
LLM-Driven Composite Neural Architecture Search for Multi-Source RL State Encoding

Yu Yu
Shanghai Jiao Tong University

Qian Xie*
Cornell University

Nairen Cao
New York University

Li Jin[†]
Shanghai Jiao Tong University

Abstract

Designing state encoders for reinforcement learning (RL) with multiple information sources—such as sensor measurements and time-series signals—remains underexplored and often requires manual design. We formalize this challenge as a problem of composite neural architecture search (NAS), where multiple source-specific modules and a fusion module are jointly optimized. Existing NAS methods overlook useful side information about each module’s representation quality, limiting their sample efficiency in this multi-source RL setting. To address this, we propose an LLM-driven NAS pipeline that leverages language-model priors over module design choices and representation quality to guide sample-efficient search for high-performing composite state encoders. On a mixed-autonomy traffic control task, our approach discovers higher-performing architectures with fewer evaluations than traditional NAS baselines and the LLM-based GENIUS framework.

1 Introduction

Reinforcement learning (RL) often requires transforming raw observations into compact state representations. In many real-world domains, the state is observed through multiple heterogeneous sources—such as sensors, time-series signals, images, or text—necessitating specialized encoders for each source and a fusion module to combine them. Existing RL systems typically rely on *manually* designed encoders, which can be suboptimal and hard to generalize across environments. While neural architecture search (NAS) offers a way to automate encoder design, most NAS methods target single-modality supervised tasks and overlook the composite, multi-source nature of RL state encoding. Moreover, evaluating architectures in RL is notoriously costly, as each candidate must be trained through thousands of simulator interactions, making sample efficiency a key challenge.

We formulate the underexplored problem of *composite NAS* for RL state encoding, where multiple source-specific modules and a fusion module are jointly optimized. Beyond sample efficiency, a further challenge is leveraging *side information*—such as the representation quality or specialization of each module—signals that existing NAS methods typically ignore or cannot model effectively.

To address this limitation, we propose an LLM-driven NAS pipeline that leverages language-model priors over module design choices and their representation quality to automatically discover high-performing composite state encoder architectures for RL. Our main contributions are:

1. We introduce and formally define the problem of composite NAS for state encoding in RL with multiple information sources.

*Correspondence to: Qian Xie <QX66@CORNELL.EDU>.

[†]Correspondence to: Li Jin <LI.JIN@SJTU.EDU.CN>.

2. We propose an LLM-driven NAS pipeline that incorporates language-model priors to guide the search using side information about module representation quality.
3. We instantiate and evaluate the proposed framework on a mixed-autonomy traffic control task, demonstrating improved search efficiency and RL performance.

2 Background and Related Works

2.1 RL state encoding

State encoding is a form of representation learning that maps raw observations (e.g., images, textual descriptions, or sensor measurements) into compact latent representations that can be used by RL agents as input for policy and value estimation. Typical architectures include convolutional neural networks (CNNs) for image observations, Transformer or recurrent networks (e.g., LSTMs, GRUs) for textual or time-series inputs, and feed-forward networks (FFNs) for structured inputs.

Various strategies have been explored for obtaining such representations. A widely adopted approach is *end-to-end training*, where the encoder is jointly optimized with the RL policy using algorithms such as Proximal Policy Optimization (PPO), optimizing rewards directly. Alternatively, some works pretrain the encoder using self-supervised or contrastive learning objectives (e.g., representation consistency across views) before fine-tuning in RL. In this work, we focus on the end-to-end setting, where the encoder architecture is optimized jointly with the RL agent.

2.2 Neural architecture search

Neural architecture search (NAS) aims to automatically discover high-performing neural architectures. For RL, NAS methods can be applied either in conjunction with end-to-end training—searching for architectures that directly maximize task rewards—or in a two-stage manner where the architectures are optimized for auxiliary objectives such as contrastive loss, and then transferred to RL. In this work, we focus on the former, i.e., searching architectures trained end-to-end with the RL agent.

Traditional methods. A wide range of NAS algorithms have been developed, including gradient-based DARTS [7], RL-based ENAS [9], and evolutionary-based PEPNAS [12], and Bayesian optimization, including Gaussian process-based methods such as those implemented in BoTorch [1] with mixed-type (ordinal and categorical) kernels, as well as BOHB [6] and BANANAS [11].

LLM-based methods. Recently, LLMs have been used to guide architecture search by generating architecture descriptions or candidates to discover high-performing architectures. Representative methods include GENIUS [14], LLMatic [8], LAPT-NAS [15], and SEKI [2]. However, they are primarily designed for single-modality supervised learning tasks (e.g., image classification) and do not consider composite modules or RL state encoding.

2.3 LLM for RL

Yan et al. [13] investigates the use of LLMs as action priors to guide policy learning in RL, but not as neural architecture priors for RL state encoding. Recent surveys provide broader perspectives: Schoepp et al. [10] categorizes three roles of LLMs in RL—Agent, Planner, and Reward—and further discusses modifying LLM architectures to serve directly as state representations, while Cao et al. [3] categorizes LLMs into four roles—information processor, reward designer, decision-maker, and generator—with representation learning discussed under the generator role. In contrast, our work explores a complementary approach: employing LLMs to guide neural architecture search for state encoders, where the resulting architectures—not the LLM itself—form the state representation.

3 LACER: An LLM → State Encoder → RL Pipeline

Our approach, LLM-driven Neural Architecture Search for Composite State Encoders in RL (LACER) iteratively uses an LLM to generate candidate state encoder architectures, evaluates each candidate in an RL environment, and feeds the resulting performance back to the LLM.

