Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

GenePlan: Evolving Better Generalized PDDL Plans Using Large Language Models

Andrew Murray, Danial Dervovic, Alberto Pozanco, Michael Cashmore

J.P. Morgan AI Research

{andrew.murray, danial.dervovic, alberto.pozancolancho, michael.cashmore}@jpmorgan.com

Abstract

We present GenePlan (GENeralized Evolutionary Planner), a novel framework that leverages large language model (LLM) assisted evolutionary algorithms to generate domain-dependent generalized planners for classical planning tasks described in PDDL. By casting generalized planning as an optimization problem, GenePlan iteratively evolves interpretable Python planners that minimize plan length across diverse problem instances. In empirical evaluation across six existing benchmark domains and two new domains, GenePlan achieved an average SAT score of 0.91, closely matching the performance of the state-of-the-art planners (SAT score 0.93), and significantly outperforming other LLM-based baselines such as chain-of-thought (CoT) prompting (average SAT score 0.64). The generated planners solve new instances rapidly (average 0.49 seconds per task) and at low cost (average $1.82 per domain using GPT-4o).

Extended version — https://arxiv.org/abs/2603.09481

1 Introduction

Large language models (LLMs) are neural networks that have been trained on vast amounts of data and have been applied successfully to a broad spectrum of tasks: from question-answering to writing code (Bubeck et al. 2023). Despite recent advances in LLM reasoning (Guo et al. 2025), their use in sequential decision making tasks such as planning has thus far shown sub-par performance (Valmeekam et al. 2022).

Recent advances in the field of *generalized planning* (Jiménez, Segovia-Aguas, and Jonsson 2019) has shown promise, whereby purpose built Python planners are designed by the LLM to solve new instances in a given domain (Silver et al. 2024). Generating such solutions requires significant domain knowledge, a task that the LLM is well suited to, given its vast training set. However these approaches focus on generating satisficing solutions with no consideration for plan quality. This is insufficient in many practical applications where the quality of the solution matters.

In tandem, there has been some success in using LLMs as *optimizers*, where the optimization problem and its solution are defined in natural language (Yang et al. 2023). Romera-Paredes et al. (2024) build upon this by embedding an LLM within an evolutionary optimization framework. In this approach, the solution is a Python method and the optimization space is any valid Python code. The authors applied this framework to generate new heuristic functions for combinatorial optimization problems and found that the LLM was able to come up with heuristics surpassing the best human-generated heuristic.

In this paper, we build upon both these recent lines of work. In particular, we extend the chain-of-thought (CoT) approach taken by Silver et al. (2024), to use the evolutionary LLM framework introduced by Romera-Paredes et al. (2024).

Our method, GenePlan, treats generalized planning as an *optimization* problem. The goal is to produce a generalized plan, written in Python, that minimizes the number of actions across a training set of planning tasks. GenePlan iteratively develops a population of candidate Python methods (generalized plans). The LLM proposes new methods, which are evaluated for plan quality. These methods are stored in the population and fed back to the LLM through evolutionary sampling. The population evolves, with the lowest-performing members pruned each generation, converging on high-quality solutions. The optimal solution, an efficient Python method, is extracted at the end, capable of generating *high-quality* and *interpretable* solutions for new PDDL planning tasks in the domain.

We empirically evaluate our approach on 8 domains, comparing 10 baseline approaches and show that our approach is comparable in terms of plan quality with state of the art planners given a 30 minute time limit. The resulting Python planner is interpretable and is capable of solving new planning instances rapidly (an average of 0.49 seconds per task). The average cost for generating a planner using our approach was just $1.82 per domain.

In section 2 we describe the background relevant to this work. In section 3 we place the contribution of this paper in context with related work. In section 4 we outline the GenePlan procedure. In section 5 and 6 we describe the setup and results of our experimental evaluation. We conclude and address avenues for future research in section 7.

2 Background

2.1 PDDL Planning

In this paper, we focus on classical, deterministic, fully observable planning tasks specified in the Planning Domain Definition Language (PDDL) (Ghallab et al. 1998). For illustration, we refer to the trading domain, where agents navigate a map, trade resources, and deposit them in inventories.

Definition 1 (PDDL Planning Task)**.** *A PDDL planning task $\Pi = (\mathcal{D}, \mathcal{P})$, consists of a domain $\mathcal{D}$ and problem $\mathcal{P}$. The domain $\mathcal{D} = (T, P, A)$, defines how the world works and is composed of a set of object types T, predicates P and actions A. The problem $\mathcal{P} = (O, s_0, S_g)$ contains the task specific information, including the object instances O, the initial state s_0 and the set of goal states $S_G \subseteq S$.*

A *type* $t = \{o_1, o_2, \ldots, o_n\}$, is a set of objects sharing common properties. A *predicate* $p : t_1 \times t_2 \times \cdots \times t_m \rightarrow \{\text{True}, \text{False}\}$, is a boolean function with object arguments and typing and arity requirements. For example, `located(?p - person ?l - location)` is a predicate with arity 2, where the first argument is a `person` and the second a `location`. A ground atom $p(o_1, o_2, \ldots, o_n)$, is a predicate applied to specific object instances: for example `located(p1 l1)` indicates that person `p1` is `located` at location `l1`.

Planning can be expressed as a search problem: $\Phi = (\Sigma, s_0, S_g)$, where $\Sigma = (S, A, \gamma)$ is the state transition system with set of states S, actions A and state transition function $\gamma : S \times A \rightarrow S$ defining which actions can be legally applied in which state.

Each *state* $s = \bigwedge_{i=1}^{k} p_i(o_1, o_2, \ldots, o_n)$, is a conjunction of true ground atoms. Actions have preconditions $pre(a)$, add effects $eff^+(a)$, delete effects $eff^-(a)$, and may have an associated cost $c(a)$. The default cost is 1. An action a is applicable in s if $pre(a) \subseteq s$, such that $\gamma(s, a) = s'$ yields a new state $s' = (s \setminus eff^-(a)) \cup eff^+(a)$. See Listing 1 for an example from the trading domain. An example action instance could be: `(deposit p1 r1 l1 i1)`. This action would be applicable in a state if the following preconditions hold: the person `p1` is `located` at the location `l1`, `l1` `contains` the inventory `i1` and `p1` is `carrying` the resource `r1`. In the resulting state, `p1` will no longer be `carrying` `r1` and `r1` will be `deposited` in `i1`.

```
(:action deposit
    :parameters (?p - person ?r - resource ?l - location
      ?i - inventory)
    :precondition (and (located ?p ?l) (contain ?i ?l)
        (carrying ?p ?r))
    :effect (and (not (carrying ?p ?r)) (deposited ?r ?i
    )))
```

Listing 1: Example action description

The goal of planning is to find a sequence of actions, or plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$. A plan is *valid* if, the actions applied sequentially to the initial state s_0, reaches a goal state: $\gamma(\ldots\gamma(\gamma(s_0, a_1), a_2), \ldots, a_n) \in S_g$. The cost of a plan is denoted as $c(\pi) = \sum_{i=1}^{n} c(a_i)$.

