; invalid solution; feasibility; infeasible; out of range; invalid index; invalid customer; invalid item

b) Source regeneration and acceptance: The selected class conditions the repair request on the observed failure rather than issuing a generic debugging instruction. The LLM receives the problem specification and I/O requirements, the failed algorithm's name, description, and complete source, the aggregate diagnostic evidence, and class-specific correction objectives. It is asked to regenerate the complete algorithm source while retaining useful algorithmic ideas where appropriate. The exact executable prompt and guidance strings are provided in the public source-code repository.[5]

The regenerated source passes the same code and interface checks and is then re-evaluated from scratch on the complete training set. It is admitted to the archive only if every training instance succeeds; otherwise, the repair is recorded as failed and the candidate is discarded under the reported default one-attempt policy. This all-or-nothing re-evaluation prevents REPAIR from weakening feasibility or execution requirements while recovering promising designs that would otherwise be lost.

G. Baselines

We compare ATLAS against human-designed, domain-specific optimization methods and LLM-based synthesis baselines. This section summarizes their implementations and configurations.

1) Human-Designed Domain-Specific Baseline Implementations: The configurations below identify both the published method and the exact implementation used. We use documented defaults or fixed upstream example configurations and do not tune a method using test results. Runtime caps are common upper bounds rather than typical runtimes: many LLM-synthesized algorithms terminate well before the cap, whereas runtime-sensitive domain-specific baselines can continue improving until the allotted time is exhausted. Thus, the caps do not disadvantage these baselines and may be more beneficial to methods designed to exploit the full runtime budget. Unless otherwise noted, the runtime-sensitive baselines use 30 s for CVRP, 120 s for CVRPTW, 210 s for FSS, and 240 s for QAP in the default settings (Appendix B-B).

[5]https://github.com/Danial-Yazdani/ATLAS

a) CVRP:

- **PyVRP [52, 53]:** PyVRP v0.13.3 with its default iterated local search configuration, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **Google OR-Tools [14]:** OR-Tools v9.15.6755 using `PATH_CHEAPEST_ARC` as the first-solution strategy and `GUIDED_LOCAL_SEARCH` as the local-search metaheuristic, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **VROOM [6, 50]:** VROOM v1.16.0-dev, commit `07be776`, built with `USE_ROUTING=false` and supplied an explicit cost matrix so no external routing service is involved. It runs single-threaded (`-t 1`), at exploration level 5 (`-x 5`), and with the benchmark-setting-specific limit supplied through `-l`.

b) CVRPTW:

- **PyVRP [52, 53]:** PyVRP v0.13.3 with its default iterated local search configuration for VRPTW, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **Google OR-Tools [14]:** OR-Tools v9.15.6755 using `PATH_CHEAPEST_ARC` as the first-solution strategy and `GUIDED_LOCAL_SEARCH` as the local-search metaheuristic, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **VROOM [6, 50]:** the same VROOM build and execution configuration used for CVRP, with capacities, service times, and time windows supplied through the problem input and every returned route independently checked by the common evaluator.

c) FSS:

- **NEH [32]:** NEH heuristic implementation following the original method, run to completion because it is a deterministic constructive heuristic with polynomial-time complexity and terminates quickly on the default FSS benchmark setting.
- **IG-TB with Taillard acceleration [38, 45, 12]:** the Ruiz–Stützle IG configuration implemented by `js-aguiar` [20], with the Fernández-Viagas–Framinan idle-time insertion tie-breaking rule and Taillard's $O(km)$ insertion evaluation, using Python with Cython kernels.
- **Iterative Beam Search [24]:** anytime constructive tree search using the authors' CATS-PFSP implementation (commit `41edc0e`), with the bidirectional variant and `g4` guidance (`variant=1`, `guide=3`) fixed from the upstream example.

d) QAP:

- **Robust Tabu Search (RoTS) [46]:** RoTS implementation following the original method, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **Simulated Annealing [5]:** SA implementation following the original method, evaluated under the benchmark-setting-specific per-instance runtime cap.
- **Breakout Local Search (BLS) [1]:** iterated local search alternating steepest-descent improvement with adaptive perturbations, using Benlic and Hao's released source as integrated in the public ISA-QAP-Algorithms collection [19]. The integration accepts an external RNG seed and runtime cap and exposes the returned permutation for independent scoring.
- **Memetic Search (BMA) [2]:** population-based search integrating BLS, crossover, pool updating, and adaptive mutation, using the implementation distributed in the same ISA-QAP-Algorithms collection [19]. The integration supplies the external seed and cap and retains a self-verified best returned permutation so the common evaluator can recompute its objective.

2) Component-Synthesis Baseline Implementations: We compare against component-oriented implementations of three LLM-based methods: ReEvo [57], EoH [25], and MCTS-AHD [61]. Their official repositories target different problems and use non-uniform scaffolds, prompts, and evaluators, making a controlled end-to-end comparison under their native setups difficult to interpret.[6] We therefore evaluate all three methods using the LLM4AD platform [26],[7] which provides a shared scaffold, prompts, and evaluators.

a) Shared benchmark interface: All three component-synthesis baselines are evaluated under the same LLM4AD-based component-execution and benchmarking infrastructure:

- **Common scaffold:** all methods synthesize heuristic components within the same LLM4AD greedy-construction scaffold.
- **Common prompts and interface specification:** all methods use the same LLM4AD problem prompts and interface specification under this scaffold, so differences are not attributable to prompt wording or scaffold design.
- **Common evaluators:** all methods use the same LLM4AD-based execution and evaluation pipeline, including timeout handling, feasibility handling, objective computation, and reporting.
- **Common benchmark instances:** all methods use the same 31 training instances and 31 test instances for each default benchmark setting, with fixed seeds. For CVRP, CVRPTW, and QAP, we use the LLM4AD benchmark generators, which are also used by ATLAS. For FSS, LLM4AD's default Co-Bench setup is replaced with the standard generator used throughout this paper (Appendix B-B3) so that all methods are evaluated on the same benchmark setting.

[6] ReEvo: https://github.com/ai4co/reevo, EoH: https://github.com/FeiLiu36/EoH, MCTS-AHD: https://github.com/zz1358m/MCTS-AHD-master
[7] LLM4AD (v1.0.0): https://github.com/Optima-CityU/llm4ad

b) Method-specific preservation and controlled adaptations: Within this shared interface, we retain the default search mechanisms and parameter settings of the official implementations as closely as possible (e.g., selection and population-management logic), modifying only evaluation-facing components needed for fair comparison:

- **Termination criterion:** all methods use the same benchmark-specific token budgets as ATLAS, namely 5M tokens for FSS, 6.5M for CVRP, 7M for CVRPTW, and 6M for QAP.
- **Per-instance runtime caps:** all methods use the same benchmark-setting-specific per-instance runtime caps as the rest of the paper, namely 210 s for FSS, 30 s for CVRP, 120 s for CVRPTW, and 240 s for QAP.
- **LLM backend:** all methods use GPT-5-MINI with temperature $T = 1.0$ and reasoning effort set to *low*.

This standardization reduces confounding from scaffold design, prompts, evaluators, benchmark instances, and stopping criteria when comparing the three search procedures in their tested component-oriented implementations.