3.1 Problem Setup

We consider an RL agent that interacts with an environment and receives observations composed of multiple input sources, such as sensor values, time-series signals, textual instructions, or image observations. Each input source may require a different type of neural architecture (e.g., MLP/FFN, Transformer, CNN, RNN) to extract relevant features. Instead of searching for a single shared encoder, our goal is to automatically discover a set of architecture modules—one for each input source—and a fusion module that combines their outputs into a final latent state representation.

Formally, let $x = (x_1, \dots, x_M)$ denote the raw observations from M input sources, and define the overall state encoder as

$$s = g_\phi(f_{\theta_1}(x_1), \dots, f_{\theta_M}(x_M)),$$

where each f_{θ_i} is a neural module for source i with design choices θ_i , and g_ϕ is a fusion module with design choices ϕ . Let \mathcal{E} denote the downstream RL environment. Our goal is to search over design choices $\{\theta_i\}_{i=1}^M$ and ϕ to maximize

$$\mathcal{M}(\pi \circ g_\phi \circ (f_{\theta_1}, \dots, f_{\theta_M}); \mathcal{E}),$$

where $\mathcal{M}(\cdot; \mathcal{E})$ measures the performance of the RL policy π in environment \mathcal{E} (e.g., average traffic speed or average return), which may differ from the reward used to train π .

Equivalently, varying θ_i selects a specific function f_{θ_i} from the family $\mathcal{F}_i = \{f_{\theta_i} \mid \theta_i \in \Theta_i\}$, and the search therefore operates over the Cartesian product $\Theta_1 \times \dots \times \Theta_M \times \Phi$ of design choices.

3.2 LLM-Driven Neural Architecture Generation

We begin with an expert-designed initial architecture. At each subsequent iteration, we query an LLM with a textual prompt that summarizes the current set of architecture modules and their associated performance. The LLM then responds with *one or a batch of new composite architecture candidates*.

For each module (including the source-specific encoders and the fusion block), the search is restricted to a module-specific architecture space typically used for that type of neural network (e.g., CNNs for image inputs, GRUs for textual instructions, transformers for time-series inputs, and FFNs for vector inputs). To illustrate this, Figure 4 and Table 1 in Appendix B provide an example of a composite architecture for the traffic-control benchmark, involving transformers for time-series encoders and FFNs for vector inputs. Similarly, Figure 7 and Table 2 provide an example of a composite architecture for the MiniGrid goal-oriented task benchmark, involving an image encoder, a text encoder, and a fusion encoder, along with their corresponding search spaces.

Finally, we convert the LLM output into executable architectures using simple tokenization and pattern matching (for implementation details, see Appendix B). For a comparison between the design of our method and other LLM-based NAS methods such as GENIUS, see Appendix A.

3.3 RL Training and Evaluation

Each generated composite architecture is trained in an end-to-end manner together with the RL agent. We train an RL algorithm (e.g., PPO) for a fixed number of interaction steps T . To provide richer feedback than existing methods (e.g., GENIUS), we feed the LLM not only the task metric but also the average reward and feature information (a measure of representation quality of each source-specific encoder; see Appendix A), offering additional context for refining candidate architectures. These three signals are jointly used as performance feedback for the next iteration. The RL policy architecture remains fixed; only the state encoder modules vary during the search. This LLM-training-evaluation loop is repeated until the evaluation budget is exhausted. In the batch setting, the RL agent is trained independently for each candidate within the batch, so the overall training cost scales with the number of candidates evaluated per batch. See Appendix B for more training and testing details.

4 Experiment: RL-Based Mixed-Autonomy Traffic Control

Benchmark. We evaluate our method on an RL-based mixed-autonomy traffic control task studied in Cheng and Jin [4] in which both connected autonomous vehicles (CAVs) and human-driven vehicles coexist in the same environment. The CAV penetration ratio is set to 0.9. At each environment

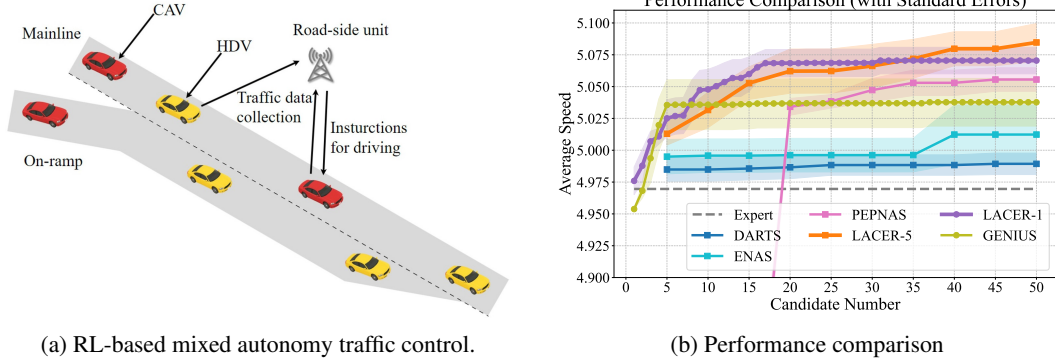


Figure 1: Left: Illustration of RL-based mixed-autonomy traffic control; Right: Comparison of performance (i.e., average traffic speed) between our two LACER variants and baselines.

step, the observation contains three distinct input sources: (i) the temporal traffic evolution of key metrics (e.g., speed, density, and flow rate), (ii) the current traffic state (lane-specific densities, speed distributions, and CAV penetration ratio), and (iii) the distribution history of the vehicle sequence. A schematic of this traffic control scenario is illustrated in Figure 1a. The presence of *multiple sources of inputs* makes this benchmark suitable for evaluating composite state encoders.