2.2 Generalized Planning

Generalized planning refers to the process of generating strategies that are valid for multiple problem instances (Martín and Geffner 2004):

Definition 2 (Generalized Planning Instance)**.** *A generalized planning instance is a finite set of classical planning tasks $\Pi_G = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$, sharing a common structure.*

In line with recent definitions (Jiménez, Segovia-Aguas, and Jonsson 2019), we restrict our analysis to the case where the tasks have the same PDDL domain, such that only the objects, initial state and goals are different. A generalized plan, is a solution to a generalized planning instance:

Definition 3 (Generalized Plan)**.** *A generalized plan Φ for Π_G is a function that maps all problem instances $\Pi \in \Pi_G$ to valid plans π.*

Note, to avoid confusing the term *generalized plan* (the method which outputs a plan for a given planning task) and *plan* (the solution to the planning task); we henceforth refer to a generalized plan as a *planner*.

2.3 Generalized Planning as an Optimization Problem

In this paper we are interested in finding *high quality* plans. We can do this by treating the problem as an *optimization problem*, where we seek to find the planner Φ, minimizing plan cost $c(\pi)$, across all tasks $\Pi \in \Pi_G$.

Definition 4 (Generalized Planning Optimization Problem)**.** *A generalized planning optimization problem is an optimization problem:*

$$\arg\min_{\Phi} \frac{1}{|\Pi_G|} \sum_{\Pi \in \Pi_G} c(\Phi(\Pi)) \tag{1}$$

We consider only the problem of minimizing the number of actions in the plan, such that plan cost $c(\pi) = |\pi|$, referred to as *plan length*, hence the overall objective function is just the mean plan length across all tasks.

2.4 Evolutionary Algorithms

Given an optimization problem: $\arg\min_x f(x)$, where $\mathcal{X}$ is the feasible solution space and $f : \mathcal{X} \rightarrow \mathbb{R}$ is the objective function to be minimized, the goal is to find $x^* \in \mathcal{X}$ such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{X}$. Evolutionary algorithms are meta-heuristic optimization methods which can find good approximate solutions to complex optimization problems by evolving a population of candidates through selection, crossover, mutation, and replacement.

Definition 5 (Population)**.** *Evolutionary algorithms maintain a population of candidate solutions: $\mathcal{P}(t) = \{x_1^t, x_2^t, \ldots, x_\mu^t\}$, where $x_i^t \in \mathcal{X}$. The population $\mathcal{P}(t)$ at generation t consists of μ individual candidates, such that each candidate x_i^t is a feasible solution to the optimization problem: $\min_{x \in \mathcal{X}} f(x)$.*

Candidates are evaluated using a *fitness function*, which quantifies the quality of the candidate. This guides the evolutionary process by determining which solutions are more likely to survive and reproduce.

Definition 6 (Fitness Function)**.** *The fitness function $\hat{f} : \mathcal{X} \to \mathbb{R}$, maps each candidate solution to a value measuring how well the solution solves the optimization problem.*

On each iteration, new candidates are identified by applying three operators: *selection*, *crossover* and *mutation*. Parents for crossover are chosen via the *selection* operator.

Definition 7 (Selection)**.** *A selection operator $\tilde{S} : \mathcal{X}^\mu \to \mathcal{X}^k$, chooses individuals from the current population to become parents for producing offspring. Given the current population $\mathcal{P}(t)$, the selection operator produces a parent pool $\mathcal{P}_{parents}(t) = \tilde{S}(\mathcal{P}(t))$.*

Selection is directly based on the fitness function values $\hat{f}(x)$, with better-fit individuals having higher probability of being selected for reproduction. The crossover operator combines multiple parents to generate offspring.

Definition 8 (Crossover)**.** *A crossover operator $C : \mathcal{X}^k \to \mathcal{X}^\lambda$, takes k selected parents $s_1, s_2, \ldots, s_k$, and generates λ offspring $y_1, \ldots, y_\lambda = C(x^t_{s_1}, \ldots, x^t_{s_k})$ where each $x^t_{s_j} \in \mathcal{P}_{parents}(t)$.*

The mutation operator introduces random changes to maintain diversity. Mutation is typically applied with a small probability $p_m \ll 1$, to ensure that genetic material is mostly preserved while still allowing for occasional exploration of new solution characteristics.

Definition 9 (Mutation)**.** *A mutation operator $M : \mathcal{X} \to \mathcal{X}$, introduces random variations to a candidate solution. For each offspring y_j produced by crossover, mutation can be applied to create a modified offspring $y'_j = M(y_j)$.*

After selection, crossover, and mutation, λ offspring are produced and added to the cumulative offspring population, $\mathcal{P}'(t)$. This process repeats until reaching a threshold number of offspring λ_{max}. The next generation $\mathcal{P}(t+1)$ is then formed by selecting μ individuals from $\mathcal{P}(t) \cup \mathcal{P}'(t)$ using a replacement strategy (Smith 2007).

Definition 10 (Replacement Strategy)**.** *A replacement strategy $R : \mathcal{X}^\mu \times \mathcal{X}^\lambda \to \mathcal{X}^\mu$, constructs the next generation population $\mathcal{P}(t+1)$ by selecting exactly μ individuals from the current population $\mathcal{P}(t)$ and/or the offspring population $\mathcal{P}'(t)$. Formally: $\mathcal{P}(t+1) = R(\mathcal{P}(t), \mathcal{P}'(t)) = \{x^{t+1}_1, x^{t+1}_2, \ldots, x^{t+1}_\mu\}$.*

At any generation t, the incumbent solution x^*_t, is the best solution found so far: $x^*_t = \arg\min_{x \in \{\mathcal{P}(t) \cup \mathcal{P}'(t)\}} \hat{f}(x)$. The process repeats for G generations, after which the best solution x^*_G is returned as an approximation to the global optimum x^*. Evolutionary algorithms do not guarantee optimality, but often yield high-quality solutions when exact optimization is infeasible. See Algorithm 1 for pseudocode.

