Automatic Generation of Safety-compliant Linear Temporal Logic via Large Language Model: A Self-supervised Framework

Junle Li[1], Siqi Chen[1], Jiakai Li[1], Meiqi Tian[1], Bingzhuo Zhong[1*]
[1]*Hong Kong University of Science and Technology (Guangzhou)*, Guangzhou, China
Email: {juliejunleli, bingzhuoz}@hkust-gz.edu.cn, {schen972, jli196, mtian837}@connect.hkust-gz.edu.cn

***Abstract*—Converting high-level natural-language task descriptions into formal specifications such as Linear Temporal Logic (LTL) is essential for ensuring safety in cyber-physical systems (CPS). Existing work, however, *only* optimizes translation quality without explicitly verifying the output against safety constraints. We present `AutoSafeLTL`, a self-supervised, cloud–edge–collaborative framework that automates the generation of safety-compliant LTL specifications while preserving logical consistency and semantic fidelity. A lightweight edge-side three-stage-fine-tuned LLM offers real-time conversion from natural language to LTL specifications (NL2LTL) and *guarantees* safety-critical latency and data locality. Two larger-capacity cloud-side agents then iteratively refine the alignment: 1) *LLM-as-an-Aligner* matches atomic propositions to safety constraints, and 2) *LLM-as-a-Critic* interprets counterexamples from Inclusion Check to guide corrective regeneration. This collaborative architecture provides a safety-guaranteed alignment mechanism between high-level user intent and formally verifiable system behavior, demonstrating the potential of our framework to advance AI Alignment in safety-critical domains. Our approach achieves 0% violation rates on multiple benchmarks, enabling trustworthy specification generation and verification for both AI and critical CPS applications.**

***Index Terms*—Linear Temporal Logic, Large Language Models.**

I. Introduction

Motivation: Ensuring the safety of cyber-physical systems (CPS) is central to modern system design, especially in safety-critical domains. Formal specifications like Linear Temporal Logic (LTL) [1] provide mathematical rigor for verification and controller synthesis, ensuring CPS safety. The challenge lies in translating high-level, often unstructured requirements into formal specifications. However, a critical issue frequently neglected by these methods is whether the generated LTL formula inherently conflict with the safety constraints imposed on the system. We refer to this aspect as the *compliance of LTL specifications* with respect to safety constraints, and to those LTL specifications with no such conflict as *safety-compliant LTL*. Specifically, being safety-compliant means that the temporal behavior expressed by the LTL formula entirely aligns with the intended temporal behavior mandated by safety constraints. Accordingly, we focus on how to generate *safety-compliant LTL* specifications via Large Language Models (LLMs), while maintaining formal safety-compliance guarantees to prevent critical risks to CPS.

*Corresponding author.

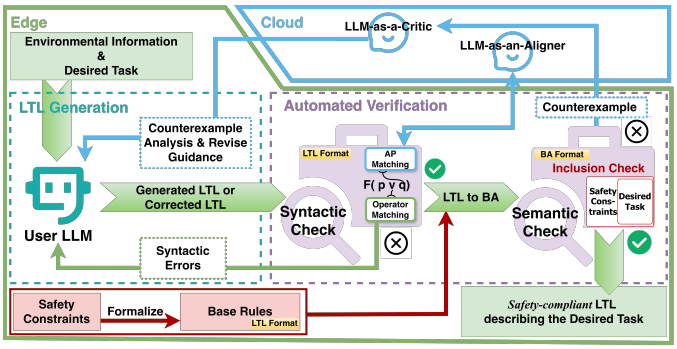

Fig. 1. Overview of `AutoSafeLTL` Framework.