Baselines. We consider three groups of baselines: (i) *Expert-designed*, i.e., encoder architectures manually specified by a domain expert; (ii) *Traditional NAS*, including DARTS [7], ENAS [9], and PEPNAS [12], each generates 5 candidates per iteration; (iii) *LLM-based NAS*, including GENIUS [14], which uses GPT-4 to generate one candidate per iteration.

Evaluation metric. For each method, we track the *average traffic speed* achieved by the *best architecture evaluated so far*, and plot this task metric over the number of evaluated candidates. This allows us to assess the sample efficiency of the neural architecture search, i.e., how quickly each method discovers architectures with higher performance.

Experiment setup. Following Cheng and Jin [4], we adopt four encoder modules: a *traffic encoder*, a *time encoder*, a *sequence encoder*, and a *fusion encoder*. The traffic state is represented as a fixed-dimensional vector and processed by an FFN. The remaining two inputs (temporal traffic evolution and action sequence history) are treated as time-series, and we therefore search over transformer-based architectures for their corresponding encoders. The fusion module is also implemented as an FFN. The architecture search space for each module is designed following the taxonomy in the survey of Chitty-Venkata et al. [5]. Detailed module-specific search spaces are reported in Appendix B.

Each candidate architecture is trained for 200k interaction steps using PPO and then evaluated for 50k steps to obtain performance metrics, including average traffic speed, average reward, and feature information. We consider two variants of our method: LACER-1, which generates one candidate per iteration, and LACER-5, which generates five candidates per iteration. For a fair comparison, all methods are evaluated using 50 candidates in total—10 iterations for each batch method and 50 for others. To assess variability, each experiment is repeated with 8 random seeds.

Experiment results. Figure 1b shows the average traffic speed as a function of the number of evaluated architecture candidates. Both of our LACER variants significantly outperform the expert-designed architecture, traditional NAS baselines, and the LLM-based GENIUS baseline. These results demonstrate that combining LLM-based priors with composite state encoding and richer performance signals leads to more sample-efficient architecture search in RL settings.

In Appendix C, we also report ablation studies to analyze the effect of different design choices in our pipeline, including: (i) additionally providing the task metric (e.g., average traffic speed) of the initial expert-designed architecture; (ii) using only the task metric versus also including the average reward in the feedback; (iii) additionally providing the feature information (representation quality).

5 Conclusion and Future Directions

In this work, we proposed LACER, an LLM-driven composite NAS pipeline that automatically discovers effective state encoders for multi-source RL, achieving better task performance than traditional and LLM-based NAS baselines. In future work, we plan to apply LACER in broader applications such as goal-oriented tasks and robotic control with visual, textual and sensor inputs.

6 Acknowledgments and Funding Disclosure

This work was in part supported by the National Natural Science Foundation of China under Grant 62473250. The authors acknowledge the support of undergraduate researchers JunPing Li, Aghamatlab Akbarzade, Yuchen Jiang for their contributions to preliminary experimentation.

References

- [1] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. Botorch: A framework for efficient monte-carlo bayesian optimization. *Advances in neural information processing systems*, 33:21524–21538, 2020.
- [2] Zicheng Cai, Yaohua Tang, Yutao Lai, Hua Wang, Zhi Chen, and Hao Chen. Seki: Self-evolution and knowledge inspiration based neural architecture search via large language models. *arXiv preprint arXiv:2502.20422*, 2025.
- [3] Yuji Cao, Huan Zhao, Yuheng Cheng, Ting Shu, Yue Chen, Guolong Liu, Gaoqi Liang, Junhua Zhao, Jinyue Yan, and Yun Li. Survey on large language model-enhanced reinforcement learning: Concept, taxonomy, and methods. *IEEE Transactions on Neural Networks and Learning Systems*, 2024.
- [4] X. Cheng and L. Jin. Learning-based vehicle sequencing for on-ramp merging in mixed traffic. In *Proceedings of the 23rd IEEE International Conference on Industrial Informatics (INDIN)*, pages 0–0. IEEE, 2025.
- [5] Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, and Arun K Somani. Neural architecture search for transformers: A survey. *IEEE Access*, 10:108374–108412, 2022.
- [6] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*, pages 1437–1446. PMLR, 2018.
- [7] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.
- [8] Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. Llmatic: neural architecture search via large language models and quality diversity optimization. In *proceedings of the Genetic and Evolutionary Computation Conference*, pages 1110–1118, 2024.
- [9] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.
- [10] Sheila Schoepp, Masoud Jafaripour, Yingyue Cao, Tianpei Yang, Fatemeh Abdollahi, Shadan Golestan, Zahin Sufiyan, Osmar R Zaiane, and Matthew E Taylor. The evolving landscape of llm-and vlm-integrated reinforcement learning. *arXiv preprint arXiv:2502.15214*, 2025.
- [11] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10293–10301, 2021.
- [12] Yu Xue, Jiajie Zha, Danilo Pelusi, Peng Chen, Tao Luo, Liangli Zhen, Yan Wang, and Mohamed Wahib. Neural architecture search with progressive evaluation and sub-population preservation. *IEEE Transactions on Evolutionary Computation*, 2024.

- [13] Xue Yan, Yan Song, Xidong Feng, Mengyue Yang, Haifeng Zhang, Haitham Bou Ammar, and Jun Wang. Efficient reinforcement learning with large language model priors. *arXiv preprint arXiv:2410.07927*, 2024.
- [14] Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. Can gpt-4 perform neural architecture search? *arXiv preprint arXiv:2304.10970*, 2023.
- [15] Xun Zhou, Xingyu Wu, Liang Feng, Zhichao Lu, and Kay Chen Tan. Design principle transfer in neural architecture search via large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23000–23008, 2025.