Algorithm 1: Evolutionary Algorithm Pseudocode

1. Initialize population $\mathcal{P}(0) = x^0_1, x^0_2, \ldots, x^0_\mu$
2. Evaluate fitness $\hat{f}(x^0_i)$ for each individual in $\mathcal{P}(0)$
3. Set $t \leftarrow 0$
4. While $t \leq G$:
   (a) Initialize offspring population $\mathcal{P}'(t) \leftarrow \emptyset$
   (b) While $|\mathcal{P}'(t)| < \lambda_{\text{max}}$
      i. Select parents $\mathcal{P}_{\text{parents}}(t)$ based on fitness values
      ii. Apply crossover to produce offspring $y_1, \ldots, y_\lambda$
      iii. Apply mutation to offspring to produce $y'_1, \ldots, y'_\lambda$
      iv. Evaluate fitness of offspring: $\hat{f}(y'_j)$ for each $j \in 1, \ldots, \lambda$
      v. $\mathcal{P}'(t) \leftarrow \mathcal{P}'(t) \cup \{y'_1, \ldots, y'_\lambda\}$
   (c) Apply replacement strategy to form new population $\mathcal{P}(t+1)$
   (d) $t \leftarrow t + 1$
5. Return best solution found $x^*_G = \arg\min_{x \in \mathcal{P}(G)} \hat{f}(x)$

3 Related Work

In this section we discuss relevant related work on planning using LLMs. For a comprehensive overview we refer the reader to a relevant survey (Pallagani et al. 2024; Chiari et al. 2025).

Hybrid Approaches Early studies found LLMs had limited success generating plans directly from PDDL (Silver et al. 2022; Valmeekam et al. 2023). Hybrid approaches overcome this by using LLMs to translate natural language to PDDL which is then solved by downstream planners (Liu et al. 2023; Dagan, Keller, and Lascarides 2025; Zhang et al. 2025; Mahdavi et al. 2024; Guan et al. 2023). These approaches often require expert intervention to ensure the generated PDDL is correct and unambiguous. Errors or ambiguities in translation can result in unsolvable or incorrect plans, and the process may fail to capture the full range of user intent due to the inherent ambiguity of natural language.

LLM + Search Several works used LLMs to perform specific search functions: for example generating successor and goal test functions (Katz et al. 2025), action selection (Stein et al. 2025) and heuristic generation (Meng et al. 2024; Corrêa, Pereira, and Seipp 2025). These approaches all rely on conventional, often costly, search procedures. In contrast, GenePlan directly generates efficient, interpretable Python implementations, eliminating the need for integration with existing search algorithms.

Instruction Tuning Verma et al. (2025) fine-tune an LLM on labeled planning tasks with explicit reasoning about action preconditions, effects, and state transitions. They enhance performance through step-by-step CoT reasoning with external validation. However, this requires direct model weight access for retraining, while GenePlan works with API-only access, making it more broadly accessible.

Generalized Planning Generalized planning (Martín and Geffner 2004) has been studied without generative AI, e.g., by learning abstract actions (Bonet, Frances, and Geffner 2019) or policies via deep RL (Rivlin, Hazan, and Karpas 2020); see (Jiménez, Segovia-Aguas, and Jonsson 2019) for a survey. Winner and Veloso (2003, 2007) learn conditional and looping strategies by exploiting the underlying problem

structure. These approaches learn simple strategies without accounting for any domain specific heuristics that could improve the plan quality.

Wang et al. (2024) use LLMs to generate planners by creating natural language hints and strategies, converting them into solution sketches, then translating to pseudocode and finally Python code. This approach is not designed for PDDL planning problems. Silver et al. (2022, 2024) showed that CoT prompting can extract Python strategies for generalized planning. A similar approach is taken by Chen et al. (2025) using few-shot-prompting. These approaches are satisficing and are not optimized. Our work addresses this gap by focusing on plan quality optimization.

Evolutionary LLMs Recent work has demonstrated that evolutionary frameworks leveraging LLMs can automatically generate and refine Python methods for combinatorial optimization (Romera-Paredes et al. 2024; Liu et al. 2024). By iteratively applying selection, crossover, and mutation to candidate heuristics, these approaches have produced solutions that surpass the best human-designed heuristics. Liu et al. (2024) further introduced prompt strategies inspired by evolutionary algorithms to enhance exploration and diversity in generated solutions. This approach has been successfully applied to other tasks, for example automated feature engineering (Abhyankar, Shojaee, and Reddy 2025; Murray, Dervovic, and Cashmore 2025; Gong et al. 2025), drug discovery (Wang et al. 2025) and generating financial trading strategies (Yuksel and Sawaf 2025a,b). Our work extends this paradigm to generalized PDDL planning.

4 Method

Our insight is to frame the Generalized Planning Optimization Problem (Definition 4) as searching for a generalized planner Φ within the feasible set $\mathcal{X}$ of executable code. In our experiments, $\mathcal{X}$ is restricted to valid Python code, though the approach is language-agnostic. We introduce GenePlan, an evolutionary algorithm that evolves a population of planners to approximately minimize Equation (1).

The GenePlan architecture is shown in Figure 1 and is inspired by Romera-Paredes et al. (2024). It can be thought of as an evolutionary algorithm, whereby the right loop (1-9) creates candidate offspring planners (stored in the `planner_db`) via selection and LLM-assisted recombination and mutation; and the left loop (10-11) prunes the worst performing planners via replacement. After G generations, the best planner is extracted from the `planner_db`. Inputs include a prompt template, the PDDL domain $\mathcal{D}$, and training tasks $\Pi_{train} \subseteq \Pi_G$. The initial population can be seeded with a provided Python planner or generated via CoT prompting (Silver et al. 2024). Details of each component are described below.

Evaluate Fitness Each candidate is stored alongside its fitness function score within the `planner_db`. The fitness function is computed in the *evaluate fitness*, block 8 in Figure 1. As a surrogate of Equation (1), our fitness function simply computes the average plan length of the plans found

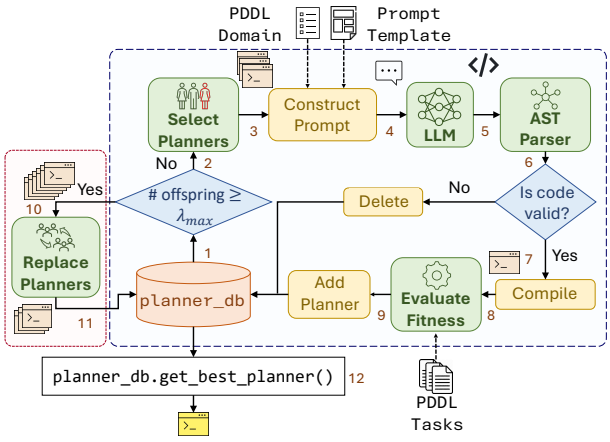

Figure 1: Figure showing architecture of GenePlan. Python planners are stored in the `planner_db`. The right loop (1-9) generates new candidate planners, while the left loop (10-11) prunes low scoring candidates at the end of each generation. After running, the best planner can be extracted by querying the `planner_db` (12)

by the planner on the set of training tasks:

$$\hat{f}(\Phi) = \frac{1}{|\Pi_{train}|} \sum_{\Pi \in \Pi_{train}} |\Phi(\Pi)| \tag{2}$$

Note that each plan output from the planner is first validated using a plan validator (Howey and Long 2003) to ensure validity. We define a failure score $F \gg 0$, such that if a planner Φ us unable to solve a particular instance $\Pi \in \Pi_{train}$, then the score for that problem $|\Phi(\Pi)| = F$.