Related Works: Linear Temporal Logic (LTL) is fundamental for formal verification in autonomous systems. Recent NL2LTL approaches improve translation efficiency and syntactic validity via decomposition, dynamic prompting, grammar-constrained decoding, and fine-tuning [2]. However, these methods focus exclusively on *semantic alignment*. By treating user instructions as the absolute truth, they ignore *external safety constraints*, risking the faithful translation of unsafe intents into dangerous specifications. Meanwhile, frameworks integrating LLMs with formal verifiers typically position the verifier as a *post-hoc filter* for LLM-generated plans. This remedial paradigm leads to inefficient "trial-and-error" cycles and only verifies one-off execution traces rather than the underlying task logic, lacking a persistent safety foundation against environmental variations.

In this work, we present `AutoSafeLTL` (as shown in Fig. 1), a self-supervised framework that integrates automata-based verification with a cloud–edge collaborative LLM architecture to generate *safety-compliant* LTL specifications in a fully-automated manner. An edge-deployed *User LLM*, fine-tuned in three progressive stages (translation, syntactic, and semantic correction), handles both real-time LTL generation and local repair with minimal latency. For rigorous verification, cloud-side agents—*LLM-as-an-Aligner* and *LLM-as-a-Critic*—are invoked to interpret complex violations detected via Inclusion Checks. This collaborative synergy ensures data locality while leveraging cloud-scale inference to achieve formal alignment

between user intent and safety properties. Our contributions are summarized as follows:

- We propose a *fully automated*, end-to-end framework for NL-driven LTL specification generation that targets *early-stage safety-compliant* specification generation. By shifting the focus to the generation of the specification itself, our framework ensures the governing logic is formally verified before system execution. We enhance formal verification tools with an efficient counterexample path extraction algorithm, enabling *automated* LTL refinement and a 0% violation rate against safety constraints.
- We develop a novel three-stage fine-tuning strategy that progressively instills *intrinsic safety* capability into a small language model. This strategy enables an 8B edge model to outperform GPT-4 in generating safety-compliant LTL specifications, proving the viability of formal methods on resource-restricted hardware.
- We establish a cloud–edge collaborative framework that enforces data locality through on-device LTL generation and correction. By delegating abstracted tasks—AP alignment and counterexample interpretation—to cloud-side agents, this architecture enables the scalable deployment of formal methods in safety-critical and privacy-sensitive domains.
- We construct fine-tuning and evaluation datasets to address the lack of suitable datasets for evaluating safety-compliance in NL2LTL tasks. Specifically, our datasets comprise over 35k synthetic LTL specifications for pre-training, 200 carefully curated human–LTL pairs for fine-tuning to enhance the model's correction ability, and 300 evaluation cases, featuring navigation-style instructions with human-centered APs and rich temporal structures, enabling safety-realistic evaluation.

II. Preliminary

For every LTL formula φ, there exists a corresponding Büchi automaton $\mathcal{A}$ such that: $L(\mathcal{A}) = Words(\varphi)$. This conversion enables BA-based modeling of dynamic system behavior, facilitating formal reasoning and verification.

A. Automata-Based Language Inclusion Checking

The results in [3] show that safety constraints could also be interpreted as LTL (therefore BA). Therefore, the compliance of LTL specifications can be converted to an Automata-based Inclusion Check [4].

Definition 1 (Language Inclusion): Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two Büchi automata over the same input alphabet Σ. If every infinite word accepted by $\mathcal{A}_1$ is also accepted by $\mathcal{A}_2$, then the language of $\mathcal{A}_1$ is *included* in the language of $\mathcal{A}_2$, denoted as $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$:

$$L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2) \iff \forall w \in \Sigma^{\omega},\ w \in L(\mathcal{A}_1) \Rightarrow w \in L(\mathcal{A}_2) \tag{1}$$

This foundation enables rigorous verification of LTL specification compliance with safety constraints.

B. Problem Formulation

Let $\mathcal{D}$ be an unstructured natural language description of a desired task, and let $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$ be a set of predefined safety constraints, each expressed as an LTL formula. Design a framework that synthesizes an LTL formula φ based on $\mathcal{D}$, such that: $L(\mathcal{A}_\varphi) \subseteq \bigcap_{i=1}^{n} L(\mathcal{A}_{\psi_i})$, where $\mathcal{A}_\varphi$ and $\mathcal{A}_{\psi_i}$ are the Büchi automata corresponding to φ and ψ_i, respectively. The objective is to generate an LTL specification φ according to $\mathcal{D}$ which is guaranteed to satisfy all the safety constraints in Ψ.