A Methodology Illustration

Pipeline comparison. Compared to other LLM-based NAS methods (e.g., GENIUS in Figure 2), our approach is designed to enhance both sample efficiency and solution quality when searching for composite neural architectures for state encoders in RL.

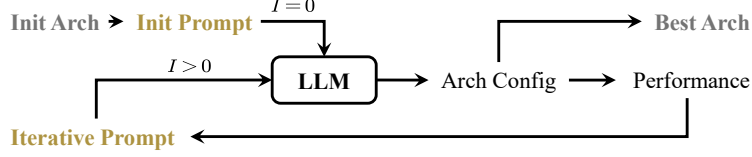


Figure 2: Pipeline of GENIUS, where I indicates the iteration count.

To achieve this, we incorporate evaluation metrics of the initial architecture into the initial prompt to provide richer prior context. Additionally, beyond standard task metric, we introduce two supplementary performance signals—*average reward* and *feature information* (i.e., representation quality)—as comprehensive feedback to the LLM, enabling iterative refinement of candidate architectures (Figure 3).

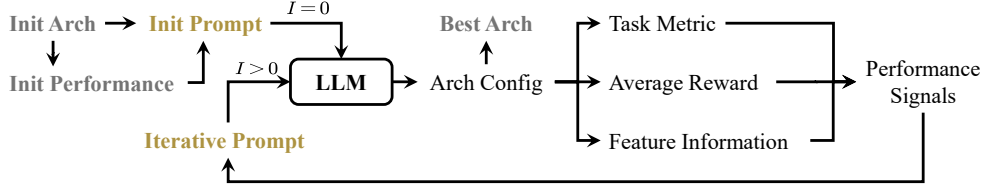


Figure 3: Pipeline of LACER (our method), where I indicates the iteration count.

Performance signals. Our method’s performance signals include three components: (i) task metric: average speed in the mixed-autonomy traffic control setting, served as the ultimate indicator of target task; (ii) average reward: incorporated as feedback given that the candidate architectures are for RL training, which characterizes key RL properties like convergence efficiency; (iii) feature information: quantified via mutual information (defined as $I(X; Y) = H(X) - H(X | Y)$ for random variables X and Y) and redundancy (defined as $R(X; Y) = H(X) + H(Y) - H(X, Y)$), served as a direct measure of representation quality for the composite state representation architecture. Specifically, we compute mutual information for feature pairs: time data before/after time encoder, traffic data before/after traffic encoder, sequence data before/after sequence encoder, and encoded features from time/traffic/sequence encoders with fused features after fusion encoder to comprehensively address the composite nature of the state encoder with multiple modules.

Prompt construction. The iterative prompt construction process used in our method is detailed in Algorithm 1. The conversation history \mathcal{H} is strategically pruned to retain only essential information—including initial architecture, performance signals, task description, and search space definition—while eliminating redundant and non-structural content. This reduces noise and mitigates potential LLM forgetfulness in long interactions. In the prompt, different roles are explicitly distinguished: the *assistant* role logs the LLM’s responses in \mathcal{H} , while the *system* and *user* roles provide setup and queries, respectively. The initial user prompt \mathcal{U}_0 is designed to activate the LLM’s prior knowledge through task description and structured search space modeling, while iterative prompts \mathcal{U}_i reinforce the search space and constraints to maintain robustness throughout generations.

B Experimental Setup and Implementation Details

B.1 RL-based mixed-autonomy traffic control

All experiments were run in parallel on various nodes of the cluster supported by the Center for High Performance Computing at Shanghai Jiao Tong University, which is equipped with Intel Xeon

Algorithm 1 Prompt Construction of LACER

Input: System prompt \mathcal{S} , Task description \mathcal{D} , Search space \mathcal{X} , Request \mathcal{R} , Initial architecture a_0 and its performance p_0 , Max iterations N

Output: Candidate architecture lists $\mathcal{L}_1, \dots, \mathcal{L}_N$

- 1: Initialize conversation history $\mathcal{H} \leftarrow \emptyset$
 - 2: $\mathcal{U}_0 \leftarrow \mathcal{D} + \mathcal{X} + a_0 + p_0 + \mathcal{R}$ {First iteration user prompt}
 - 3: $\text{Prompt}_0 \leftarrow \mathcal{S} + \mathcal{U}_0$
 - 4: $\mathcal{L}_1 \leftarrow \text{LLM}(\text{Prompt}_0)$
 - 5: $\vec{v}_{\text{raw}} \leftarrow \text{ParseLLMResponse}(\mathcal{L}_1)$ {Using Algorithm 2}
 - 6: Append *system* : \mathcal{S} , *user* : \mathcal{U}_0 , *assistant* : \mathcal{L}_1 to \mathcal{H}
 - 7: **for** $i = 1$ to $N - 1$ **do**
 - 8: $\mathcal{U}_i \leftarrow \mathcal{P}_{i-1} + \mathcal{X} + \mathcal{R}$ {Subsequent user prompts}
 - 9: $\text{Prompt}_i \leftarrow \mathcal{H} + \mathcal{U}_i$
 - 10: $\mathcal{L}_{i+1} \leftarrow \text{LLM}(\text{Prompt}_i)$
 - 11: $\vec{v}_{\text{raw}} \leftarrow \text{ParseLLMResponse}(\mathcal{L}_{i+1})$ {Using Algorithm 2}
 - 12: Append *user* : \mathcal{U}_i , *assistant* : \mathcal{L}_{i+1} to \mathcal{H}
 - 13: Train and evaluate each architecture in \mathcal{L}_{i+1} in RL framework
 - 14: **end for**
-

ICX Platinum 32-core CPUs and NVIDIA HGX A100 GPUs. Specifically, RL experiments were implemented within SUMO to simulate traffic environment. Each experiment is allocated with 512GB of memory, repeated with 8 random seeds and reported using mean with error bars, given by two times the standard error.

