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

Yu Yu
Shanghai Jiao Tong University

Qian Xie*
Cornell University

Li Jin
Shanghai Jiao Tong University

Abstract

Designing state encoders for reinforcement learning (RL) with multiple input sources—such as sensor measurements and time-series signals—is challenging and typically requires manual effort. Neural architecture search (NAS) can automate this process, but applying NAS in RL is expensive due to slow environment interactions. We introduce an automatic LLM-driven NAS pipeline that leverages large language models to generate effective composite neural architectures for RL state encoding. The pipeline outputs specialized modules for each input source along with a fusion module and evaluates them end-to-end. On a mixed-autonomy traffic control task, our pipeline discovers architectures with better performance using fewer evaluations than traditional NAS baselines and LLM-based GENIUS, demonstrating the value of LLM priors for sample-efficient search.

1 Introduction

Reinforcement learning (RL) often requires transforming raw observations into latent state representations. In many real-world settings, the state is observed from diverse input sources—such as sensor measurements, time-series signals, image observations, or textual instructions—and therefore requires different modules to encode each source. Existing RL systems typically rely on *manually* and *empirically* designed encoders for this purpose, which can be suboptimal and difficult to generalize across domains. A natural alternative is to use neural architecture search (NAS) to automatically discover effective encoder architectures. However, in contrast to standard supervised learning tasks such as image classification, training and evaluating encoder architectures in RL can be time-consuming due to repeated interaction with a simulator or environment. Such architectures often require thousands of episodes to converge and make sample efficiency a key challenge for NAS in the RL setting.

At the same time, large language models (LLMs) have demonstrated remarkable capabilities in representing and retrieving domain knowledge. This raises an exciting question: *can LLMs provide useful priors to guide NAS for RL state encoding, thereby reducing the search cost and improving the final RL performance?*

To this end, we propose an LLM-driven NAS pipeline that leverages language-model priors to automatically discover effective composite state encoder architectures for RL. Our contributions are:

1. We propose an LLM-driven NAS pipeline that automatically designs RL state encoders consisting of specialized modules for each input source and a fusion module.
2. We instantiate and evaluate the generated state encoders on a representative RL task, viz. mixed-autonomy traffic control.
3. We demonstrate that language models can express informative priors over composite state encoder architectures through prompting, without requiring any fine-tuning.

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

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

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 and Bayesian optimization approaches, including Gaussian process–based methods such as those implemented in BoTorch [1] with mixed-type 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, **L**LM-driven Neural **A**rchitecture Search for **C**omposite State **E**ncoders in **R**L (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) to extract relevant features. Instead of searching for a single shared encoder, our

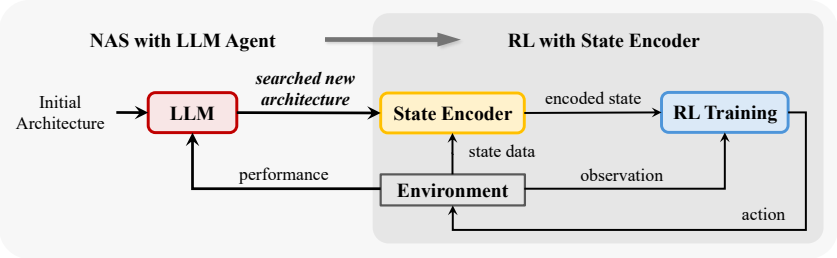


Figure 1: LLM-agentic prompting → architecture generation → RL training & evaluation pipeline

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, we denote the raw observation by $x = (x_1, \ldots, x_M)$ where each x_i corresponds to one of the M distinct input sources. Define the overall state encoder as $s = g_\phi\big(f_{\theta_1}(x_1), \ldots, f_{\theta_M}(x_M)\big)$, where f_{θ_i} is the architecture for source i and g_ϕ is the fusion module. Let $\mathcal{E}$ denote the RL environment. Our objective is to select architectures $\{f_{\theta_i}\}_{i=1}^{M}$ and g_ϕ such that the downstream task metric

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

is maximized, where $\mathcal{M}$ denotes a task-specific performance metric (e.g., average traffic speed), which may differ from the reward used for training the policy π within $\mathcal{E}$.