III. Framework

A. Overview

We propose `AutoSafeLTL` (Fig. 1), comprising two collaborative modules: *LTL Generation* and *Automated Verification*. Given *Desired Tasks* and *Environmental Information* in natural language, a lightweight edge-side *User LLM*—fine-tuned in three stages —generates and locally revises candidate LTL specifications.

The Automated Verification process consists of two stages: *Syntactic Check* and *Semantic Check*. Syntactic Check ensures grammatical validity. After passing the Syntactic Check, the Semantic Check ensures that the properties of state paths expressed by the candidate LTL specification adhere to safety constraints. In particular, both the Desired Task in LTL-format and the formalized *safety constraints* are converted into BA format for the Inclusion Check.

The principles of feedback and regeneration further ensure safety-compliance. Two cloud-side *Agent LLMs* iteratively refine the LTL specifications: 1) *LLM-as-an-Aligner* ensures atomic proposition alignment; and 2) *LLM-as-a-Critic* guides corrective regeneration using Language Inclusion counterexamples. This cloud-edge synergy balances data locality and rigorous safety verification, highlighting a practical path towards AI alignment for safety-critical CPS applications. For a better illustration of the framework, we used the following running example throughout the entire session.

Running example Consider a traffic scenario, an NL navigation task along with a related safety constraint. Our goal is to automatically generate an LTL specification that represents the desired task while adhering to the safety constraint. This specification can then support downstream applications such as path planning.

Desired Task: Start from the current lane, go straight for 200m. Then, turn right onto Maple St. and proceed for 500m. Finally, turn left onto Oak St., and you will arrive at the destination after 300m.

Env. Info: Elm St. (Straight lane, 50 km/h limit); Car overtaking on right; Target: Oak St. (1 km away).

Safety Constraint: $G(($straight_500m $\vee$ right_turn$) \rightarrow ($right_turn $\rightarrow$ straight_500m $\rightarrow$ left_turn$)) \rightarrow G($straight_1km $\rightarrow$ arrive_destination$)$.

Remark 1. While the running example and the evaluation part in Section V focus on a traffic scenario for illustration, our framework allows for flexible addition and adaptation of desired tasks and specifications as needed.

Remark 2. While Environmental Information is not required to generate the initial LTL formula, it is essential for LLMs to *regenerate* safety-compliant specifications when violations occur, especially when the original task description is inherently unsafe.

B. LTL Generation

The LTL Generation module performs the initial NL2LTL translation from task descriptions and auxiliary environmental information on the edge. We deploy a lightweight *User LLM* that has been sequentially fine-tuned in three stages—NL2LTL translation (S1), syntactic correction (S2), and semantic correction (S3)—so that a single on-device model jointly handles generation and subsequent local repair, satisfying the CPS-driven requirement: data locality for privacy, or regulation-sensitive deployments.

Given an NL input $\mathcal{D}$ and context E, *User LLM* extracts an initial LTL $\varphi^{(0)}$. This output is then passed to the Automated Verification module. Fine-tuning details are illustrated in Section IV.

C. Automated Verification

Verification ensures that the generated LTL specification satisfies safety constraints. Our Automated Verification scheme, the core of the framework, comprises a *Syntactic Check* and a *Semantic Check*.

1) Syntactic Check: The syntactic check consists of two steps: Atomic Proposition (AP) Matching and Operator Matching. For AP Matching, APs are extracted from safety constraints to build a library. The cloud-side *LLM-as-an-Aligner* aligns APs in the Desired Task with entries in this library and replaces them with the closest matches to ensure AP consistency. For Operator Matching, a token-based algorithm checks operator usage and parenthesis balance. Any detected error (along with its type and position) is fed to *User LLM* for immediate on-device correction.