Composite architecture design of the RL state encoder. Following Cheng and Jin [4], the architecture of the state encoder includes a time encoder, traffic encoder, and sequence encoder for respective data encoding while the fusion encoder processes their concatenated outputs to generate the encoded state for RL training, as illustrated in Figure 4. All modules use Transformer (chosen over recurrent alternatives such as LSTMs, given its superior performance on sequential data) with FFNs. Time and sequence encoders add multi-head self-attention (MHSA) to capture their higher complexity, temporal variability, and dynamics, unlike preprocessed, macro-level, weakly temporal traffic data, and the fusion encoder which focused on integration without extra temporal processing uses only FFN.

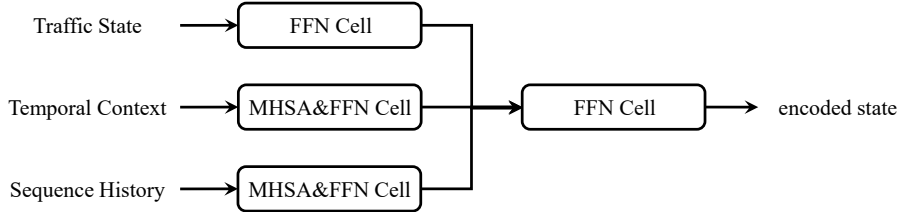


Figure 4: Composite architecture design of the RL state encoder for mixed-autonomy traffic control.

Module-specific search spaces. Based on the state encoder architecture with multiple modules of distinct functions and specific architecture types in Figure 4, we define module-specific search spaces, as detailed in Table 1. Following Chitty-Venkata et al. [5], key design choices commonly used in neural architecture search for Transformers are included: hidden layer dimension (denoted as "dimension" in the table), dimension expansion ratio ("ratio"), and number of neural network layers ("depth") for the FFN of each module. For modules with MHSA, the number of attention heads ("heads") is additionally included; for modules with only FFN, the type of activation function ("activation") is included instead. Common value ranges are set for all design choices, resulting in a total of approximately 26 million possible architectures throughout the search space.

Baseline alignment. Traditional NAS methods which we consider as baselines such as DARTS, PEPNAS, and ENAS were originally designed for computer vision tasks like image classification,

Table 1: Module-specific search spaces where bold values denote the configurations used by the Expert baseline.

Module	Operation	Heads / Activation	Dimension	Ratio	Depth
Time	MHSA, FFN	{ 2 , 4, 8}	{ 8 , 16, 32}	{1, 2 , 4}	{1, 2 , 3}
Traffic	FFN	{ relu , gelu, swish}	{16, 32 , 64}	{ 1 , 2, 4}	{ 1 , 2, 3}
Sequence	MHSA, FFN	{2, 4 , 8}	{8, 16 , 32}	{1, 2 , 4}	{1, 2 , 3}
Fusion	FFN	{ relu , gelu, swish}	{64, 128 , 256}	{ 1 , 2, 4}	{ 1 , 2, 3}

where performance is measured by accuracy and the concept of sample size is applicable. However, in RL scenarios, accuracy is absent, and the concept of sample size differs. Thus, when applying these methods to neural architecture search for state encoders in RL, certain corresponding mappings are required: (i) Performance: The accuracy used in ENAS and PEPNAS is replaced here by average speed; similarly, the gradient in DARTS, which reflects validation performance, is also mapped to average speed. (ii) Sample size: In PEPNAS, the sample size for validating candidates within each generation increases incrementally, which corresponds here to an incremental increase in training steps when validating candidates within each generation.

RL training and evaluation details. RL training requires sufficient steps for policy convergence, typically manifested by reward. In the mixed-autonomy traffic control settings simulated via SUMO, traffic flow arrives with a fixed periodic distribution, causing observed average vehicle speed to exhibit corresponding periodicity. Thus, RL policy evaluation also requires adequate steps to encompass multiple such cycles. To balance RL training convergence, evaluation comprehensiveness, and training cost, we analyzed the average reward and average speed (recorded every 1k steps over 1M steps of baseline RL training) as shown in Figure 5. The results indicate that the average reward converges around 200k steps, while the average speed exhibits a periodicity of approximately 25k steps. Based on this, each candidate state encoder architecture was evaluated by integrating it into the RL framework for 200k steps of training and 50k steps of evaluation.

Parsing of LLM response. To ensure automation of the LLM-based neural architecture search process, we adopt an algorithm to parse the structured natural language output from LLM into a vector of design choices, as presented in Algorithm 2. Firstly, the LLM is instructed to frame its architectural descriptions using a specific prefix (e.g., 'New Architecture'), which allows for the reliable extraction of the relevant text segment from its complete response. This segment is subsequently tokenized and parsed using a set of regular expression patterns that map directly to the design choices of the search space (e.g., heads, depth). The algorithm outputs the raw parsed values, which are then used directly to instantiate the state encoder for reinforcement learning.