3) EOH-FULL*: Full-Algorithm Synthesis Baseline:* To isolate the contribution of ATLAS's search organization, we construct EOH-FULL, a controlled full-synthesis baseline derived from EOH [25]. EOH-FULL retains EOH's fitness-driven evolutionary search framework while using ATLAS's problem- and interface-aware full-algorithm setting. By matching synthesis operators, prompts, repair, evaluator, benchmark instances, LLM configuration, runtime caps, and token budget, this comparison is designed to reduce confounding from operator expressiveness and evaluation conditions when assessing the archive-based semantic quality-diversity search.

a) Formulation: EOH-FULL adapts EOH from component synthesis to full-algorithm synthesis. It preserves the evolutionary search structure of the original method, including its fitness-driven selection, population-update logic, population size, initialization scheme, and operator probabilities, but replaces component-level generation with the same full-synthesis operators used in ATLAS: CREATE, IMPROVE, TUNE, SIMPLIFY, COMBINE, DIVERGE, and REPAIR. In particular, EOH-FULL keeps the original EOH population size of 20 rather than matching ATLAS's archive cap $N_{\max} = 100$, since the latter is a bounded semantic archive rather than an active evolutionary population. It also uses EOH's original initialization scheme rather than ATLAS's two-phase initialization.

b) Key differences from ATLAS:

- **Reference selection strategy:** EOH-FULL selects parent/reference algorithms through fitness-driven evolutionary selection (rank-based proportional selection), whereas ATLAS selects references through semantic organization and three-layer search, using embedding neighborhoods, cluster representatives, and cross-cluster combinations to control where search effort is allocated.
- **Population vs. semantic archive:** EOH-FULL maintains a fixed-size active evolutionary population, with offspring inserted and survival determined through fitness-driven selection each generation. This setup is primarily geared toward improving the currently strongest-performing lineage. In contrast, ATLAS maintains an embedding-organized archive with coverage-preserving management, enabling it to retain and continue refining competitive algorithms in multiple embedding-space regions rather than relying only on global fitness rank.
- **Diversity dynamics and search outcome:** EOH-FULL uses fitness-driven survival, which may concentrate sampling around high-performing lineages and is primarily oriented toward identifying a single best final algorithm. ATLAS instead applies embedding-distance-aware archive management and continues refining multiple represented regions throughout search. As a result, ATLAS can retain multiple competitive algorithms from distinct regions of the archive and return the corresponding cluster representatives as a diverse set of candidate algorithms, in addition to the overall best algorithm.

c) Shared components for controlled comparison: To ensure a controlled comparison, EOH-FULL shares the following components with ATLAS:

- **Full-synthesis operator tasks:** EOH-FULL uses the same full-synthesis operator responsibilities as ATLAS, namely CREATE, IMPROVE, TUNE, SIMPLIFY, COMBINE, DIVERGE, and REPAIR, including the same multi-reference settings ($m_{\text{comb}} = 2$ for COMBINE and $m_{\text{div}} = 3$ for DIVERGE).
- **Repair and evaluation:** the same candidate repair mechanism and execution/evaluation pipeline (Appendix B-F), including the same benchmark-setting-specific per-instance runtime caps
- **LLM configuration:** GPT-5-MINI with temperature $T = 1.0$ and reasoning effort set to *low*
- **Benchmark instances:** the same 31 training instances and 31 test instances with identical random seeds
- **Termination criterion:** the same benchmark-specific token budgets used for cross-method comparison

d) EoH-specific preserved mechanisms: Within this shared full-synthesis setting, EOH-FULL keeps the method-specific evolutionary mechanisms of the original EOH implementation, including its selection procedure and operator probability schedule. The original EOH operator groups map naturally onto the ATLAS full-synthesis operator set: modification operators $\mathbf{M_1}, \mathbf{M_2}, \mathbf{M_3}$ correspond to IMPROVE, TUNE, and SIMPLIFY, while exploration operators $\mathbf{E_1}, \mathbf{E_2}$ correspond to DIVERGE and COMBINE, respectively. Thus, the operator responsibilities are the same as in ATLAS. REPAIR has no direct counterpart in the original EOH schedule; however, similar to ATLAS, it is applied only when a generated candidate fails validation. This shared use of REPAIR addresses the additional execution, interface, and feasibility failure modes exposed by complete-program generation and avoids giving ATLAS a recovery mechanism unavailable to EOH-FULL. Under the shared full-synthesis machinery and

evaluation conditions described above, the comparison is designed to estimate the effect of replacing EoH's fitness-driven evolutionary search with ATLAS's semantic quality-diversity search.

Appendix C
Component Ablations and Design Analyses

We distinguish component ablations, which disable or replace ATLAS search mechanisms and evaluate final algorithm quality, from supporting design analyses, which study representation quality, initialization behavior, or the sensitivity of ATLAS to its LLM backend. The semantic search-guidance and search-layer studies are component ablations; the embedding, initialization, reasoning-effort, and underlying-LLM studies are design analyses.

A. Embedding Representation Analysis Against Curated Family Labels

1) Aim and Research Questions:

a) Aim: ATLAS uses pretrained embeddings as the operational representation for k-nearest-neighbor retrieval, clustering, deduplication, and similarity-based pruning. Unlike the fitness- and tree-statistic baselines in this analysis, this representation directly determines archive neighborhoods and coverage management. This study evaluates which embedding strategies are most effective for organizing algorithms in a way that supports synthesis search in combinatorial optimization.

b) Research questions: We study four design questions:

RQ1: Author-labeled family discrimination: To what extent do pretrained embeddings distinguish the broad algorithm-family labels assigned in the curated benchmark (e.g., greedy, local search, and metaheuristic-style methods)?

RQ2: Modality contribution: Which information source contributes most to embedding quality: textual descriptions, source code, or both?

RQ3: Fusion strategy: If multiple modalities are beneficial, should they be combined through early fusion (concatenation) or late fusion (weighted similarity)?

RQ4: Model selection: How do encoder characteristics, such as size, specialization, and availability, affect performance?

These analyses provide empirical justification for the embedding design used in ATLAS.

2) Dataset Construction and Family Annotation: To evaluate embedding quality under controlled conditions, we constructed a curated benchmark of executable algorithm implementations. The dataset comprises 308 algorithms across four combinatorial optimization problems: Flow Shop Scheduling (FSS), Capacitated Vehicle Routing Problem (CVRP), Capacitated Vehicle Routing Problem with Time Windows (CVRPTW), and Quadratic Assignment Problem (QAP). This selection spans routing (CVRP, CVRPTW), scheduling (FSS), and assignment (QAP), providing diverse algorithmic structure across the problem set.

a) Algorithm Generation: For each problem, we independently generated a large candidate pool of algorithm implementations using multiple LLM backends and varied prompting strategies with high-temperature sampling to induce implementation diversity. Each candidate was then validated for syntactic correctness and functional execution on problem-specific benchmark instances, yielding more than 2,000 validated algorithms across the four problems. From this validated pool, we selected a balanced subset across problems and broad author-assigned mechanism labels, while preserving implementation diversity within each label and excluding near-duplicate variants.

b) Why curated evaluation is needed: Embedding evaluation in this setting requires two properties: (i) *inter-label separation*: algorithms should cover different author-assigned broad families, and (ii) *intra-label diversity*: algorithms assigned to the same family should differ in implementation choices rather than only superficial code style. A curated construction process is therefore useful because naively collecting implementations from public sources would make it difficult to control either the breadth of the author-assigned labels or the degree of meaningful implementation variation within each label. Our curation procedure was designed to provide broad label coverage and non-trivial implementation diversity for embedding evaluation. Family labels were assigned internally by the authors from the code, generated description, and recognized search logic.

3) Candidate Models and Selection Rationale: We evaluate 10 pretrained embedding models chosen to cover a diverse range of training focuses, model scales, access modes, and context capacities. The candidate set includes both general-purpose embedding models and code-specialized encoders, allowing us to examine how these characteristics affect algorithm representation quality in our setting. Table C.1 summarizes the model specifications.

a) Selection Rationale: The selected models span a broad range of characteristics that may affect embedding behavior, including training focus, scale, access mode, and context capacity. This diversity allows us to compare general-purpose and code-specialized encoders, assess whether larger models consistently yield stronger representations, and examine the effect of limited versus long input contexts when representing full synthesized algorithms.

Including both proprietary and open-source models also lets us benchmark against strong commercial embeddings while identifying configurations that remain reproducible with publicly available encoders. In particular, the variation in context capacity is important because some source-code inputs exceed the limits of shorter-context models, making this a relevant factor in the design space.

TABLE C.1: *Embedding Model Specifications.* Models are grouped into general-purpose encoders and code-specialized encoders.