Now, we revisit the running example to demonstrate the syntactic check.

Example (continued) Having the safety constraint in LTL-format, we extract the AP library as [`right_turn`, `straight_500m`,`left_turn`,`straight_1km`, `arrive_destination`]. Then, *LLM-as-an-Aligner* is invoked to compare the LTL-formatted Desired Tasks with the AP Library. Accordingly, `go_straight_500m` in the original LTL formula is replaced with `straight_500m` from the AP Library due to their similarity, and `go_straight_200m` is replaced with the format-similar AP `straight_200m`. All other parts in the LTL formula remain unchanged, which results in a new LTL formula as follows:

$$F(\text{straight\_200m}) \wedge X\Big(G\Big(\text{right\_turn\_Maple\_St} \rightarrow F\big(\text{straight\_500m} \wedge X(\text{left\_turn\_Oak\_St} \wedge F(\text{straight\_300m}))\big)\Big)\Big) \tag{2}$$

In the Operator Matching step, both are correctly matched. The resulting LTL then undergoes the Inclusion Check. *

2) Inclusion Check: After the syntactic check, the LTL formulas for the Desired Task and safety constraints are converted into BA format `Input_BA` and `Comparison_BA` for the Inclusion Check. We adopt `Spot` [5] for LTL2BA translation and `RABIT` [6] for Inclusion Check. Rather than simply integrating these formal tools, we adapt and enhance them to meet specific application objectives, while keeping the pipeline compatible with alternative solutions.

Remark 3. In practice, multiple safety constraints may exist as separate LTL formulas. Our framework verifies each formula in parallel, supports adding new rules dynamically, and allows prioritizing checks by importance to provide different levels of safety assurance. To provide more intuition for the Inclusion Check, we now revisit the running example.

Example (continued) Both the Safety Constraint and Desired Task are converted into Büchi automata and then subjected to a Ramsey-based Inclusion Check. The process terminates once a counterexample is found, which in this case is `!straight_200m`. This counterexample guides the subsequent LTL correction.

3) Counterexample-guided Automatic Correction: In Language Inclusion Check, a single counterexample word σ can be found if there exist any violations, but σ may correspond to multiple dynamic behaviors, as the same symbol $\sigma_i \in \sigma$ can trigger different transitions depending on the current state of the automata. Consequently, identifying a concrete counterexample path - a sequence of state transitions combined with the driving symbol - is essential for accurately locating the source of the violation. This path serves as a precise semantic witness for non-inclusion and provides actionable guidance for correcting the LTL specification. To this end, we formally define the notion of a counterexample path and establish its existence Theorem. The mathematical definition of this path, the proof of its existence, and the detailed extraction algorithm are provided in the extended arXiv version [7]. The formal proposition demonstrates the usability of this theoretical foundation, which constitutes a key contribution of our work.

To maximize the utility of counterexample paths, we introduce *LLM-as-a-Critic*. We discovered that enabling one LLM to understand another LLM can overcome the LLM's lack of knowledge in formal verification and format conversion. First, we let *LLM-as-a-Critic* directly analyze the counterexample path and provide suggestions for modifying the input LTL specification. Then, we pass the output of *LLM-as-a-Critic* to *User LLM*, which has access to Environmental Information, to perform the actual LTL modifications. This process not only enhances the extraction of counterexample information but also allows for more effective modifications by incorporating Environmental Information (cf. Section V-D for the ablation experiment).

To illustrate this process, we revisit the running example.