Algorithm 2 Parse LLM Response to Architectures

Input: LLM response R , Prefix string P , Pattern set \mathcal{P} (regex patterns for each parameter)

Output: Raw parameter vector \vec{v}_{raw}

- 1: text_block \leftarrow ExtractTextAfterPrefix(R, P) {Get the structured output}
 - 2: tokens \leftarrow Tokenize(text_block) {Break into processable units}
 - 3: $\vec{v}_{\text{raw}} \leftarrow []$ {Initialize an empty list for design choices}
 - 4: **for each** pattern $p_i \in \mathcal{P}$ **do**
 - 5: $value \leftarrow$ ApplyRegex(p_i , tokens) {Match pattern against tokens}
 - 6: $\vec{v}_{\text{raw}}.append(value)$ {Append the parsed value}
 - 7: **end for**
 - 8: **return** \vec{v}_{raw}
-

LLM and temperature parameter selection. For the proposed LACER method (applied to NAS for RL state encoders), LLM type and temperature (a hyperparameter regulating LLM output randomness: higher values enhance diversity, lower values improve determinism) significantly affect task performance. To analyze the impact of design choices, such as the base LLM and its temperature, experiments were conducted on LACER using two representative LLMs (Claude Sonnet 4.0, GPT-4) under temperature configurations of 0.0 and 1.0. As shown in Figure 6, LACER achieved optimal

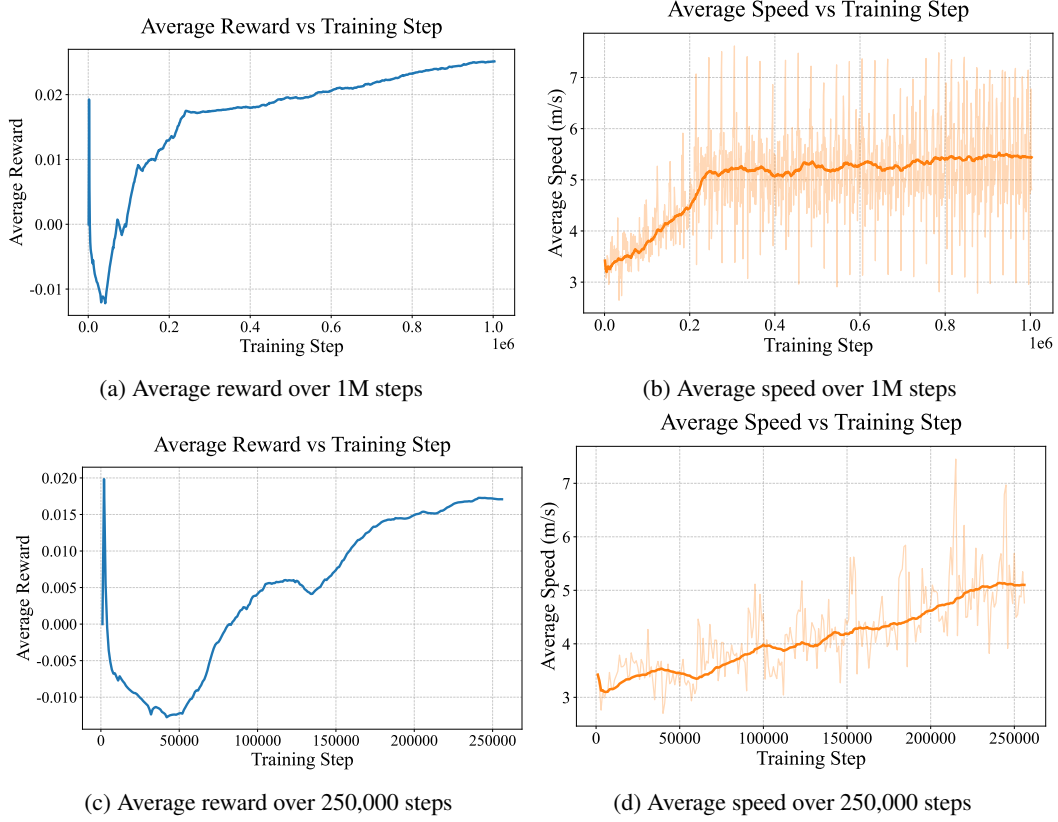


Figure 5: Average reward and average speed during RL training.

performance with Claude Sonnet 4.0 (temperature = 1.0), which was thus adopted in the main experiments.

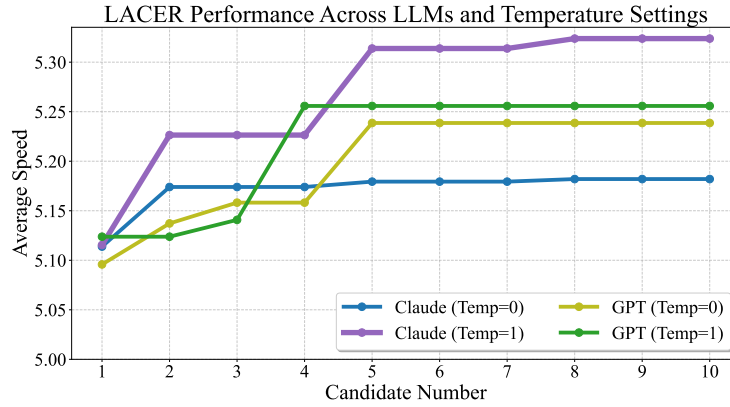


Figure 6: Performance of LACER-1 under different LLM models and temperature settings.