Model	Access	Params	Dim.	Context
General-Purpose Models				
TEXT-EMBEDDING-3-LARGE (OPENAI) [34]	Proprietary	–	3072	8K
TEXT-EMBEDDING-3-SMALL (OPENAI) [34]	Proprietary	–	1536	8K
QWEN3-EMBEDDING-0.6B [60]	Open	0.6B	1024	32K
QWEN3-EMBEDDING-4B [60]	Open	4.0B	2560	32K
MGTE-LARGE-EN-V1.5 [58]	Open	434M	1024	8K
Code-Specialized Models				
NOMIC-EMBED-CODE [44]	Open	7.0B	768	32K
CODERANKEMBED [44]	Open	137M	768	8K
CODEBERT-BASE [11]	Open	125M	768	512
GRAPHCODEBERT-BASE [17]	Open	125M	768	512
CODET5-SMALL [51]	Open	60M	512	512

4) Code Preprocessing: All source code is preprocessed before embedding to reduce implementation noise while retaining the core program structure. The preprocessing pipeline applies the following transformations:

1) **Docstring Removal:** Triple-quoted strings (`"""..."""` and `'''...'''`) are removed, as they contain natural language documentation that often overlaps with the algorithm description.
2) **Comment Removal:** Single-line comments (starting with #) are removed while preserving # characters within string literals.
3) **Empty Line Removal:** Blank lines are removed as formatting artifacts without semantic content.
4) **Whitespace Normalization:** Multiple consecutive spaces are converted to single spaces while preserving indentation structure.
5) **Type Hint Preservation:** Type annotations are retained because they encode structural interface information rather than natural-language description, complementing rather than duplicating the algorithm description field.

This pipeline achieves an average 23% character reduction across the corpus. For this representation analysis, preprocessing helps reduce the influence of natural-language artifacts in code inputs, making the comparison between text-based and code-based representations cleaner. The same preprocessing is applied throughout both the embedding representation analysis and the main ATLAS system.

5) Embedding Strategies: We evaluate four embedding strategies that differ in which algorithm components are represented and how the resulting views are combined. Each algorithm is represented as $a = (n, d, c)$, comprising a name n, description d, and source code c. All code-based strategies use preprocessed source code as described in Appendix C-A4.

a) Text-Only: This baseline embeds only the textual components, namely the algorithm name and description:

$$\text{sim}_{\text{text}}(a_i, a_j) = \cos(\mathbf{e}(n_i \oplus d_i), \mathbf{e}(n_j \oplus d_j)), \tag{C.1}$$

where $\mathbf{e}(\cdot)$ denotes the embedding model applied to the corresponding input under the corresponding strategy, and $\oplus$ denotes concatenation. This strategy tests whether textual descriptions alone provide enough information to distinguish the curated broad-family labels.

b) Code-Only: This strategy embeds only the source code implementation:

$$\text{sim}_{\text{code}}(a_i, a_j) = \cos(\mathbf{e}(c_i), \mathbf{e}(c_j)). \tag{C.2}$$

This strategy tests whether preprocessed source code alone provides enough information to distinguish the curated broad-family labels.

Concatenation (Early Fusion): Early fusion concatenates all available components into a single input:

$$\text{sim}_{\text{concat}}(a_i, a_j) = \cos(\mathbf{e}(n_i \oplus d_i \oplus c_i), \mathbf{e}(n_j \oplus d_j \oplus c_j)). \tag{C.3}$$

This strategy tests whether a single embedding model can jointly represent textual intent and implementation structure when both are presented in one sequence.

c) Dual-View (Late Fusion): Late fusion computes separate similarities for text and code, then combines them linearly:

$$\begin{aligned}\text{sim}_{\text{dual}}(a_i, a_j; \alpha) = \alpha \cdot \cos(\mathbf{e}(n_i \oplus d_i), \mathbf{e}(n_j \oplus d_j)) \\ + (1 - \alpha) \cdot \cos(\mathbf{e}(c_i), \mathbf{e}(c_j)),\end{aligned} \tag{C.4}$$

where $\alpha \in [0, 1]$ controls the relative contribution of the text and code views. We sweep $\alpha \in \{0.0, 0.1, 0.2, \ldots, 1.0\}$ to examine the sensitivity of late fusion to the weighting between the two views. This strategy tests whether separating text and code and combining them at the similarity level is more effective than representing them through a single unified input.

6) Evaluation Metrics: We employ two complementary metrics to assess agreement with the curated author-assigned labels: Nearest Neighbor Accuracy (NNA) and Class Cohesion Index (CCI), a silhouette-based score. Both metrics are computed for each embedding strategy defined in Appendix C-A5.

a) Nearest Neighbor Accuracy (NNA): For each algorithm a_i, we identify its nearest neighbor in the embedding space:

$$a_i^{\text{NN}} = \arg\max_{j \neq i} \text{sim}(a_i, a_j), \tag{C.5}$$

where $\text{sim}(\cdot, \cdot)$ denotes the similarity function defined by the strategy (e.g., sim_{text}, sim_{dual}), and a_i^{NN} is the nearest algorithm to a_i. NNA measures the fraction whose nearest neighbor shares the same author-assigned family label:

$$\text{NNA} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\left[y(a_i) = y(a_i^{\text{NN}})\right], \tag{C.6}$$

where N is the total number of algorithms, $y(a_i)$ denotes the author-assigned family label of algorithm a_i, and $\mathbb{1}[\cdot]$ is the indicator function. This metric measures local agreement between the embedding and those labels.

b) Class Cohesion Index (CCI): To complement nearest-neighbor agreement, we measure continuous separation using a silhouette-based cohesion score computed from the author-assigned labels. CCI follows the standard Silhouette Coefficient formulation [37], but uses the curated labels rather than unsupervised clusters. For each algorithm a_i assigned to family C_k, we compute:

$$\text{cohesion}_i = \frac{d^i_{\text{between}} - d^i_{\text{within}}}{\max(d^i_{\text{within}}, d^i_{\text{between}})}, \tag{C.7}$$

where:

- $d^i_{\text{within}} = \frac{1}{|C_k|-1} \sum_{j \in C_k, j \neq i} d(a_i, a_j)$ is the mean distance to other algorithms in the same family (within-family distance).
- $d^i_{\text{between}} = \min_{C_\ell \neq C_k} \frac{1}{|C_\ell|} \sum_{j \in C_\ell} d(a_i, a_j)$ is the mean distance to the nearest different family (between-family distance).

Here, $d(a_i, a_j) = 1 - \text{sim}(a_i, a_j)$ converts similarity to distance, and C_k denotes the author-assigned family containing a_i. The overall Class Cohesion Index is the mean across all N algorithms:

$$\text{CCI} = \frac{1}{N} \sum_{i=1}^{N} \text{cohesion}_i. \tag{C.8}$$

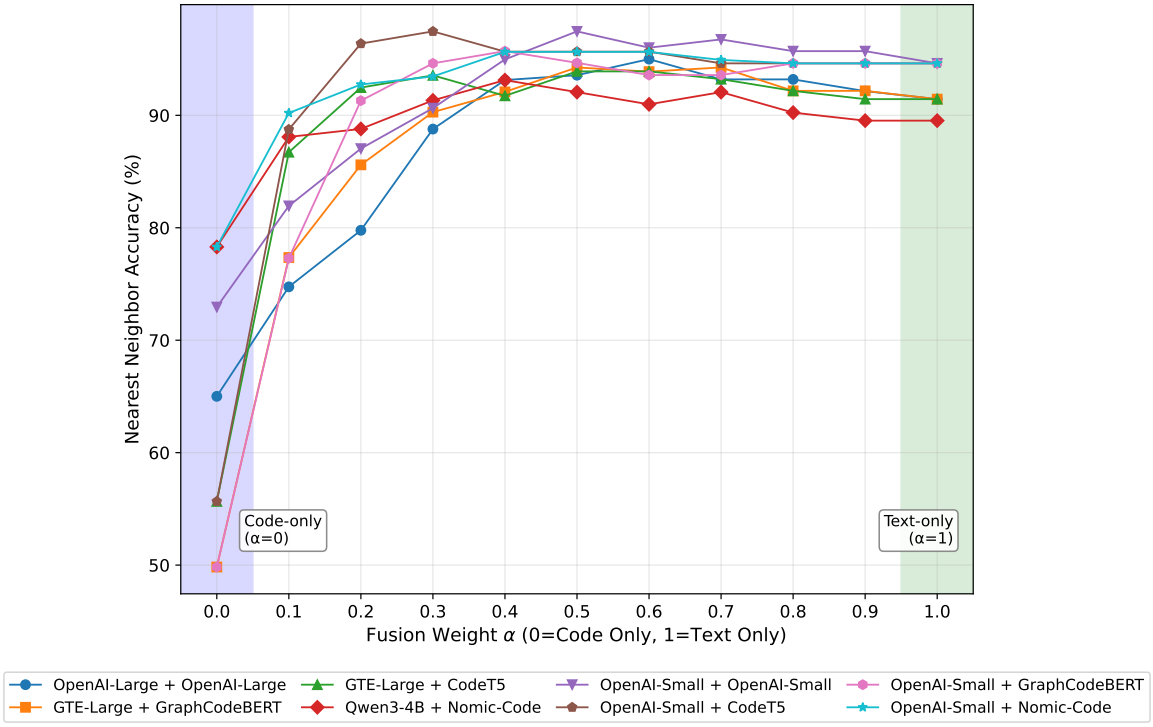
Values range from -1 to $+1$, with larger values indicating tighter grouping and clearer separation with respect to the curated labels.