Example (continued) First, we input the counterexample path into the *LLM-as-a-Critic*. The obtained violation analysis and correct guidance are then be fed to the User LLM to modify the LTL specification. Although the LTL still fails the Inclusion Check, after iterating the automated verification loop three times, the final LTL that adheres to the rules is obtained as follows:

$$\varphi = X(\text{destinationLeftOnMapleStreet}) \wedge G\big(\text{location\_start} \rightarrow (F(\text{str\_200m}) \vee G(\neg \text{str\_200m}))\big) \wedge X\Big(G\big(\text{right\_turn\_Maple\_St} \rightarrow F(\text{straight\_500m} \wedge X(\text{left\_turn\_Oak\_St} \wedge F(\text{straight\_300m})))\big)\Big) \quad (3)$$

Our framework prioritizes safety-compliance over strict semantic fidelity to the Desired Task: when the original route violates safety, it is reordered with a safe alternative that still reaches the same destination. In this example, the initial LTL formula already satisfies the latter part of the Safety Constraint, so the segment from "turn right onto Maple Street" remains unchanged. And because the Safety Constraint requires the journey to begin by proceeding straight or making a right turn, we add the `straight_200m` restriction. It also stipulates that the destination is reached after a left turn followed by straight driving. Using Environmental Information, we identify that the last left turn occurs after entering Maple Street and therefore add $X(\text{destinationLeftOnMapleStreet})$ to ensure correct arrival. $*$

Remark 4. Our ideal objective is to search for an LTL specification that satisfies all safety constraints while making the minimal semantic modification to the Desired Task. In practice, we set a maximum number of repair rounds and a timeout to prevent infinite correction loop for the Desired Task over which a safety-compliant case can't be found.

Once the Desired Task in LTL format passes the semantic check, a safety-compliant LTL $\varphi^{(k)}$ after k rounds with respect to predefined safety constraint is obtained. Note that based on our framework's mechanism and the Inclusion Check's mathematical rigor, it is guaranteed that *any LTL specification that is not safety-compliant will not be output*, which is critical for its involvement in subsequent applications. This would also be shown by the experiment in Section V-D.

IV. Finetuning Approach

A. Construction of Datasets

A major limitation of existing NL2LTL datasets is their abstract APs lacking practical context. We emphasize human-centered APs to aid counterexample-driven correction. Since our goal is to fine-tune a lightweight LLM, the dataset must be compact yet effective.

We chose the traffic navigation as the primary scenario. To support the three-stage-fine-tuning process of our framework, we construct dedicated datasets for each stage, following a generate–filter–refine paradigm that ensures syntactic correctness, semantic alignment, and safety-compliance, respectively.

1) For stage-1, 50 navigation instruction templates were expanded via GPT-4 into NL-LTL pairs. Generated formulas were validated with SPOT [5], retaining only syntactically correct ones. We first perform a manual semantic check to ensure each natural language instruction accurately reflects the intent of its corresponding LTL formula. Then, we standardize and refine the wording of all instructions using a simplified prompt template, making their style consistent with downstream applications. The final result is a set of 200 high-quality Instruction-LTL training pairs.

2) For stage-2, erroneous LTL specifications from generation were annotated with error types and paired with corrected versions passing SPOT, forming 225 Instruction–Corrected LTL pairs across 69 error types.

3) For stage-3, using RABIT [6], we curated 50 triples (Instruction-Counterexample-Corrected LTL) from cases that initially failed Inclusion Checks. These rare and time-consuming cases make the collection especially valuable for training; we demonstrate its effectiveness in Section IV-C.

The Stage-1 dataset contains formulas with up to 8 nesting depths, with 99.0% conjunctions and 39.5% until operators. Detailed dataset statistics are provided in our arXiv version [7]. For evaluation, we construct a set of 300 naturalistic navigation tasks featuring diverse logical structures and previously unseen combinations of atomic propositions. This dataset is completely held out from all training stages and serves as the evaluation benchmark for all LTL specification generation tasks in our work.

B. Model Fine-tuning

We employ LLaMA-3-8B-Instruct as the edge-side *User LLM*, fine-tuned using 4-bit QLoRA to ensure on-device efficiency. This parameter-efficient strategy enables the incremental injection of repair knowledge across stages while avoiding catastrophic drift.