B.2 MiniGrid goal-oriented tasks

Architecture of the state encoder. Following Cheng and Jin [4], the architecture of the state encoder includes an image encoder and a text encoder for respective data encoding while the fusion encoder processes their concatenated outputs to generate the encoded state for RL training, as illustrated in Figure 7. The image encoder uses convolution neural networks (CNNs) added with a pooling layer to process image observation. The text encoder uses an embedding layer added with

a gate recurrent unit (GRU) to represent the text instruction. The fusion encoder concatenates the processed data of image encoder and text encoder to merge them into final representation as state.

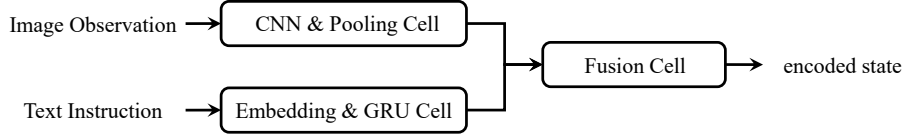


Figure 7: Composite Architecture of the state encoder in the MiniGrid goal-oriented tasks

Module-specific search spaces. Based on the state encoder architecture with multiple modules of distinct functions and specific architecture types in Figure 7, we define module-specific search spaces, as detailed in Table 2. Following Chitty-Venkata et al. [5], key hyperparameters commonly used in NAS for CNNs with pooling layers are included: the type of pooling method (denoted as "pooling type" in the table), the layer where the CNN is followed with the pooling layer ("pooling layer"), the type of activation function ("activation"), the size of kernels ("kernel size"), the number of channels ("channel number") and number of neural network layers ("depth") for the image encoder. Following Chitty-Venkata et al. [5], key parameters commonly used in NAS for RNNs with embedding layer are included: the dimension of embedding layer ("dimension"), the hidden size of RNNs ("hidden size"), the ratio of drop-out in RNNs ("dropout") and number of neural network layers ("depth") for the text encoder. Following Chitty-Venkata et al. [5], key variance of fusion method could be summarized as: the merge type ("merge type"), the dimension of fusion network ("dimension"), the type of activation function ("activation") and the hidden size of networks ("hidden size"). Common value ranges are set for all search parameters, resulting in a total of approximately 19 million possible architectures throughout the search space.

Table 2: Module-specific search spaces for the MiniGrid benchmark where bold values denote the configurations used by the Expert.

(a) Image Encoder (CNN & Pooling)	
Hyperparameter	Value Range
Pooling type	{ max , average}
Pooling layer	{ 1 , ..., depth}
Activation	{ sigmoid , tanh, relu, elu}
Kernel size	{1, 2 }
Channel number	{8, 16 , 32, 64}
Depth	{2, 3 , 4, 5}

(b) Text Encoder (Embedding & GRU)	
Hyperparameter	Value Range
Dimension	{16, 32 , 64, 128}
Hidden size	{64, 128 , 256, 512}
Dropout	{ 0.0 , 0.1, 0.2, 0.3}
Depth	{ 1 , 2, 3, 4}

(c) Fusion Encoder (Fusion Module)	
Hyperparameter	Value Range
Merge type	{ cat , add, gate, bilinear}
Dimension	{ 128 , 256, 512, 1024}
Activation	{ sigmoid , tanh, relu, elu}
Hidden size	{32, 64 , 128, 256}

Baseline alignment. Similar with Appendix B.1, when applying traditional NAS methods to NAS for state encoders in RL for goal-oriented tasks, certain corresponding mappings are required: (i) Performance: The accuracy used in ENAS and PEPNAS is replaced here by average return; similarly, the gradient in DARTS, which reflects validation performance, is also mapped to average return. (ii) Sample size: In PEPNAS, the sample size for validating candidates within each generation increases incrementally, which corresponds here to an incremental increase in training steps when validating candidates within each generation.

RL training and evaluation details. RL training requires sufficient steps for policy convergence, typically manifested by reward. In the MiniGrid goal-oriented tasks, agent uses reshaped return to train policy and requires several trials to finish the task under the trained policy. Thus, RL policy evaluation also requires adequate steps to encompass multiple such trials. To balance RL training convergence, evaluation comprehensiveness, and the cost of evaluating state encoder architectures, we analyzed the average reshaped return of training (recorded every 2,048 steps over 2M steps of baseline RL training) and average return of evaluation (recorded every step over 500 steps of baseline evaluation) as shown in Figure 8. The results indicate that the average reshaped return converges around 1M steps, while the average return converges around 100 steps. Based on this, each candidate state encoder architecture was evaluated by integrating it into the RL framework for 1M steps of training and 100 steps of evaluation.