Select Planners Parent planners are selected in the *select planners* part (2) in Figure 1. As per Romera-Paredes et al. (2024), probabilities are assigned to each planner based on their fitness scores. Unlike Romera-Paredes et al., we chose to use an inverse decaying function: $T = a/|\mathcal{P}(t) \cup \mathcal{P}'(t)| + b$ to model the temperature T, at any iteration as a function of the current number of planners in the `planner_db`, $|\mathcal{P}(t) \cup \mathcal{P}'(t)|$, where $b = (T_{\min}(\mu + \lambda_{\max}) - T_{\max})/(\mu + \lambda_{\max} - 1)$ and $a = T_{\max} - b$. The temperature T decays hyperbolically from $T_{\max}$ to $T_{\min}$ as the number of planners in the `planner_db` increases from 1 to $\mu + \lambda_{\max}$. We use $T_{\max} = 50$, $T_{\min} = 10$ and $\mu = 10$ and $\lambda_{\max} = 10$ in our experiments.

Given the temperature T at any given iteration, we can compute the probability of sampling a planner Φ_j as:

$$P_{\Phi_j} = \frac{e^{\left(-\hat{f}(\Phi_j)/T\right)}}{\sum_{\Phi \in \mathcal{P}(t) \cup \mathcal{P}'(t)} e^{\left(-\hat{f}(\Phi)/T\right)}} \tag{3}$$

The lower the temperature, the higher probability is assigned to planners with low values of $\hat{f}$. Since the temperature is hyperbolically decaying, this has the effect of encouraging exploration early on in the generation, and exploitation towards the end. Equation (3) defines a discrete probability distribution P over the set of planners $\mathcal{P}(t) \cup \mathcal{P}'(t)$,

where each planner Φ_j has probability P_{Φ_j} of being sampled. The distribution satisfies $\sum_{\Phi \in \mathcal{P}(t) \cup \mathcal{P}'(t)} P_\Phi = 1$ and $P_\Phi \geq 0$ for all Φ.

The selection operator $\tilde{S}$ samples k parent planners without replacement from $\mathcal{P}(t) \cup \mathcal{P}'(t)$ according to P_Φ, yielding $\mathcal{P}_{\text{parents}}(t) = \Phi_{s_1}, \Phi_{s_2}, \ldots, \Phi_{s_k}$. Note that we sample from both the current population and cumulative offspring, unlike Definition 5. If fewer than k planners are available, we sample $\min(|\mathcal{P}(t) \cup \mathcal{P}'(t)|, k)$; crossover is skipped when only one planner can be sampled.

Prompt Construction The prompt template used in the experiments is provided in Figure 2. The placeholder `@@domain@@` is replaced with the PDDL domain. Parent planners sampled in the selection phase are inserted in place of `@@examples@@` to construct an n-shot prompt (3). Note that for each planner, if the code fails, we catch the exception and pass the error message `planner.error`. If the planner fails to solve some problems, then we pass the result of the plan validator `planner.plan_failure_error` for these instances. This prompt is then passed to the LLM (4) and a new offspring candidate y' is generated. Note that this step encompasses both the evolutionary *crossover* and *mutation* operators.

AST Parser At (5), the LLM outputs Python code as a string. Prior to execution we validate it using an abstract syntax tree (AST) parser. This ensure it adheres to a predefined set of allowable nodes, packages, functions, and attributes, serving as a guardrail. If the code is valid (6), it is compiled and executed to produce a method, stored with the code string as a planner object. The fitness of valid offspring planners is evaluated (8) and then added to the `planner_db` (9). Invalid planners are deleted, returning to (1).

Replace Planners This process repeats (1 - 9), until the number of offspring planners stored in the `planner_db` reaches the maximum λ_{max}. At this point replacement occurs in the left loop (10-11), whereby we replace the current population with the best performing candidates from the generation. As a replacement strategy, we initialize the next generation's population using the planners with the lowest (best) fitness score: $\mathcal{P}(t+1) = \{\Phi \in \mathcal{P}(t) \cup \mathcal{P}'(t) \mid \text{rank}_{\hat{f}}(\Phi) \leq \mu\}$, where $\text{rank}_{\hat{f}}(\Phi)$ gives the rank of individual Φ when all individuals in $\mathcal{P}(t) \cup \mathcal{P}'(t)$ are sorted by their fitness values $\hat{f}$ in ascending order: $\text{rank}_{\hat{f}}(\Phi) = 1 + |\{\Phi_j \in \mathcal{P}(t) \cup \mathcal{P}'(t) \mid \hat{f}(\Phi_j) < \hat{f}(\Phi)\}|$. Note that we select the next generation from both the previous generation $\mathcal{P}(t)$, and the current offspring $\mathcal{P}'(t)$. This is referred to as $\mu + \lambda$ selection (μ members in the population + λ offspring) or an *elitist* replacement strategy (Beyer and Schwefel 2002).

5 Experimental Setup

We experimentally validated our method on 8 PDDL domains. Separate train (for use within GenePlan) and test (for evaluation) problems were generated for each domain. We used 30 test problems for each domain and 5-10 training tasks. A summary of the domains and their source are provided below. Most of these domains have been extracted

```
You have access to the following PDDL domain:

@@domain@@

Your goal is to implement a method called get_plan in
Python that solves problem instances for this domain
with the fewest possible actions. The method should
be of the form:

def get_plan(objects, init, goal):
    """
    Description of heuristic.
    """
    Your code here
    return plan

where
- `objects` is a @@typing@@.

- `init` is a set of ground atoms represented as
tuples of predicate names and 1 or more arguments
(e.g., ('predicate-foo', 'object-bar', ...))

- `goal` is also a set of ground atoms represented in
the same way

- `plan` is a list of actions, where each action is a
ground operator represented as a string (e.g.,
'(operator-baz object-qux ...)')

I will now provide you with some example methods.
Below each method you will be provided with a system
message telling you whether the code worked or not.
This will be either:

System: The code did not work. Error: {planner.error}.
System: The code failed to solve some problems.
Error: {planner.plan_failure_error}. Score
{planner.score}
System: The code worked. Score: {planner.score}.

The score is the average plan length across all test
problems (lower is better). Here are the examples:

@@examples@@

EVOLUTIONARY OPTIMIZATION INSTRUCTIONS:

Given these examples, your task is to use
evolutionary optimization to generate a better
heuristic:

- Perform crossover: Combine the best algorithmic
components from each method. Merge these segments in
a way that preserves their functionality.

- Apply mutations: Introduce strategic modifications
such as efficiency improvements, novel heuristic
calculations, redundant action elimination, or
alternative goal prioritization.

- Optimize for fitness: Optimize to minimize the
average plan length (score). Consider both solution
quality and computational efficiency.
```