C. Proof of Efficiency

Ablation experiments evaluate the impact of each stage in our progressive fine-tuning strategy. Stage-1 focuses on improving the accuracy (i.e., the correctness of initial NL2LTL translation before any repair or refinement). Stage-2 enhances the model's ability to generate syntactically correct formulas and efficiently self-correct. Stage-3 further equips the model to align with safety constraints and enhances its ability to perform semantic repair.

Stage-1 Similar to [8], we adopt binary accuracy (i.e., 100% correct or not) to measure raw translation performance, although this is not our primary focus. The evaluation is conducted on the 300-sample evaluation benchmark with 6 different settings.

- **LLaMA-Origin**: the base LLaMA-3-8B-Instruct model without fine-tuning.
- **GPT-4**: zero-shot prompting using the same task format.
- **LLaMA-200-raw**: LLaMA fine-tuned on 200 pairs filtered only by `SPOT` for syntactic validity (no semantic cleaning).
- **LLaMA-200-clean**: LLaMA fine-tuned on 200 pairs after both syntactic validity and manual semantic refinement.
- **LLaMA-35k-lifted**: LLaMA fine-tuned on 35,000 NL–LTL pairs adapted from the 28k lifted STL corpus in [8]. Each STL formula is first converted into an LTL specification, where atomic propositions are rewritten to represent traffic navigation instructions. The corresponding natural language descriptions are then generated by GPT-4.

- **LLaMA-200-refined**: LLaMA fine-tuned on 200 cleaned pairs with deeper refinement and instruction-level optimization (cf. Section IV-A).

The results (cf. Table II) highlight the importance of data quality: simply cleaning a 200-example corpus (LLaMA-200-clean) narrows the gap to GPT-4, while a refined dataset makes LLaMa outperform GPT-4 with a lightweight 8B model. Conversely, a much larger but less targeted corpus (LLaMA-35k-lifted) degrades performance confirming that *task-specific, semantically aligned data outweighs sheer quantity* for compact LLMs in structured specification generation. Among Stage-1 configurations, LLaMA-200-refined achieves the highest accuracy and is therefore selected for subsequent fine-tuning.

Stage-2 To evaluate the effect of syntactic correction fine-tuning, we measure the *average number of syntax correction iterations* required to produce a syntactically correct LTL formula. This metric reflects both the model's initial correctness and its ability to efficiently repair errors when necessary.

As shown in Table II, `AutoSafeLTL` achieves the best results, reducing the average to **0.56** versus 1.72 at Stage-1—a 67% drop—outperforming LLaMA-Origin and GPT-4. This shows Stage-2 fine-tuning effectively instills syntactic knowledge, with many formulas correct on the first attempt.

Stage-3 To evaluate the effect of semantic correction fine-tuning, we measure the *average number of semantic correction iterations*—i.e., revisions required to achieve an `Included` status from a safety-violating LTL formula. This metric reflects the model's *intrinsic safety capability* and ability to efficiently repair non-compliant formulas.

The final stage further enhances semantic alignment through verification-guided correction. `AutoSafeLTL (Stage-3)` achieves the best performance on both syntax (**0.51** fixes) and semantic repair (**4.3** fixes), outperforming all baselines including GPT-4. And since Stage-3 introduces perturbations during counterexample-driven refinement, which may reduce surface-level matching but improve safety-compliance, its accuracy (93.6%) is slightly below the Stage-1's peak, yet still outperforms GPT-4. This aligns with our main goal—ensuring safety-compliant LTLs.

V. Framework Evaluation

A. Setup

We evaluate our framework's end-to-end effectiveness under a cloud–edge collaborative architecture, using *violation counts*, i.e., safety violations detected via Inclusion Check against predefined safety constraints, as the primary metric. Lower violation counts indicate better safety-compliance.

We deploy `GPT-4` as cloud-edge Agent LLMs without fine-tuning, accessed via API and prompt engineering. Our evaluation is conducted on the same evaluation benchmark (cf. Section IV-A).