3.2 LLM-Driven Neural Architecture Generation

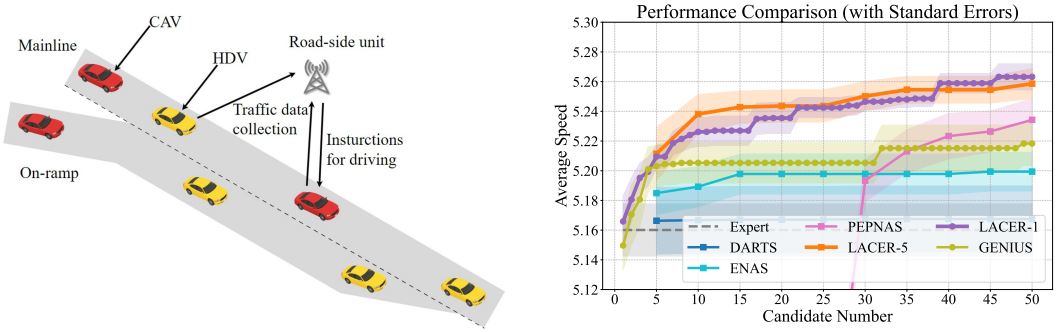
We begin with an expert-designed initial architecture. At each subsequent iteration, we query an LLM with a textual prompt that summarizes the current set of architecture modules and their associated performance. The LLM then responds with *one or a batch of new composite architecture candidates*. For each module (including the input-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, transformers for time-series inputs, and FFNs for vector inputs). Figure 5 and Table 1 in Appendix B provide an example of a composite architecture with transformers and FFNs, along with the corresponding search space. 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, see Appendix A.

3.3 RL Training and Evaluation

Each generated composite architecture is trained end-to-end 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 (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 for a total of N iterations (see Appendix B for training and testing details). 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 iteration.

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 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 2a. The presence of *multiple sources of inputs* makes this benchmark suitable for evaluating composite state encoders.



(a) left (b) right

Figure 2: Left: RL-based mixed-autonomy traffic control; Right: Comparison of performance (i.e., average traffic speed) between our two LACER variants and baselines.

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 2b 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 feature-based evaluation information.

5 Conclusion and Future Directions

In this work, we proposed LACER, an LLM-driven composite NAS pipeline that leverages language model-prior to automatically discover effective state encoders for 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 robotics with visual, textual and sensor inputs.

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 3), our approach is designed to enhance both sample efficiency and solution quality when searching for composite neural architectures for state encoders in RL.

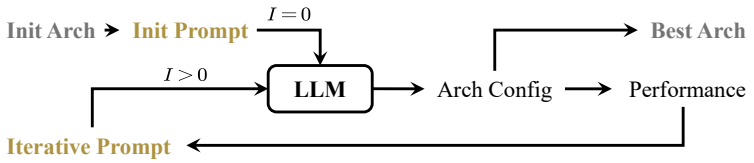


Figure 3: Pipeline of GENIUS, where I indicates the iteration count.

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*—as comprehensive feedback to the LLM, enabling iterative refinement of candidate architectures (Figure 4).

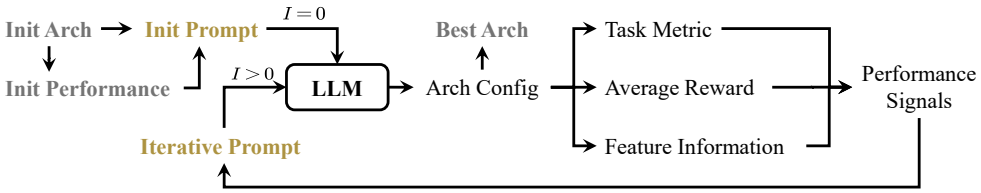


Figure 4: Pipeline of LACER (our method), where I indicates the iteration count.