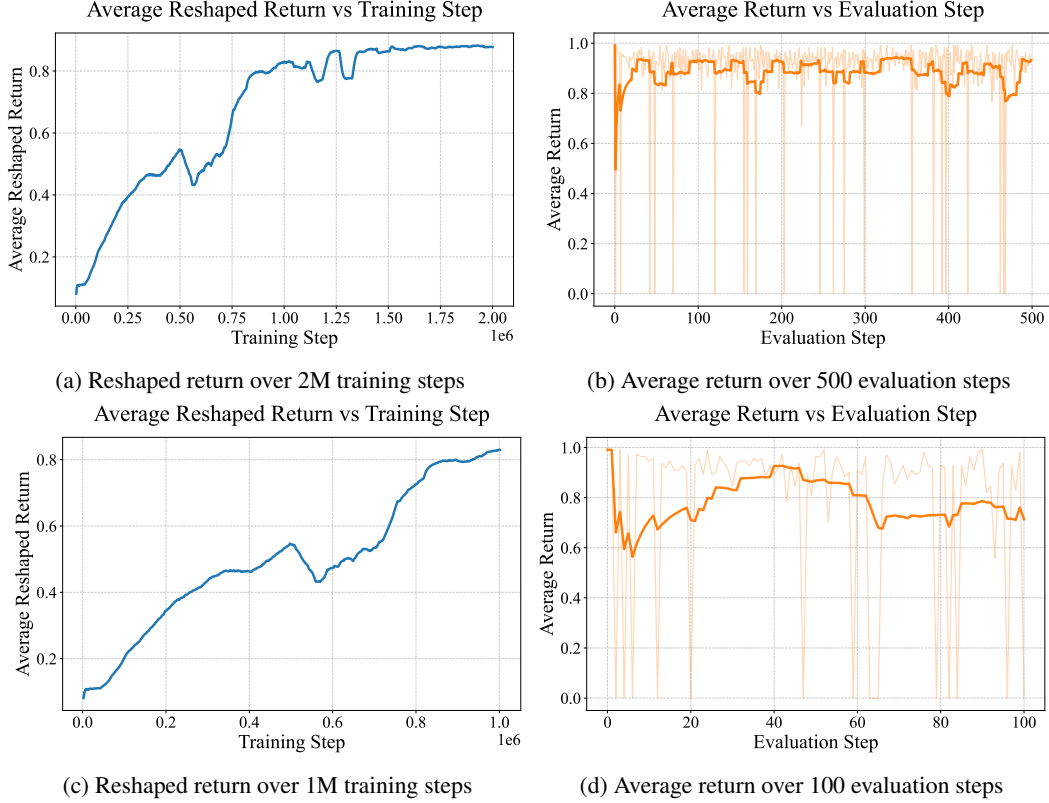


Figure 8: Reshaped return and during training and average return during evaluation in the MiniGrid goal-oriented tasks.

C Additional Experiment Results

Ablation studies To verify the necessity of each core module in the proposed LACER method, we designed a series of ablation experiments, in which three key prompt components were removed, respectively: the feature information (FI), the average reward (RI), and the initial architecture evaluation (IE). This resulted in three variant methods: LACER-1 without FI, LACER-1 without FI + RI and LACER-1 without FI + RI + IE. The results in Fig. 9 indicate that the original LACER-1 method achieves the best performance. When any of the components mentioned above is removed, the performance of LACER deteriorates significantly. This phenomenon demonstrates that each of these components in our LACER method is useful and essential, and their collaborative operation contributes to the superior performance of the proposed method.

Time cost The time cost of traditional NAS methods is composed of the evaluation time of each candidate and the search time of each generation, while the search cost of LLM-based NAS methods includes the query time of LLM and the processing time of analyzing results of candidates and

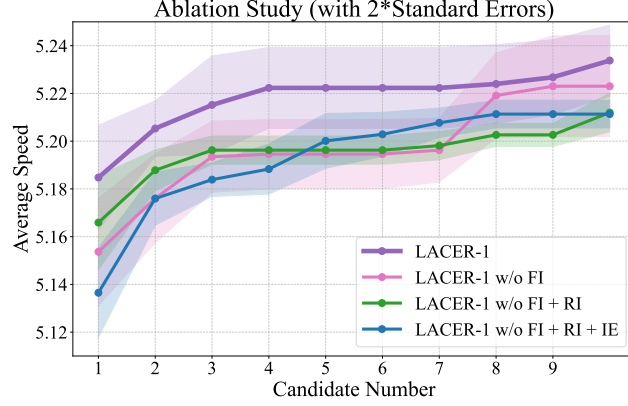


Figure 9: Performance comparison of LACER-1 with and without different key prompt components.

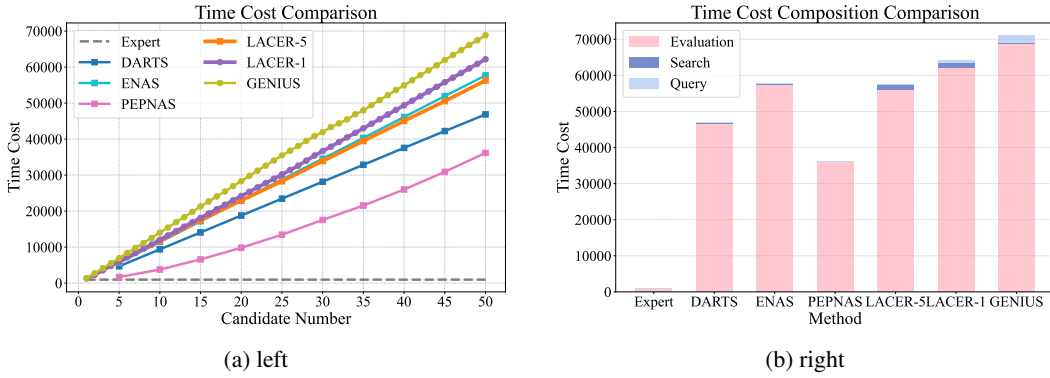


Figure 10: Left: Time cost comparison between our two LACER variants and baselines; Right: Time cost composition comparison between our two LACER variants and baselines. The query time accounts for 1% of the overall time cost of LACER, while the evaluation time accounts for over 97% of the overall time cost of all methods.

constructing prompts. To analyze the impact of query time, we run LACER and other baselines on an Intel Core i7 CPU and an NVIDIA RTX 4070 GPU to avoid the time cost of node scheduling. We then compare the time cost of the above compositions as shown in Figure 10, where evaluation refers to the evaluation time, search refers to the search time that does not include query time, and query refers to the query time. The result indicates that the query time of LLM has a negligible impact on the overall time cost compared to the evaluation time.