Figure 2: Evolutionary prompt template.

from prior studies (Yang et al. 2022; Silver et al. 2024). To ensure that all of the domains are not within the LLMs training set, we also created two entirely new domains (trading and research). All domains contain problems of varying sizes and initial and goal configurations. We include below, details of problem sizes for the new domains [1]. Full PDDL for these domains is provided in the extended version of this paper (Murray et al. 2026).

- **Trapnewspapers** (Yang et al. 2022): Pick up newspapers from a base and deliver them to specified locations.
- **Hiking** (Yang et al. 2022): Navigate a map, climbing hills and avoiding water obstacles.
- **Manygripper** (Yang et al. 2022): Use a robot with two grippers to transport balls between locations.
- **ManyFerry** (Yang et al. 2022): Transport cars between islands using ferries.
- **Manymiconic**(Yang et al. 2022): Pick up and drop off passengers in multi-building elevator systems.
- **Heavypack**(Silver et al. 2024): Stack items in a box, only allowing heavier items to be stacked upon.
- **Research**: Researchers read papers, run experiments, write papers and advisors accelerate learning.
- **Trading**: Agents navigate, gather resources, deposit in inventories and trade.

We compare against the following baselines:

- **cot4** (Silver et al. 2024): CoT prompting with GPT-4.
- **cot4o**: CoT prompting with GPT-4o.
- **evo**: The evolutionary planner proposed in this paper, using GPT-4o.
- **evo_ds**: Evo variant where the LLM generates a natural language domain summary for the prompt instead of using the PDDL domain.
- **evo_mini**: Evo variant using GPT-4o mini.
- **evo_abl**: Evo variant with predicate, object and action names ablated. This was done by replacing names with generic names like `action1` and `predicate1`.
- **evo_nev**: Evo variant with no evaluator; all planners receive constant scores.
- **fd_x**: Fast Downward (Helmert 2006) LAMA setting (Richter and Westphal 2010), an anytime portfolio approach with time bound x ($x = 300$s, $x = 1800$s).
- **fd_opt**: Fast Downward SEQ-OPT-LMCUT setting running A^* with the LMCUT heuristic (Helmert and Domshlak 2009) (30 minute limit).

We used the unified planning framework (Micheli et al. 2025) for all experiments.

6 Results

Results are provided in Table 1.

[1]Problem sizes for new domains—Research: train (5-10 researchers, 2-5 projects, 10-20 papers/experiments), test (10-20 researchers, 5-15 projects, 20-50 papers/experiments); Trading: train (2-5 agents/inventories, 10-20 locations/resources), test (5-10 agents/inventories, 20-30 locations/resources).

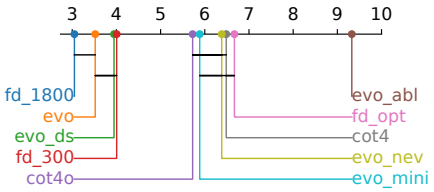

Figure 3: Critical difference diagram showing average rank per method across all problem instances. Lower ranks (left) are better and the horizontal lines connecting approaches indicate statistical indistinguishability.

SAT Score To compare planner performance across domains and baselines, we use the SAT score $SAT_m(\Pi) = c(\pi^*)/c(\pi_m)$, from the International Planning Competition (IPC) (Taitler et al. 2024), where $c(\pi^*)$ is the cost of the best plan π^*, found across all methods and $c(\pi_m)$ is the cost from the method m under evaluation. A score of 1 means the method found the best plan, whereas 0 indicates an unsolved task. Table 1 presents the mean and standard error SAT scores for all methods and domains. Our evolutionary planner (evo) achieved an average SAT score of 0.91 across all domains, closely matching the performance of Fast Downward with a 30-minute time limit (0.93). Notably, evo outperformed other LLM-based baselines with an average improvement in SAT score of 0.27 versus cot4o (0.64).

We observe that using a domain summary (evo_ds) performed well in simpler domains, such as *manygripper* and *trapnewspapers*, but struggled in more complex domains like *manymiconic*. This suggests that the completeness of domain representation in the prompt can significantly impact plan quality. Consistent with prior studies (Silver et al. 2024), we found that ablating names from the PDDL severely impairs the LLM's ability to generate working solutions. This highlights the importance of contextual information for effective plan synthesis. It's worth mentioning that fd_opt achieved a much lower score (average of 0.42) than fd_1800 with the same time limit. In many domains, fast downward failed to find an optimal plan within 30 minutes resulting in a score of 0 on that problem. In contrast, fd_1800 returns the best plan found within the time limit.

We would like to highlight that while our experimental analysis used proprietary models (GPT-4o and GPT-4o mini), newer, open source models such as Deepseek-V3 and R1 are available and have shown superior performance on math and coding benchmarks (Guo et al. 2025).

Relative Performance across Problem Instances We performed a comprehensive statistical comparison across all 240 problem instances solved (30 per domain). For each task, methods were ranked by their performance scores, and these ranks were averaged to assess overall effectiveness. To determine whether differences in method performance were statistically significant, we used the Friedman test (Friedman 1940), which compares multiple algorithms across multiple datasets. A significant Friedman test result ($p < 0.05$) indicates that at least one method performs differently. To