7) Results: We evaluated all embedding strategies across the candidate models. Table C.2 summarizes the performance of the text-only, code-only, and concatenation strategies. Figure C.1 shows late-fusion performance across fusion weights $\alpha \in [0.0, 1.0]$ for eight representative model pairs.

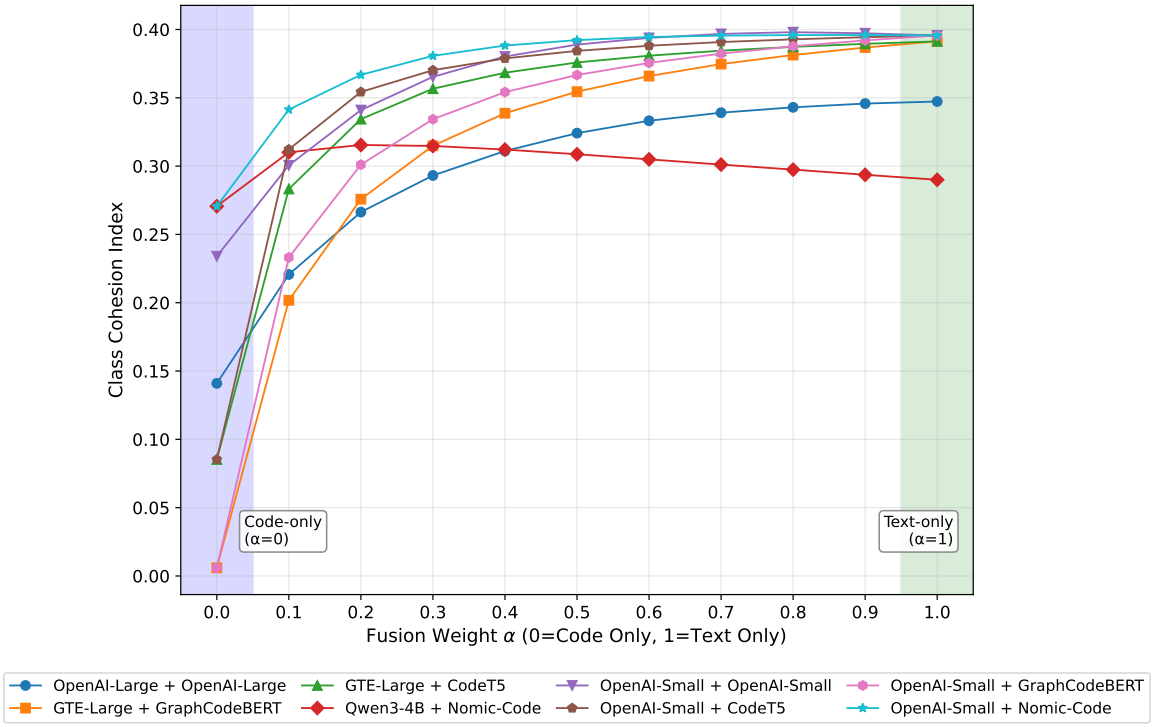
TABLE C.2: *Single-Encoder Strategy Results.* Nearest Neighbor Accuracy (NNA) and Class Cohesion Index (CCI) for text-only, code-only, and concatenation strategies across all models. General-purpose models are evaluated on all three strategies, while code-specialized models are evaluated only in the code-only setting. **Bold** indicates the best NNA performance within each strategy. Higher is better for both metrics.

	Text-Only		**Code-Only**		**Concatenation (Early Fusion)**	
Model Architecture	**NNA (%)↑**	**CCI↑**	**NNA (%)↑**	**CCI↑**	**NNA (%)↑**	**CCI↑**
General-Purpose Models						
TEXT-EMBEDDING-3-SMALL (OPENAI)	**94.62**	0.40	73.32	0.23	91.33	0.32
TEXT-EMBEDDING-3-LARGE (OPENAI)	91.44	0.35	65.38	0.14	86.62	0.25
QWEN3-EMBEDDING-0.6B	88.77	0.29	75.81	0.22	91.80	0.31
QWEN3-EMBEDDING-4B	89.53	0.29	77.09	0.21	95.33	0.39
MGTE-LARGE-EN-V1.5	91.44	0.39	72.99	0.23	**95.38**	0.44
Code-Specialized Models						
NOMIC-EMBED-CODE	—	—	**78.29**	0.27	—	—
CODERANKEMBED	—	—	63.87	0.21	—	—
CODEBERT-BASE	—	—	54.36	0.02	—	—
GRAPHCODEBERT-BASE	—	—	49.83	0.01	—	—
CODET5-SMALL	—	—	55.66	0.09	—	—

8) Analysis and Interpretation: The results in Table C.2 and Figure C.1 reveal several patterns that inform the embedding configuration used in ATLAS.



(a) NNA vs. Weighting (α)



(b) CCI vs. Weighting (α)

Fig. C.1: *Dual-View (Late Fusion) Sensitivity Analysis.* We sweep the mixing parameter α from 0.0 (code-only) to 1.0 (text-only).

a) Text embeddings agree strongly with the curated broad-family labels: Text-only strategies achieve 89–95% NNA across all tested general-purpose models, substantially outperforming code-only approaches (50–78% NNA). This pattern is consistent across diverse model families, with the strongest text-only configuration (TEXT-EMBEDDING-3-SMALL, 94.62%) exceeding the best code-specialized encoder (NOMIC-EMBED-CODE, 78.29%) by more than 16 percentage points. Overall, these results show that the algorithm name and description provide a strong signal for reproducing the author-assigned broad-family labels in this

curated benchmark.

b) Model scale does not guarantee stronger representations: Smaller models match or outperform larger ones in several cases. Within the OpenAI family, TEXT-EMBEDDING-3-SMALL exceeds TEXT-EMBEDDING-3-LARGE across all three single-encoder strategies. Similarly, MGTE-LARGE-EN-V1.5 achieves the best concatenation result (95.38% NNA) while remaining far smaller than QWEN3-EMBEDDING-4B, which performs comparably (95.33%). These results suggest that, for this task, embedding quality depends on more than parameter count alone. One possible explanation is that smaller models may produce representations that are less sensitive to superficial lexical variation, although this mechanism is not directly tested here.

c) The benefit of concatenation is model-dependent: The effect of early fusion varies substantially across encoders. For MGTE-LARGE-EN-V1.5 and both Qwen models, concatenation improves over text-only performance (mGTE: +3.94pp, Qwen3-4B: +5.80pp, Qwen3-0.6B: +3.03pp). In contrast, both OpenAI models degrade under concatenation (OpenAI-Small: -3.29pp, OpenAI-Large: -4.82pp). This shows that combining text and code in a single input is not uniformly beneficial; its effectiveness depends on the embedding model.

d) Code contributes complementary information for compatible encoders: Although code-only representations are weaker than text-only baselines, source code can still improve representation quality when combined with text in compatible models. For example, concatenation improves both NNA and CCI for MGTE-LARGE-EN-V1.5 and the Qwen models. This suggests that code can provide useful disambiguating information beyond the textual description, especially when the encoder can integrate mixed text-code inputs effectively.

e) Late fusion offers high performance at greater complexity: Late-fusion strategies can achieve very strong results. The strongest configuration, using TEXT-EMBEDDING-3-SMALL for both the text and code views, reaches 97.40% NNA at $\alpha = 0.6$, and performance remains above 96% over a relatively broad range of α values. However, these gains come with added complexity: late fusion requires separate embeddings for text and code, maintains two representation views, and introduces the mixing weight α as an additional hyperparameter. This makes late fusion less convenient to deploy than a single-encoder alternative, especially when the performance gap is small.