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 $I(X;Y) = H(X) - H(X \mid Y)$ 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

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 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

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 $\mathcal{L}_1, \ldots, \mathcal{L}_N$

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

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 5. 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.

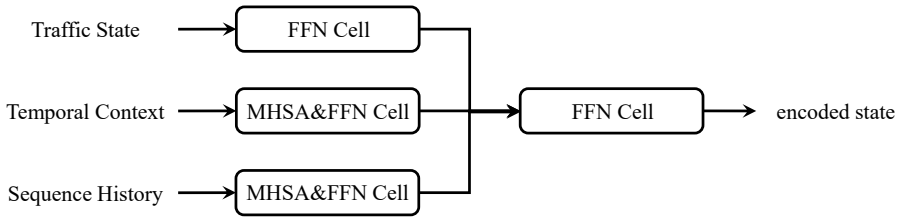


Figure 5: Composite architecture design of the RL state encoder for mixed-autonomy traffic control.

Module-specific search spaces. Based on the state encoder architecture with multiple modules of distinct functions and specific architecture types in Figure 5, we define module-specific search spaces, as detailed in Table 1. Following [5], key parameters 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 search parameters, 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, 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

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	$\{\mathbf{2}, 4, 8\}$	$\{\mathbf{8}, 16, 32\}$	$\{1, \mathbf{2}, 4\}$	$\{1, \mathbf{2}, 3\}$
Traffic	FFN	{**relu**, gelu, swish}	$\{16, \mathbf{32}, 64\}$	$\{\mathbf{1}, 2, 4\}$	$\{\mathbf{1}, 2, 3\}$
Sequence	MHSA, FFN	$\{2, \mathbf{4}, 8\}$	$\{8, \mathbf{16}, 32\}$	$\{1, \mathbf{2}, 4\}$	$\{1, \mathbf{2}, 3\}$
Fusion	FFN	{**relu**, gelu, swish}	$\{64, \mathbf{128}, 256\}$	$\{\mathbf{1}, 2, 4\}$	$\{\mathbf{1}, 2, 3\}$

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 the cost of evaluating state encoder architectures, we analyzed the average reward and average speed (recorded every 1,000 steps over 1,000,000 steps of baseline RL training) as shown in Figure 6. The results indicate that the average reward converges around 200,000 steps, while the average speed exhibits a periodicity of approximately 25,000 steps. Based on this, each candidate state encoder architecture was evaluated by integrating it into the RL framework for 200,000 steps of training and 50,000 steps of evaluation.

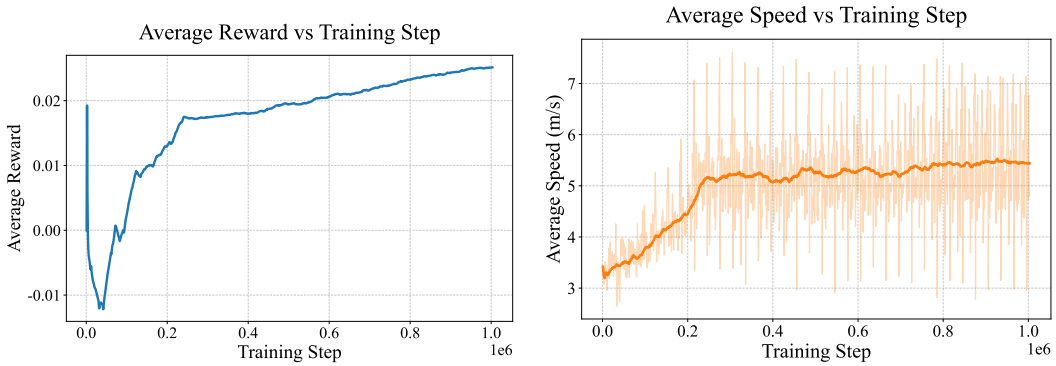
Parsing of LLM response. To ensure automation of the LLM-based neural architecture search process, we adopt an algorithm to parse the structured natural language output from LLM into a vector of architectural parameters, as presented in Algorithm 2. Firstly, the LLM is instructed to frame its architectural 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 parameters 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 $\vec{v}_{\text{raw}}$