method	domain metric	heavypack	hiking	manyferry	manygripper	manymiconic	research	trading	trapnewspapers
fd_300	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	90.0	43.33	83.33
	Runtime	21.19 ± 2.05	5.45 ± 0.27	18.69 ± 4.20	63.57 ± 13.28	22.86 ± 5.55	83.48 ± 12.57	219.94 ± 15.79	71.69 ± 16.64
	Score	**1.00 ± 0.00**	0.89 ± 0.05	0.93 ± 0.01	0.94 ± 0.01	0.98 ± 0.00	0.90 ± 0.06	0.43 ± 0.09	0.64 ± 0.05
fd_1800	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	93.33	86.67
	Runtime	10.04 ± 0.89	2.73 ± 0.09	110.76 ± 49.71	88.17 ± 27.73	49.12 ± 18.65	31.87 ± 4.28	394.48 ± 50.64	216.66 ± 30.44
	Score	**1.00 ± 0.00**	0.89 ± 0.05	0.95 ± 0.01	**0.96 ± 0.01**	**0.99 ± 0.00**	**1.00 ± 0.00**	**0.93 ± 0.05**	0.70 ± 0.05
fd_opt	% Solved	**100.0**	**100.0**	56.67	26.67	53.33	0.0	0.0	0.0
	Runtime	11.49 ± 1.37	12.68 ± 0.87	744.50 ± 143.88	953.29 ± 207.66	87.54 ± 28.83	-	-	-
	Score	**1.00 ± 0.00**	**1.00 ± 0.00**	0.57 ± 0.09	0.27 ± 0.08	0.53 ± 0.09	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
cot4	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	0.0	**100.0**	**100.0**	**100.0**
	Runtime	1.15 ± 0.09	0.91 ± 0.02	**0.18 ± 0.01**	**0.45 ± 0.01**	-	5.95 ± 0.51	0.24 ± 0.01	0.10 ± 0.00
	Score	**1.00 ± 0.00**	0.55 ± 0.06	0.90 ± 0.01	0.90 ± 0.01	0.00 ± 0.00	0.06 ± 0.00	0.21 ± 0.01	0.75 ± 0.00
cot4o	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	0.0	**100.0**	**100.0**	**100.0**
	Runtime	1.25 ± 0.10	0.90 ± 0.02	0.21 ± 0.01	0.50 ± 0.01	-	0.93 ± 0.05	0.15 ± 0.00	0.12 ± 0.00
	Score	**1.00 ± 0.00**	0.55 ± 0.06	0.92 ± 0.01	0.90 ± 0.01	0.00 ± 0.00	0.11 ± 0.01	0.88 ± 0.01	0.75 ± 0.00
evo_nev	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	16.67	**100.0**	**100.0**	**100.0**
	Runtime	1.24 ± 0.10	0.90 ± 0.02	0.24 ± 0.02	0.47 ± 0.01	0.23 ± 0.05	5.88 ± 0.50	0.25 ± 0.01	0.10 ± 0.00
	Score	**1.00 ± 0.00**	0.55 ± 0.06	0.90 ± 0.01	0.90 ± 0.01	0.11 ± 0.05	0.06 ± 0.00	0.19 ± 0.01	0.75 ± 0.00
	Cost	1.26	1.45	1.65	1.66	1.94	2.32	1.9	1.22
	Gen Time	460.61	602.82	620.1	718.18	13359.07	853.06	901.03	462.0
evo_abl	% Solved	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Runtime	-	-	-	-	-	-	-	-
	Score	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	Cost	-	-	-	-	-	-	-	-
	Gen Time	-	-	-	-	-	-	-	-
evo_mini	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	0.0	**100.0**	**100.0**	**100.0**
	Runtime	1.14 ± 0.09	1.03 ± 0.06	0.19 ± 0.01	0.46 ± 0.01	-	5.90 ± 0.50	**0.14 ± 0.00**	0.11 ± 0.00
	Score	**1.00 ± 0.00**	0.55 ± 0.06	0.90 ± 0.01	0.91 ± 0.01	0.00 ± 0.00	0.06 ± 0.00	0.92 ± 0.01	0.75 ± 0.00
	Cost	**0.07**	**0.09**	**0.08**	**0.1**	**0.12**	**0.15**	**0.15**	**0.08**
	Gen Time	895.7	16105.96	1111.67	1128.4	61870.86	1589.72	2061.29	1088.75
evo_ds	% Solved	**100.0**	**100.0**	**100.0**	73.33	0.0	**100.0**	**100.0**	**100.0**
	Runtime	**1.12 ± 0.08**	0.93 ± 0.02	0.19 ± 0.01	0.50 ± 0.01	-	1.49 ± 0.11	0.15 ± 0.00	**0.09 ± 0.00**
	Score	**1.00 ± 0.00**	0.89 ± 0.05	**0.98 ± 0.00**	0.70 ± 0.08	0.00 ± 0.00	0.29 ± 0.01	0.91 ± 0.01	**1.00 ± 0.00**
	Cost	1.3	1.55	1.89	1.76	2.04	2.52	1.87	1.44
	Gen Time	**437.16**	**573.33**	721.06	**574.7**	**630.98**	**738.03**	659.57	1761.47
evo	% Solved	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
	Runtime	1.15 ± 0.09	**0.88 ± 0.02**	0.19 ± 0.01	0.49 ± 0.01	**0.20 ± 0.02**	**0.75 ± 0.04**	0.15 ± 0.00	0.10 ± 0.00
	Score	**1.00 ± 0.00**	0.89 ± 0.05	0.95 ± 0.01	0.90 ± 0.01	0.98 ± 0.01	0.71 ± 0.02	0.92 ± 0.01	0.94 ± 0.01
	Cost	1.26	1.67	1.62	2.02	2.25	2.81	1.83	1.11
	Gen Time	461.24	657.0	**610.51**	745.6	703.94	933.97	**653.72**	**396.38**

Table 1: Performance comparison across different methods and domains

identify which specific pairs of methods differ, we applied the Nemenyi post-hoc test (Nemenyi 1963), which accounts for the increased risk of false positives when making many pairwise comparisons. This ensures that the chance of incorrectly finding a significant difference remains below a chosen threshold (such as 0.05).

The results are summarized in a critical difference diagram (Demšar 2006), which plots the average ranks of all methods along a horizontal axis, with lower average ranks (better performing methods) positioned further to the left. Horizontal bars connect groups of methods whose differences are not statistically significant at the 0.05 level according to the Nemenyi test. Notably, evo and fd_1800 are grouped together, indicating comparable performance, while both evo and evo_ds are separated from cot4o, reflecting a statistically significant improvement over the cot4o baseline.

Dollar Cost We report the dollar cost of each method, reflecting the resources required to generate solutions using large language models (LLMs). Cost is determined by API usage fees, calculated using OpenAI pricing (OpenAI 2025): \$10/*M* output and \$2.5/*M* input tokens for GPT-4o, and \$0.6/*M* output and \$0.15/*M* input tokens for GPT-4o mini. On average, generating plans with GPT-4o cost just \$1.82 per domain, while GPT-4o mini averaged \$0.10 per domain but with lower plan quality (score 0.64 vs. 0.91 for GPT-4o). This performance gap is primarily due to the smaller size and reduced training data of GPT-4o mini, which hinders its reasoning capabilities on complex tasks.