9) Configuration Selection: Balancing Performance and Cohesion: For the main ATLAS experiments, we select **MGTE-LARGE-EN-V1.5 with concatenation**. While TEXT-EMBEDDING-3-SMALL with text-only achieves competitive NNA (94.62%), and late-fusion configurations reach higher peak accuracy (97.40%), MGTE-LARGE-EN-V1.5 with concatenation provides the most practical balance across multiple criteria:

- **Strong Label Agreement:** 95.38% NNA is the best result among the single-encoder strategies, with only a 2pp gap to the strongest late-fusion configuration.
- **Strong Cohesion Relative to the Curated Labels:** CCI of 0.437 is the highest among all strategies. Higher cohesion indicates tighter grouping and clearer separation relative to those labels, which supports the selected representation for neighborhood and clustering operations on the curated benchmark but does not independently validate evolved archive clusters.
- **Architectural Simplicity:** Single-encoder concatenation requires one embedding call per algorithm rather than two, reducing computational overhead and eliminating the need to tune a fusion weight.
- **Accessibility:** MGTE-LARGE-EN-V1.5 is open-source and moderate in size, improving reproducibility without API dependencies or large specialized code models.

Although the strongest late-fusion configuration achieves slightly higher peak NNA, the margin over MGTE-LARGE-EN-V1.5 with concatenation is small. Given its superior CCI and strong NNA performance, together with its simplicity and accessibility, MGTE-LARGE-EN-V1.5 with concatenation is the embedding configuration used in ATLAS.

B. Initialization Strategy Analysis: Validity and Embedding-Space Coverage

1) Aim and Research Question:

a) Aim: A key challenge in full algorithm synthesis is to generate a diverse and valid initial archive. Prior work in LLM-based code generation has explored prompt variation, including persona-based prompting [9], as a way to induce diversity in generated outputs. However, it remains unclear whether such prompt-level variation is sufficient to produce broad embedding-space coverage in full-algorithm synthesis. This study evaluates whether ATLAS's dual-phase initialization, which combines zero-reference generation with reference-based divergence, yields a more dispersed set of valid initial algorithms under the selected embedding metrics than prompt variation alone.

b) Research question: Does ATLAS's dual-phase initialization yield a more diverse set of valid initial algorithms than prompt variation alone?

2) Setup: We evaluate three initialization strategies across four LLM architectures: GPT-4.1-MINI, GPT-5 MINI (configured with low reasoning effort), CLAUDE HAIKU 4.5, and LLAMA 4 MAVERICK. The study is conducted on two combinatorial optimization domains: CVRP (50 customers, 31 instances) and FSS (50 jobs, 10 machines, 31 instances). These are non-trivial synthesis settings in which generated algorithms may fail due to invalid logic, infeasibility, or execution errors, making them suitable for evaluating both diversity and validity during initialization. Evaluating across multiple models and problem types helps reduce the chance that the observations are tied to a single architecture or domain.

Each configuration generates 50 algorithms per trial with temperature $T = 1.0$. Experiments are repeated across 5 independent trials, and all reported results are averaged across trials. The three initialization strategies are:

1) **Baseline (Stochastic Only):** All 50 algorithms are generated via the *Create* operator using a fixed default persona (Table C.3). Diversity arises only from sampling stochasticity.
2) **+Random Persona:** All 50 algorithms are generated via *Create*, with the persona randomly sampled from the full persona library (Table C.3) for each generation. This tests whether prompt-level variation can improve the diversity of the initial algorithms.
3) **+Divergence:** This corresponds to ATLAS's dual-phase initialization strategy. Following the default ATLAS configuration, 20 algorithms are first generated via *Create* using the default persona to form an initial reference pool. Then, 30 additional algorithms are generated incrementally via the DIVERGE operator, also using the default persona. At each step, 3 references are sampled from the current pool and passed to the DIVERGE operator, which generates a new algorithm intended to be distinct from those references; the resulting valid algorithm is then added back to the pool.

a) Persona Design: For this initialization analysis, we constructed a diverse library of prompt personas spanning several stylistic and design-oriented preferences, including philosophical stance (e.g., modernist, traditionalist, exotic), optimization priority (e.g., efficiency, robustness, scalability), and complexity preference (e.g., simplicity, sophistication). Table C.3 lists the full persona library used in the **+Random Persona** configuration.

TABLE C.3: Persona library used in the initialization-strategy analysis. The `default` persona is used in ATLAS; the remaining personas are used to evaluate prompt-level variation during initialization.

Persona Name	**Persona Text**
`default`	You are an expert computer scientist specializing in algorithm design for combinatorial optimization problems.
Single-Dimension Personas	
`simplicity_focused`	You are an algorithm designer who prioritizes elegant, minimal solutions.
`complexity_embracing`	You are an algorithm designer who builds sophisticated, feature-rich solutions.
`modernist`	You are an algorithm designer who uses cutting-edge techniques and recent innovations.
`traditionalist`	You are an algorithm designer who relies on proven, standard approaches.
`exotic_explorer`	You are an algorithm designer who seeks unusual, creative solutions others might miss.
`efficiency_focused`	You are an algorithm designer who optimizes for speed and resource usage.
`robustness_oriented`	You are an algorithm designer who prioritizes reliability and error handling.
`scalability_minded`	You are an algorithm designer who designs for large-scale applications.
Composite Personas	
`modern_simple`	You are a modernist algorithm designer who values simplicity and elegance.
`modern_efficient`	You are a modernist algorithm designer who optimizes for speed and efficiency.
`modern_robust`	You are a modernist algorithm designer who ensures reliability and robustness.
`traditional_simple`	You are a traditionalist algorithm designer who values simplicity and proven methods.
`traditional_robust`	You are a traditionalist algorithm designer who focuses on reliability and robustness.
`traditional_scalable`	You are a traditionalist algorithm designer who designs for scalability using proven techniques.
`exotic_efficient`	You are an exotic algorithm designer who prioritizes efficiency through creative approaches.
`exotic_robust`	You are an exotic algorithm designer who ensures robustness with unusual techniques.
`exotic_scalable`	You are an exotic algorithm designer who achieves scalability through innovative methods.
`simple_efficient`	You are an algorithm designer who values both simplicity and computational efficiency.
`simple_robust`	You are an algorithm designer who combines minimalism with strong reliability.
`complex_efficient`	You are an algorithm designer who builds sophisticated solutions optimized for performance.
`complex_robust`	You are an algorithm designer who creates feature-rich, highly reliable solutions.
`complex_scalable`	You are an algorithm designer who develops comprehensive solutions for large-scale problems.

b) Metrics: We quantify diversity in the embedding space using two complementary metrics. Each algorithm is embedded using MGTE-LARGE-EN-V1.5 with concatenated name, description, and preprocessed code, following the embedding configuration selected in Appendix C-A. For a set of N algorithms with embeddings $\{\mathbf{e}_i\}_{i=1}^{N}$, where $\mathbf{e}_i$ denotes the embedding of algorithm a_i:

TABLE C.4: *Initialization-strategy analysis.* Results are averaged over 5 runs of 50 algorithms each. Valid denotes the fraction of generated algorithms that pass validation. MNND and MPD measure local and global diversity in embedding space, respectively; higher is better for all three metrics. Best results are shown in **bold**.

	Baseline			+Random Persona			+Divergence (ATLAS)		
Model	Valid↑	MNND↑	MPD↑	Valid↑	MNND↑	MPD↑	Valid↑	MNND↑	MPD↑
CVRP($n50$)									
GPT-4.1-MINI	0.96	0.015	0.035	0.98	0.023	0.045	0.93	**0.058**	**0.151**
GPT-5-MINI (low)	0.99	0.037	0.103	0.98	0.034	0.097	0.98	**0.062**	**0.159**
CLAUDE HAIKU 4.5	0.86	0.029	0.077	0.89	0.027	0.068	0.90	**0.064**	**0.147**
LLAMA 4 MAVERICK	0.85	0.020	0.095	0.87	0.019	0.094	0.84	**0.039**	**0.123**
FSS($n50m10$)									
GPT-4.1-MINI	0.93	0.019	0.069	0.97	0.022	0.077	0.92	**0.053**	**0.147**
GPT-5-MINI (low)	0.99	0.021	0.049	0.99	0.021	0.051	0.98	**0.065**	**0.160**
CLAUDE HAIKU 4.5	0.96	0.030	0.078	0.98	0.030	0.081	0.93	**0.069**	**0.158**
LLAMA 4 MAVERICK	0.85	0.017	0.100	0.93	0.017	0.106	0.89	**0.033**	**0.142**

Mean Nearest Neighbor Distance (MNND): Measures local sparsity:

$$\text{MNND} = \frac{1}{N}\sum_{i=1}^{N} \min_{j \neq i} d(\mathbf{e}_i, \mathbf{e}_j) \tag{C.9}$$

Mean Pairwise Distance (MPD): Measures global spread:

$$\text{MPD} = \frac{2}{N(N-1)}\sum_{i=1}^{N}\sum_{j=i+1}^{N} d(\mathbf{e}_i, \mathbf{e}_j) \tag{C.10}$$

where $d(\mathbf{e}_i, \mathbf{e}_j) = 1 - \cos(\mathbf{e}_i, \mathbf{e}_j)$ is the cosine distance between embeddings. MNND captures whether algorithms are well separated from their nearest neighbors, while MPD captures the overall spread of the set in embedding space.