We compare our approach against several baselines:

- **LLaMA&GPT-4**: zero-shot prompting using the same template as our method.
- **`nl2spec`** [9]: a LTL extraction tool. We adopt its default settings—`GPT-3.5-turbo` as the backend, `minimal` extraction mode, and `Number of tries` set to 3.
- **AutoSafeLTL**: our full pipeline under the collaborative cloud–edge setting.

Notably, many real-world navigation instructions in our dataset lack a rigid logical structure, resulting in very low translation coverage by `nl2spec` (less than 5% of original inputs produce valid LTL). For fair comparison, we use `GPT-4o` to rewrite all Desired Tasks into logically enhanced forms before `nl2spec` translation. These rephrased instructions serve as its evaluation input.

We also conduct ablation studies to investigate the necessity of both *User LLM* and *Agent LLM* in our framework. These variants isolate the contribution of each model and allow us to assess how collaboration between cloud and edge components influences final safety outcomes.

B. Quality of Initial Translation.

We first evaluate the *initial violation rate* on the benchmark—i.e., the proportion of initially generated LTL formulas that conflict with safety-compliance. Each initial LTL formula undergoes a single Inclusion Check: those marked INCLUDED are considered compliant, all others are violations.

Safety-compliance is often overlooked in prior LTL generation research. As shown in Table III, prior approaches exhibit a **100%** violation rate, indicating that none of their generated LTL formulas initially conform to the safety constraints. In contrast, our `AutoSafeLTL*`—a variant that uses only the *User LLM* without invoking the automated verification loop—achieves a violation rate of **98%**, successfully producing safety-compliant LTL at generation time.

This result, though modest, is significant: it confirms that progressive fine-tuning allows the model to internalize safety priors and generate compliant LTLs natively—a feat unmatched by baselines. It highlights the importance of training-level safety awareness for critical CPS applications.

C. Quality of Semantic Faithfulness

While our primary focus is provable safety-compliance rather than semantic faithfulness, we evaluated our framework's ability to preserve user intent. Following the protocol in [8], we tested models from all three fine-tuning stages on an extended dataset of 1,000 navigation tasks, using their consistency metric to verify that core objectives (e.g., waypoints, actions) remain intact in the generated LTL.

The semantic consistency scores across the three fine-tuning stages are 94.7% (S1), 92.5% (S2), and 91.1% (S3). A slight decline is observed across the stages, which aligns with our design goal: later stages increasingly prioritize safety-compliant repair when conflicts with safety constraints arise. However, the final model (Stage-3) still maintains a high consistency score ($> 91\%$), demonstrating that AutoSafeLTL effectively minimizes semantic drift while enforcing rigorous safety guarantees.

TABLE I
VIOLATION RATE AND INTERACTION ROUNDS ON THE BENCHMARK DATASET. **B** DENOTES THE AUTOMATED VERIFICATION COMPONENT; **C**, THE *LLM-as-an-Aligner*; AND **D**, THE *LLM-as-a-Critic*.

	LLaMA+B	**GPT-4+B**	**nl2spec+B**	**AutoSafeLTL**	**AutoSafeLTL-C**	**AutoSafeLTL-D**
Violation (%)	0	0	0	0	0	0
Avg. Interaction	16.2	7.8	6.5	**4.3**	15.0	11.0

TABLE II
STAGE-WISE PERFORMANCE OF OUR FRAMEWORK. ACCURACY REFLECTS THE INITIAL GENERATED LTL'S QUALITY; SYN./SEM. FIXES INDICATE AVERAGE ITERATIONS FOR SYNTAX/SEMANTIC REPAIR.