Runtime For the evolutionary (evo) methods, we report both the generation time (*Gen Time*) (the total time required to run the evolutionary algorithm and produce a plan-

Domain	k
trading	1.66
trapnewspapers	1.83
manyferry	5.52
manygripper	8.5
manymiconic	14.39
research	30.01
heavypack	51.85
hiking	354.53

Table 2: Minimum number of instances k required for LLM-based planner to be more efficient than fd_1800 per domain.

ner—and) the runtime (the average time taken to solve each PDDL problem using the generated planner). In our experiments, evo required on average 645 seconds to generate a planner_db with the chosen parameters ($\mu = 10, \lambda_{\text{max}} = 10$, $G = 10$). Importantly, this generation time is a one-time cost; once the planner is produced, it can be reused to solve new instances rapidly, with an average runtime of just 0.49 seconds per plan—significantly faster than Fast Downward.

We define T_{gen} as the one-time planner generation time, T_{evo} as the average time to solve a single instance with the evolutionary planner, and T_{fd} as the average time per instance for Fast Downward (fd_1800). GenePlan is advantageous when the total time to solve k instances, $T_{\text{gen}} + kT_{\text{evo}}$, is less than kT_{fd}, which occurs for $k > T_{\text{gen}}/(T_{\text{fd}} - T_{\text{evo}})$. Table 2 lists the required k for each domain.

The results indicate that for complicated domains like trading, the evolutionary approach becomes advantageous after solving just a few instances. Importantly, our experiments did not use early stopping; for simple domains this would avoid unnecessary computation. As an example, Figure 4 shows the average plan length versus generation on the training tasks, normalized by the optimal value from fast downward. For some domains (trapnewspapers, heavypack and hiking), GenePlan converged on the global optima within a single generation. Future work will explore early stopping criteria, such as halting when improvement plateaus or using an LLM to infer when it believes it has found the optimal solution.

Domains with no Simple Strategy Importantly, simple, generalizable strategies are not always available for all planning domains. For example, in Sokoban, certain actions can lead to irreversible states—such as pushing a stone into a corner—making the goal unreachable. We tested GenePlan on this domain and found that it tried to build a search algorithm (Murray et al. 2026). The resulting planner failed to solve any problems, while fast downward with a time limit of 30 minutes solved most instances.

In such domains, exhaustive or heuristic search remains necessary. We are not interested in reinventing search based planners which are already highly optimized for this type of problem. An interesting direction would be using an LLM as an orchestration interface to dynamically select between GenePlan-generated and traditional search-based planners based on problem context. This could be facilitated by emerging tools such as the Model Context Protocol (MCP) (Hou et al. 2025). Finally, our approach could be used to generate specific components of search algorithms, for example successor state and goal tests (Katz et al. 2025) or heuristics (Corrêa, Pereira, and Seipp 2025), while leveraging existing search algorithms.

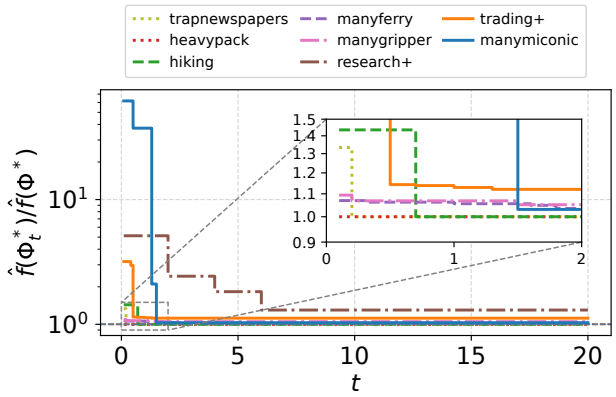

Figure 4: Normalized score $\hat{f}(\Phi_t^*)/\hat{f}(\Phi^*)$ versus generation t. $\hat{f}(\Phi^*)$ is the optimal average plan length found by Fast Downward on the training tasks (or best found within 1 hour for domains labelled +), and $\hat{f}(\Phi_t^*)$ is GenePlan's current incumbent solution.

7 Conclusion

In this paper, we introduced GenePlan, a novel framework that leverages evolutionary LLMs to generate cost-aware, domain-dependent planners for classical PDDL planning tasks. By framing generalized planning as an optimization problem and integrating LLMs within an evolutionary algorithm, our approach evolves interpretable Python planners that minimize plan length across diverse problem instances.

Our empirical evaluation across eight benchmark domains demonstrates that GenePlan achieves plan quality comparable to state-of-the-art planners such as Fast Downward, while offering significant advantages in interpretability and rapid inference. The one-time computational cost of generating a planner is offset by efficiency gains in repeated deployment, making our approach attractive for domains with recurring planning needs. We showed that using a natural language summary of the domain does not generally lead to improved performance, while ablation studies highlight the importance of contextual information for LLM-based planning. Our cost analysis indicates that high-quality planners can be generated at a reasonable expense.

We conclude by listing avenues for future work. The first involves developing early stopping criteria to reduce the computational cost of planner generation. Secondly, new optimization metrics beyond plan length should be investigated. Finally, as mentioned in Section 6, LLMs can be leveraged both to route domains to an appropriate solver and to generate better heuristic functions for search-based planning via evolutionary algorithms.

Acknowledgments

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan") and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

References

Abhyankar, N.; Shojaee, P.; and Reddy, C. K. 2025. LLM-FE: Automated Feature Engineering for Tabular Data with LLMs as Evolutionary Optimizers. *arXiv preprint arXiv:2503.14434*.

Beyer, H.-G.; and Schwefel, H.-P. 2002. Evolution Strategies–A Comprehensive Introduction. *Natural Computing*, 1(1): 3–52.

Bonet, B.; Frances, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2703–2710.

Bubeck, S.; Chandrasekaran, V.; Eldan, R.; Gehrke, J.; Horvitz, E.; Kamar, E.; Lee, P.; Lee, Y. T.; Li, Y.; Lundberg, S.; et al. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. *arXiv preprint arXiv:2303.12712*.

Chen, D. Z.; Zenn, J.; Cinquin, T.; and McIlraith, S. A. 2025. Language Models for PDDL Planning: Generating Sound and Programmatic Policies. In *Eighteenth European Workshop on Reinforcement Learning*.

Chiari, M.; Putelli, L.; Rossetti, N.; Serina, I.; and Gerevini, A. E. 2025. On planning through LLMs. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, 377–385.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

Dagan, G.; Keller, F.; and Lascarides, A. 2025. Dynamic Planning with a LLM. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

Demšar, J. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, 7(Jan): 1–30.