3) Results and Analysis: Table C.4 presents results on both CVRP and FSS. Across the tested settings, *+Divergence*, which is the initialization strategy used in ATLAS, consistently yields the most diverse initial algorithm sets. In contrast, random persona variation shows no consistent advantage over baseline stochastic sampling.

(1) Prompt variation provides limited and inconsistent diversity gains. The *+Random Persona* strategy does not consistently improve diversity. On CVRP, it reduces diversity for three of four models (GPT-5-MINI (low): MNND -8%, CLAUDE HAIKU 4.5: MNND -7%, LLAMA 4 MAVERICK: MNND -5%), while improving it only for GPT-4.1-MINI (MNND +53%). On FSS, the gains are similarly small, with only modest changes across models (0–16% MNND and 1–12% MPD). In an inspected sample set, persona variation often changed surface-level style, such as variable naming, comment verbosity, or code organization, without consistently changing the recognized algorithmic approach. Together with the embedding metrics, this suggests that persona variation alone is a weak mechanism for broadening the initial archive in this setting.

(2) *+Divergence* consistently increases diversity. The *+Divergence* strategy outperforms the other two strategies across all models, both problems, and both diversity metrics (16 out of 16 comparisons). On CVRP, relative improvements range from 68% to 287% for MNND and from 29% to 331% for MPD. On FSS, they range from 94% to 210% for MNND and from 42% to 227% for MPD. These results indicate that reference-based divergence is much more effective than stochastic sampling or persona variation at producing well-separated initial algorithms in embedding space. Conditioning generation on multiple existing references is intended to move proposals away from already represented regions; the reported result establishes greater embedding-space separation.

(3) The diversity gains come with only a modest impact on validity. The *+Divergence* strategy maintains high validity rates across all models in both domains. On CVRP, validity ranges from 84–98%, representing decreases of 1–3 percentage points for most models, while CLAUDE HAIKU 4.5 improves from 86% to 90%. On FSS, validity ranges from 89–98%, with small changes of up to 4 percentage points; most models show slight decreases, while LLAMA 4 MAVERICK improves from 85% to 89%. A slight reduction in validity is expected, since encouraging generation away from existing references can push the model toward less familiar solution patterns, which may increase the chance of invalid outputs. In the ATLAS setting, this trade-off is acceptable because initialization is followed by downstream validation and repair, allowing the method to benefit from broader early coverage while keeping failure rates manageable.

Implication for initialization design: Across both CVRP and FSS, and across all four tested LLMs (GPT-4.1-MINI, GPT-5-MINI (low), CLAUDE HAIKU 4.5, and LLAMA 4 MAVERICK), *+Divergence* consistently produces more diverse initial algorithm sets than either baseline stochastic sampling or persona variation. These results support ATLAS's use of reference-based divergence in its dual-phase initialization.

C. Semantic Search-Guidance Ablation

1) Aim and Research Question:

a) Aim: ATLAS organizes the evolving archive using semantic embeddings, grouping similar candidates into operational regions for retrieval and representative selection. These clusters guide the three-layer search strategy by defining (i) a local elite neighborhood for intensive refinement (Layer 1), and (ii) a set of diverse cluster representatives for distributed refinement and exploratory synthesis (Layers 2 and 3). This search-guidance ablation replaces both embedding-based neighbor selection and archive grouping with two alternative guidance strategies that do not rely on semantic embeddings, while retaining the remaining ATLAS machinery.

b) Research question: Does using semantic embeddings for Layer 1 neighbor selection and archive grouping improve search performance relative to fitness-based or random guidance?

2) Setup: We evaluate on two representative combinatorial optimization problems: CVRP (50 customers, 31 instances) and FSS (50 jobs, 10 machines, 31 instances). For each problem, all variants use the same training and test instance sets and follow the evaluation protocol described in §IV-A. Benchmark instances are generated using the default settings in Appendix B-B. Each variant is executed for $R = 5$ independent synthesis runs with different random seeds, under the common budget used in the end-to-end ATLAS search studies: 500 evaluated operator executions per run.

a) Controlled Adaptive Number of Clusters: To ensure a fair comparison and avoid confounding effects from different cluster counts, all variants use the same archive-driven adaptive rule for determining the number of clusters:

$$K_t = \min\Big(\max\big(\lfloor |\mathcal{A}_t|/d \rfloor,\ K_{\min}\big),\ K_{\max}\Big), \tag{C.11}$$

where $|\mathcal{A}_t|$ is the archive size at iteration t, d is a divisor targeting approximately d algorithms per cluster (default d=10), $K_{\min}$=3 ensures that at least two clusters remain for Layers 2 and 3 after removing the cluster containing the best algorithm a^*, and $K_{\max}$=10 follows the square-root heuristic ($\sqrt{|\mathcal{A}_{\max}|} = 10$) to limit cluster granularity. Thus, all variants use *the same number of clusters*; they differ only in *how algorithms are assigned to those clusters*.

b) Unified Architecture with Guidance-Specific Neighbor Selection and Grouping: To isolate the tested search-guidance strategy, all variants follow the same ATLAS architecture but differ in how Layer 1 neighbors are selected and how the archive is partitioned into groups.

At each iteration:

1) **Form the elite set (Layer 1):** Select the best algorithm a^* and its four nearest neighbors according to the variant's similarity metric (five algorithms total).
2) **Partition the archive:** Partition the archive $\mathcal{A}_t$ into K_t clusters using the variant's grouping method.
3) **Exclude the best cluster:** Remove the cluster containing a^* to avoid redundant exploration around the current best region.
4) **Form Layers 2 and 3:** Select the best-performing representative from each remaining cluster.

Search-Guidance Strategies:

- **Variant A: Semantic guidance**
  - *Elite selection:* k-nearest neighbors in embedding space (cosine similarity over combined text and code embeddings)
  - *Clustering:* K-Medoids applied in embedding space
- **Variant B: Fitness-based guidance**
  - *Elite selection:* a^* plus four algorithms with the closest fitness values
  - *Clustering:* Sort algorithms by fitness and partition them into K_t contiguous bands
- **Variant C: Random guidance**
  - *Elite selection:* a^* plus four uniformly random algorithms
  - *Clustering:* Randomly shuffle the archive and partition it into K_t equal-sized groups

All variants share the same archive management mechanisms (embedding-based deduplication and pruning), the same elite set size (five algorithms), the same adaptive cluster count (K_t from Eq. (C.11)), and the same representative selection rule (the best-performing algorithm in each cluster). They differ only in the rules used for Layer 1 neighbor selection and archive grouping, which together guide search across the three layers.

3) Results and Analysis: Table C.5 reports relative mean test gaps for the algorithm selected by training performance in each run of each variant.

a) Analysis: Semantic search guidance (Variant A), which uses embeddings for both Layer 1 neighbor selection and archive grouping, achieves the best performance on both benchmarks. This combined treatment outperforms the corresponding fitness-based and random guidance strategies. It therefore shows that the selected semantic representation provides more effective search guidance than the tested alternatives. Fitness-based guidance still introduces useful structure by relating algorithms with similar performance and outperforms random guidance on both benchmarks. However, algorithms with similar fitness can still differ substantially in their underlying search logic. Semantic guidance instead relates algorithms using representation-level similarity derived from their descriptions and code.

TABLE C.5: *Relative mean gaps for the semantic search-guidance ablation.* Percent gap to the strongest human-designed reference in each setting (PyVRP for CVRP and IG-TB for FSS); lower is better. Values are computed from the underlying raw objective means. Boldface mirrors the paired raw-objective analysis and is not based on a test of the displayed gaps.