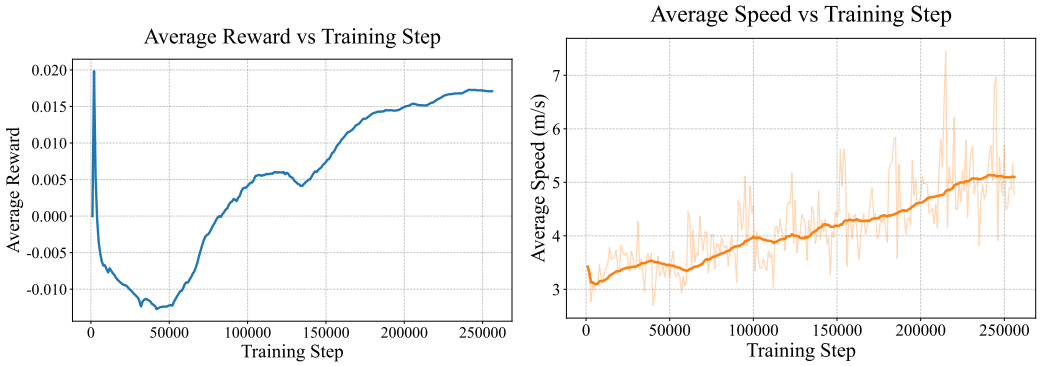
```
text_block ← ExtractTextAfterPrefix(R, P) {Get the structured output}
tokens ← Tokenize(text_block) {Break into processable units}
v_raw ← [ ] {Initialize an empty list for parameters}
for each pattern p_i ∈ P do
    value ← ApplyRegex(p_i, tokens) {Match pattern against tokens}
    v_raw.append(value) {Append the parsed value}
end for
return 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 7, LACER achieved optimal



(a) Average reward over 1,000,000 steps

(b) Average speed over 1,000,000 steps



(c) Average reward over 250,000 steps

(d) Average speed over 250,000 steps

Figure 6: 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.

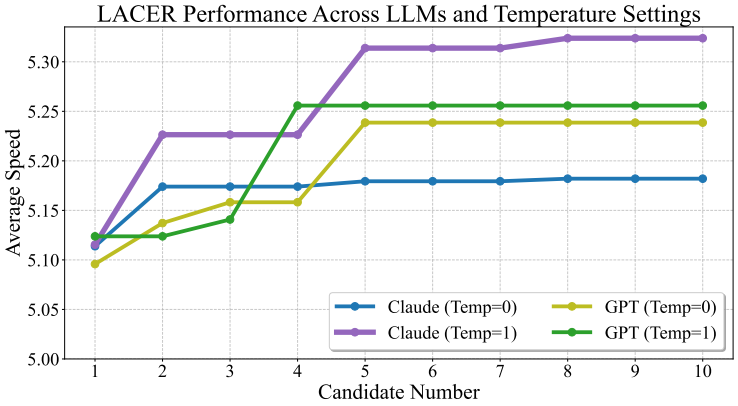


Figure 7: Performance of LACER-1 under different LLM models and temperature settings.

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. 8 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.

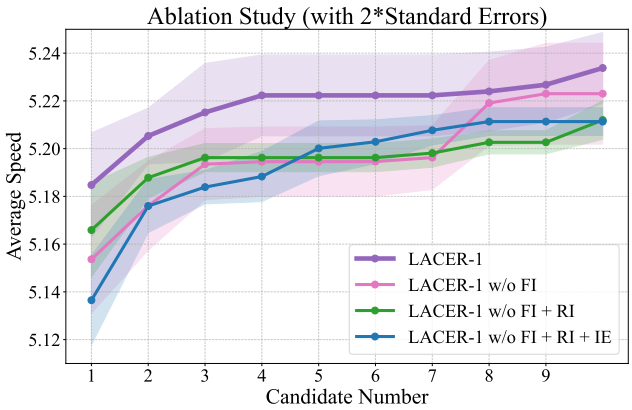


Figure 8: Performance comparison of LACER with and without different key prompt components.