Model	**Accuracy (%)**	**Syn. Fixes**	**Sem. Fixes**
LLaMA-Origin	86.7 (260/300)	2.1	16.2
GPT-4	92.3 (277/300)	0.85	7.8
AutoSafeLTL (Stage-1)	**95.7 (287/300)**	1.72	7.1
LLaMA-200-raw	91.3 (274/300)	–	–
LLaMA-200-clean	92.7 (278/300)	–	–
LLaMA-35k-lifted	90.0 (270/300)	–	–
AutoSafeLTL (Stage-2)	94.3 (283/300)	0.56	6.8
AutoSafeLTL (Stage-3)	93.6 (281/300)	**0.51**	**4.3**

TABLE III
INITIAL VIOLATION RATE COMPARISON. **AUTOSAFELTL*** USES ONLY *User LLM* WITHOUT AUTOMATED VERIFICATION.

	llama	**gpt-4**	**nl2spec**	**autosafeltl***
violation (%)	100	100	100	**98**

D. Quality of Verified Translation and Ablation Study

Next, we evaluate the effectiveness of our safety assurance component *Automated Verification module*. *Safety-compliant* is defined by an LTL formula passing the Inclusion Check; we also measure the iterations required to reach this state as an efficiency metric.

Applying our full verification pipeline (`+B`) to LTLs generated by multiple baselines like `LLaMA`, `GPT-4`, and `nl2spec` yields a 0% violation rate across all models (Table I), confirming its generalizability. Notably, `AutoSafeLTL` achieves this with the fewest interaction rounds (5.7)—significantly outperforming `GPT-4+B` (7.8) and `LLaMA+B` (16.2)—demonstrating superior efficiency in aligning with safety constraints.

To further understand the role of each cloud-side agent, we perform ablation experiments by selectively removing *LLM-as-an-Aligner* (`AutoSafeLTL-C`) and *LLM-as-a-Critic* (`AutoSafeLTL-D`). Both variants still achieve full compliance, but at the cost of more iterations (15.0 and 11.0, respectively), highlighting the complementary importance of both agents in minimizing verification overhead.

E. Inference Latency and Efficiency

To assess the practical feasibility of `AutoSafeLTL` for real-time applications, we conducted a latency analysis on an NVIDIA A6000 GPU. With a batch size of 20 and five parallel safety constraints, the framework averages 0.27s per iteration. Given the 4.3-round average to reach full compliance (Table I), the total end-to-end latency is approximately 1.16s, making it compatible with high-level planning cycles. Detailed latency statistics are provided in our arXiv version [7]. Notably, formal verification (BA Construction and Inclusion Check) accounts for only $\approx 4\%$ of total time. This confirms that our counterexample path extraction algorithm is highly efficient and that the computational bottleneck lies in LLM inference rather than the safety assurance mechanism.

ACKNOWLEDGMENT

This work was supported by Guangzhou-HKUST(GZ) Joint Funding Program (Grant No. 2025A03J4493), Education Bureau of Guangzhou Municipality, Guangdong Provincial Project 2024QN11X053, and the Youth S&T Talent Support Programme of GDSTA (SKXRC2025468).

REFERENCES

[1] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

[2] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "NL2TL: Transforming natural languages to temporal logics using large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, 2023, pp. 15 880–15 903.

[3] K. Esterle, L. Gressenbuch, and A. Knoll, "Formalizing traffic rules for machine interpretability," in *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*. IEEE, 2020, pp. 1–7.

[4] P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar, "Advanced ramsey-based büchi automata inclusion testing," in *CONCUR 2011 – Concurrency Theory*, J.-P. Katoen and B. König, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 187–202.

[5] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. Gbaguidi Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, *From Spot 2.0 to Spot 2.10: What's New?*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, vol. 13372, p. 174–187.

[6] L. Clemente and R. Mayr, "Efficient reduction of nondeterministic automata with application to language inclusion testing," *Logical Methods in Computer Science*, vol. Volume 15, Issue 1, 2017.

[7] J. Li, M. Tian, and B. Zhong, "Automatic generation of safety-compliant linear temporal logic via large language model: A self-supervised framework," 2025. [Online]. Available: https://arxiv.org/abs/2503.15840

[8] Y. Chen, R. Gandhi, and Y. e. a. Zhang, "Nl2tl: Transforming natural languages to temporal logics using large language models," no. arXiv:2305.07766, arXiv:2305.07766 [cs].

[9] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, *nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models*, ser. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, vol. 13965, p. 383–396.