Guidance Strategy	**CVRP**($n50$) gap (%)↓	**FSS**($n50m10$) gap (%)↓
Variant A: Semantic Guidance	**0.570**	**0.391**
Variant B: Fitness-Based Guidance	2.601	0.715
Variant C: Random Guidance	3.143	0.884

The combined semantic guidance can benefit the three-layer strategy in several ways. In Layer 1, embedding-based nearest neighbors can make local refinement around the current best algorithm more targeted. In Layer 2, each representative is refined with algorithms from its own embedding-based group, providing a more coherent local context for representative-level refinement. In Layer 3, semantic grouping supplies representatives separated under the same representation used by the archive. The observed final-quality gain is consistent with this context improving cross-group synthesis.

Table C.5 reports only the performance of the selected best algorithms. It therefore supports a conclusion about final search performance, but does not measure repertoire diversity, mechanistic family alignment, or deployment complementarity.

At the same time, the fitness-based and random variants remain reasonably strong because they still retain much of the ATLAS architecture. In all variants, Layer 1 continues to refine the current best algorithm, Layer 2 still distributes refinement through group representatives, and Layer 3 still performs cross-group COMBINE and DIVERGE operations. Moreover, all variants continue to benefit from the same semantically managed archive (embedding-based deduplication and pruning) and diverse initialization. Thus, the gains from semantic search guidance should be interpreted as the joint added value of semantically informed neighbor selection and grouping within the full ATLAS system, rather than as the sole source of search effectiveness.

Overall, these results support the narrower conclusion that using the selected semantic representation for neighbor selection and archive grouping improves final search guidance relative to the tested fitness-based and random strategies.

D. Search-Layer Ablation

1) Aim and Research Question:

a) Aim: ATLAS uses a three-layer search architecture to balance local refinement, distributed search across multiple archive regions, and cross-cluster synthesis. Layer 1 performs intensive refinement around the current best algorithm and its nearest semantic neighbors. Layer 2 performs distributed refinement over representatives from multiple embedding clusters, enabling search across separated archive regions. Layer 3 performs exploratory synthesis between cluster representatives to generate algorithms that may combine ideas from different regions or move toward previously underexplored ones. This ablation compares selected enabled and disabled layer combinations while keeping the remaining ATLAS components fixed.

b) Research question: How does ATLAS performance change across the selected layer configurations, and what is lost when local or representative-level refinement is removed?

2) Setup: We evaluate on the same two representative problem classes used in the other component ablations and design analyses: CVRP (50 customers, 31 instances) and FSS (50 jobs, 10 machines, 31 instances). For each problem, all variants use the same training and test instance sets and follow the evaluation protocol described in §IV-A. Each variant is executed for $R = 5$ independent synthesis runs with different random seeds, under the same computational budget of 500 evaluated operator executions per run. All configurations use the default ATLAS components unless explicitly ablated.

a) Layer Configurations::

- **Variant A: All layers active (default)**
  - *Layer 1:* local intensive refinement around the current best algorithm
  - *Layer 2:* distributed refinement over representatives from multiple clusters
  - *Layer 3:* cross-cluster synthesis and divergence
- **Variant B: Layer 1 only**
  - Uses only local intensive refinement around the current best algorithm and its local neighborhood
  - Disables distributed refinement over cluster representatives and cross-cluster synthesis
- **Variant C: Layers 2+3 only**
  - Disables local intensive refinement around the current best algorithm
  - Retains distributed refinement over cluster representatives and cross-cluster synthesis
- **Variant D: Layer 3 only**
  - Disables both local refinement and representative-level refinement
  - Uses only cross-cluster synthesis and divergence between cluster representatives
  - Emphasizes structured exploration across distinct archive regions, while still grounding synthesis in high-performing representatives

This ablation tests whether the complete configuration improves over the selected reduced configurations.

3) Analysis: Table V reports the test-set relative mean gap for each ablation configuration. The full three-layer architecture (Variant A) achieves the best performance on both benchmarks among the four configurations, which is consistent with the layers serving complementary roles. The second-best results are obtained by Variant C (Layers 2+3 only), followed by Variant D (Layer 3 only), while Variant B (Layer 1 only) performs worst.

The comparison between Variant A and Variant C shows that Layer 1 is important: removing intensive local refinement around the current best algorithm leads to a performance drop on both benchmarks. This indicates that allocating additional search effort to the neighborhood of the best discovered algorithm improves final solution quality.

However, the lower performance of Variant B shows that Layer 1 alone is not sufficient. Although it benefits from a strong warm start through the dual-phase initialization, which already provides a diverse initial archive (Appendix C-B), once search is restricted to Layer 1 it can only continue refining around the current best algorithm and its local neighborhood. Without Layers 2 and 3, the configuration loses explicit representative-level and cross-cluster operations. Layer 1 can still modify complete source and may generate internally hybrid code, but it no longer receives the cross-cluster reference combinations supplied by Layer 3. The performance result therefore supports broader archive-level search, not a claim that Layer 1 is incapable of producing any qualitatively new mechanism.

The strong performance of Variant C shows that ATLAS derives benefit from distributed search over cluster representatives together with cross-cluster synthesis. Even without the intensive local refinement of Layer 1, these two layers can still maintain competitive performance by refining multiple promising regions of the archive and generating new candidates through interactions across them. This supports the ATLAS design intuition that full-algorithm search benefits from allocating effort beyond the current-best neighborhood to representatives in multiple embedding-space regions.

Variant D (Layer 3 only) provides a direct comparison with Variant C for estimating the contribution of Layer 2 when Layer 3 remains active. It is weaker than Variant C but outperforms Variant B (Layer 1 only). The latter comparison shows that the Layer 3-only configuration is stronger than the Layer 1-only configuration in the tested settings. At the same time, Layer 3 should not be interpreted as purely exploratory. Because its synthesis operators use high-quality cluster representatives, often including the current best algorithm as a reference, Layer 3 also retains an exploitative component. It therefore acts as a structured exploration layer that recombines strong representatives from separated clusters rather than sampling without reference context.

Overall, the results are consistent with a division of labor across layers. Layer 1 strengthens local refinement around the current best algorithm, Layer 2 supports distributed refinement across multiple promising archive regions, and Layer 3 enables structured exploration through recombination and divergence across cluster representatives. The best results are obtained when all three layers are active, supporting the complete configuration over the selected reduced alternatives.

E. Reasoning-Effort Sensitivity and Cost Analysis

1) Aim and Research Question:

a) Aim: Modern reasoning-capable LLMs such as GPT-5-MINI expose a reasoning-effort parameter that controls how much internal reasoning is allocated before generating a response. Higher reasoning effort may improve the quality of discovered algorithms by enabling more careful modifications, but it also increases token usage, API cost, and latency per operator call. This sensitivity analysis studies whether increasing reasoning effort yields enough performance benefit to justify its additional computational cost in ATLAS.

b) Research question: How does reasoning effort affect the quality of discovered algorithms and the total token cost of search, and which reasoning level provides the best quality–cost trade-off?

2) Setup: We evaluate on the same two representative problem classes used in the component ablations and other design analyses: CVRP (50 customers, 31 instances) and FSS (50 jobs, 10 machines, 31 instances). For each problem, all variants use the same training and test instance sets and follow the evaluation protocol described in §IV-A. Each variant is executed for $R = 5$ independent synthesis runs with different random seeds, under the same computational budget of 500 evaluated operator executions per run.

All configurations use the default ATLAS architecture and GPT-5-MINI as the underlying synthesis model. They differ only in the assigned reasoning-effort level: minimal, low (default), medium, and high. This controlled sensitivity study examines the effect of reasoning effort on both synthesis quality and computational cost.

3) Results and Analysis: Table C.6 reports the relative mean test gaps and average total token usage per run for each reasoning level.

a) Analysis: Increasing reasoning effort from Minimal to Low improves performance on both benchmarks, showing that a modest increase in reasoning effort is beneficial for generating stronger algorithmic modifications. Importantly, this improvement comes at only a small increase in token usage: compared with Minimal, Low increases total tokens by approximately 3% on CVRP and 5% on FSS. Thus, Low provides a favorable improvement in solution quality at a relatively minor additional cost. At the same time, Minimal reasoning remains reasonably competitive, but is consistently weaker than the higher-effort settings.