Friedman, M. 1940. A Comparison of Alternative Tests of Significance for the Problem of M Rankings. *The Annals of Mathematical Statistics*, 11(1): 86–92.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Gong, N.; Reddy, C. K.; Ying, W.; Chen, H.; and Fu, Y. 2025. Evolutionary Large Language Model for Automated Feature Transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 16844–16852.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-Trained Large Language Models to Construct and Utilize World Models for Model-Based Task Planning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, volume 36, 79081–79094.

Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, 162–169.

Hou, X.; Zhao, Y.; Wang, S.; and Wang, H. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *ACM Transactions on Software Engineering and Methodology*.

Howey, R.; and Long, D. 2003. VAL's Progress: the Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of the ICAPS Workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*, volume 685.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A Review of Generalized Planning. *The Knowledge Engineering Review*, 34: e5.

Katz, M.; Kokel, H.; Srinivas, K.; and Sohrabi Araghi, S. 2025. Thought of Search: Planning with Language Models Through The Lens of Efficiency. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, volume 37, 138491–138568.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *arXiv preprint arXiv:2304.11477*.

Liu, F.; Tong, X.; Yuan, M.; Lin, X.; Luo, F.; Wang, Z.; Lu, Z.; and Zhang, Q. 2024. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design using Large Language Model. In *Proceedings of the International Conference on Machine Learning*, 32201–32223.

Mahdavi, S.; Aoki, R.; Tang, K.; and Cao, Y. 2024. Leveraging Environment Interaction for Automated PDDL Generation and Planning with Large Language Models. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 38960–39008.

Martín, M.; and Geffner, H. 2004. Learning Generalized Policies from Planning Examples using Concept Languages. *Applied Intelligence*, 20: 9–19.

Meng, S.; Wang, Y.; Yang, C.-F.; Peng, N.; and Chang, K.-W. 2024. LLM-A*: Large Language Model Enhanced Incremental Heuristic Search on Path Planning. In *Proceedings of the Association for Computational Linguistics*, 1087–1102.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; et al. 2025. Unified Planning: Modeling, Manipulating and Solving AI Planning Problems in Python. *SoftwareX*, 29: 102012.

Murray, A.; Dervovic, D.; and Cashmore, M. 2025. ELATE: Evolutionary Language model for Automated Time-series Engineering. *arXiv preprint arXiv:2508.14667*.

Murray, A.; Dervovic, D.; Pozanco, A.; and Cashmore, M. 2026. GenePlan: Evolving Better Generalized PDDL Plans usin Large Language Models. *arXiv preprint arXiv:2603.09481*.

Nemenyi, P. B. 1963. *Distribution-Free Multiple Comparisons.* Princeton University.

OpenAI. 2025. OpenAI Platform Pricing. https://platform.openai.com/docs/pricing. Accessed: 2025-11-11.

Pallagani, V.; Muppasani, B. C.; Roy, K.; Fabiano, F.; Loreggia, A.; Murugesan, K.; Srivastava, B.; Rossi, F.; Horesh, L.; and Sheth, A. 2024. On the Prospects of Incorporating Large Language Models (LLMs) in Automated Planning and Scheduling (APS). In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 432–444.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning with Deep Reinforcement Learning. *arXiv preprint arXiv:2005.02305*.

Romera-Paredes, B.; Barekatain, M.; Novikov, A.; Balog, M.; Kumar, M. P.; Dupont, E.; Ruiz, F. J.; Ellenberg, J. S.; Wang, P.; Fawzi, O.; et al. 2024. Mathematical Discoveries from Program Search with Large Language Models. *Nature*, 625(7995): 468–475.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, 20256–20264.

Silver, T.; Hariprasad, V.; Shuttleworth, R. S.; Kumar, N.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. PDDL Planning with Pretrained Large Language Models. In *NeurIPS 2022 foundation models for decision making workshop*.

Smith, J. 2007. On Replacement Strategies in Steady State Evolutionary Algorithms. *Evolutionary Computation*, 15(1): 29–59.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2025. Automating the Generation of Prompts for LLM-Based Action Choice in PDDL Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; et al. 2024. The 2023 International Planning Competition.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023. On the Planning Abilities of Large Language Models-A Critical Investigation. In *Proceedings of the Conderence on Neural Information Processing Systems (NeurIPS)*, volume 36, 75993–76005.

Valmeekam, K.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2022. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning About Change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Verma, P.; La, N.; Favier, A.; Mishra, S.; and Shah, J. A. 2025. Teaching LLMs to Plan: Logical Chain-of-Thought Instruction Tuning for Symbolic Planning. *arXiv preprint arXiv:2509.13351*.

Wang, E.; Cassano, F.; Wu, C.; Bai, Y.; Song, W.; Nath, V.; Han, Z.; Hendryx, S.; Yue, S.; and Zhang, H. 2024. Planning in Natural Language Improves LLM Search for Code Generation. *arXiv preprint arXiv:2409.03733*.

Wang, H.; Skreta, M.; Ser, C.-T.; Gao, W.; Kong, L.; Strieth-Kalthoff, F.; Duan, C.; Zhuang, Y.; Yu, Y.; Zhu, Y.; et al. 2025. Efficient Evolutionary Search over Chemical Space with Large Language Models. In *Proceedings of the International Conference on Learning Representations*.

Winner, E.; and Veloso, M. 2007. LoopDistill: Learning Looping Domain-Specific Planners from Example Plans. In *International Conference on Automated Planning and Scheduling, Workshop on Artificial Intelligence Planning and Learning*.

Winner, E.; and Veloso, M. M. 2003. DISTILL: Learning Domain-Specific Planners by Example. In *Proceedings of the International Conference on Machine Learning*, 800–807.

Yang, C.; Wang, X.; Lu, Y.; Liu, H.; Le, Q. V.; Zhou, D.; and Chen, X. 2023. Large Language Models as Optimizers. *arXiv preprint arXiv:2309.03409*.

Yang, R.; Silver, T.; Curtis, A.; Lozano-Perez, T.; and Kaelbling, L. P. 2022. PG3: Policy-Guided Planning for Generalized Policy Generation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 31, 4686–4691.

Yuksel, K. A.; and Sawaf, H. 2025a. AlphaQuant: LLM-Driven Automated Robust Feature Engineering for Quantitative Finance. In *International Conference on Learning Representations - Workshop in Advances in Financial AI*.

Yuksel, K. A.; and Sawaf, H. 2025b. EvoRisk: Autonomously Discovered Regime-Adaptive Resilience-Aware Financial Metric. *Available at SSRN 5791002*.

Zhang, X.; Qin, H.; Wang, F.; Dong, Y.; and Li, J. 2025. Lamma-P: Generalizable Multi-Agent Long-Horizon Task Allocation and Planning with LM-Driven PDDL Planner. In *IEEE International Conference on Robotics and Automation (ICRA)*, 10221–10221. IEEE.