Further increases beyond Low yield only limited additional gains. On CVRP, Low, Medium, and High are not statistically significantly different, indicating that additional reasoning budget does not translate into a reliable performance improvement.

TABLE C.6: *Reasoning-effort sensitivity and cost analysis for* GPT-5-MINI. Test performance is reported as the relative mean gap (%) to the strongest human-designed reference in each setting (PyVRP for CVRP and IG-TB for FSS); lower is better. Gaps are computed from the underlying raw objective means. Token usage is the mean total tokens consumed per run, in thousands, with standard errors in parentheses; each run uses 500 evaluated operator executions. Boldface mirrors the paired raw-objective analysis and is not based on a test of the displayed gaps.

Reasoning effort	**Relative mean gap (%)↓**		**Total tokens per run (k)**	
	CVRP($n50$)	**FSS**($n50m10$)	**CVRP**($n50$)	**FSS**($n50m10$)
Minimal	1.815	0.597	6 146 (135)	4 258 (178)
Low (default)	**0.094**	0.347	6 335 (144)	4 491 (246)
Medium	**0.092**	**0.301**	8 126 (161)	5 865 (272)
High	**0.000**	**0.292**	10 928 (241)	8 247 (294)

TABLE C.7: *Relative mean gaps for the underlying-LLM comparison on FSS.* Percent gap to IG-TB, the strongest human-designed reference for this setting; lower is better. Values are computed from the underlying raw objective means. Boldface mirrors the conclusion of the paired raw-objective analysis and is not based on a test of the displayed gaps.

Underlying LLM	**FSS** ($n50m10$) gap (%)↓
GPT-5-NANO (low)	0.557
GPT-5-MINI (low)	0.347
GPT-5 (low)	**0.284**
CLAUDE HAIKU 4.5 (default)	0.540

On FSS, Medium and High form the statistically best group and are not significantly different from each other, while Low falls outside this group; the associated improvement is nevertheless small in absolute terms, reducing the mean gap by less than 0.06 percentage points. The corresponding increase in token consumption, however, is substantial. Compared with Low, Medium increases token usage by approximately 28% on CVRP and 31% on FSS, while High increases it by approximately 73% on CVRP and 84% on FSS. These large increases in cost yield only marginal gains in solution quality.

Overall, these results suggest that, for GPT-5-MINI, Low reasoning provides the best practical operating point for ATLAS. It improves consistently over Minimal at only a modest additional cost, while Medium and High offer only diminishing returns relative to their substantially higher token usage, API cost, and latency. We therefore adopt Low reasoning as the default setting.

F. Effect of the Underlying LLM on ATLAS

This subsection examines how the underlying LLM affects ATLAS when the framework itself is otherwise kept fixed. We compare four models on FSS under the default ATLAS and benchmark settings: GPT-5-NANO (low), GPT-5-MINI (low), GPT-5 (low), and CLAUDE HAIKU 4.5 with its default setting (that is, without extended thinking). The GPT-5 family is evaluated at the same low reasoning level to keep the reasoning configuration consistent and better isolate differences due to model capability rather than reasoning budget. CLAUDE HAIKU 4.5 is included to provide a comparison with a model from a different provider. We first compare final test performance using one training-locked algorithm per run. We then inspect the training-selected archive representatives, using test gaps only for descriptive display and ordering. This descriptive view does not contribute to the headline comparison or its statistical tests.

1) Performance Comparison: Table C.7 shows that the underlying LLM affects ATLAS performance, although the stronger models remain relatively close. GPT-5 achieves the best mean performance, with GPT-5-MINI close behind, while GPT-5-NANO and CLAUDE HAIKU 4.5 are weaker but still achieve acceptable performance. This comparison establishes a difference in final selected-algorithm quality, but does not isolate whether the cause is local code quality, proposal diversity, or longer-term refinement dynamics.

2) Impact on Archive Diversity: Table C.8 reports a training-selected representative set from the displayed final archive for each LLM. Representative identities are fixed from training objectives, and the test-set gaps are used only to order and interpret the already selected repertoires. The table is not used in the headline LLM comparison or its statistical tests. Within this descriptive view, GPT-5-MINI and GPT-5 retain strong and structurally varied sets of competitive representatives. The inspected GPT-5-MINI representatives include NEH-style construction, large-neighborhood search, genetic search, spectral hybrids, and tabu-guided refinement; GPT-5 also includes backbone- and precedence-guided approaches. The displayed CLAUDE HAIKU 4.5 representatives are more concentrated around simulated-annealing and memetic variants, while the displayed GPT-5-NANO representatives have a wider test-gap spread. These training-selected examples illustrate the repertoire and hybridization capabilities of ATLAS, but do not establish comparative archive-diversity performance or a causal model effect on family discovery.

TABLE C.8: *Training-selected FSS archive representatives by LLM.* Representative identities are fixed from training objectives as the training-best members of final semantic clusters. Within each LLM, the displayed set is ordered by test-set relative mean gap only for interpretation. Values in parentheses are test-set relative mean gaps (%) to IG-TB, the strongest human-designed FSS reference; lower is better. This descriptive ordering does not affect representative membership, the training-selected deployable algorithm, the headline LLM comparison, or statistical tests.

LLM	**Training-selected representatives, ordered by descriptive test gap**
GPT-5-NANO (LOW)	Permutation Based Flow Shop Heuristic with Cascaded Insertion and Local Search (0.474%); Hybrid Seeded Cascaded Insertion with Global Lookahead (0.501%); Fixed Permutation Flow Shop Seeded Local Search (0.527%); Diversified Guided Cascaded Beam Search for Permutation Flow Shop (0.753%); Hybrid Prefix Guided Insertion with Global Lookahead and Prefix Preserving Local Search (1.333%); Genetic Algorithm based on permutation population with crossover and mutation (1.372%); Greedy NEH-inspired permutation for flow shop makespan minimization (2.809%); Recursive Skyline Block Sequencing (3.319%); Hybrid Pheromone Guided Time Slice Sequencing (3.383%); Pheromone Guided Permutation Flow Shop Scheduler (3.483%)
GPT-5-MINI (LOW)	Priority-Seeded Prefix-Guided LNS with Lightweight Anneal (0.331%); Priority-seeded NEH with Guided Neighborhood Search (0.458%); Priority-Spectral Hybrid with Tabu-Repair Local Search (0.472%); Prefix-Aware Hybrid Genetic with Annealed Polish (0.478%); Sampled Pairwise Greedy with Bounded Cache and Adaptive Iterated Refinement (0.527%); Hybrid NEH-Regret Genetic Search with Aggressive Complexity Reduction (0.714%); Guided Greedy Adaptive Cache with Pheromone and Block Relocation (0.965%); Bottleneck-Guided Greedy with ILS and Simulated Annealing (1.239%)
GPT-5 (LOW)	Pairwise backbone scheduling with regret beam construction and consensus edge relinking (0.277%); Precedence corridor scheduling with shockwave rebuild and edge voted relinking (0.280%); Vote guided anchor beam scheduling with cached insertion scoring and focused reconstruction (0.304%); Multi-sequence NEH flow shop heuristic with CDS, Palmer, and elite reinsertion search (0.391%); Consensus rollout scheduling with hot segment repair and block guided relinking (0.454%); Pairwise precedence learning with exact insertion and conflict focused segment reconstruction (0.460%); Tournament backbone search with window dynamic programming (0.800%); Cross-entropy random-key optimization with block permutation polishing (2.746%)
CLAUDE HAIKU 4.5	Hybrid Adaptive Simulated Annealing with Dynamic Critical Path Guidance (0.484%); Simulated Annealing with Critical Path Optimization for Flow Shop Scheduling (0.571%); Simulated Annealing with Greedy Initialization for Flow Shop Scheduling (0.579%); Hybrid Memetic Algorithm with Adaptive Dual-Strategy Exploration (0.809%); Hybrid Adaptive Metaheuristic with Critical Path Learning (1.016%); Hybrid Initialization and Local Search for Flow Shop Scheduling (1.237%); Hybrid Memetic Strategy with Dual-Phase Evolution and Adaptive Intensity Balancing (1.537%); Genetic Algorithm with Dual Crossover Operators and Adaptive Multi-Strategy Mutations (1.558%); Constraint-Based Graph Routing with Bottleneck-Aware Optimization (2.181%)