2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

# Stream-to-Act: ROS 2 Native Token Streaming for Continuous Motion Execution of Vision-Language-Action Models

Dahyun Kim[1,2], Yunseong Jeon[1], Hongkyun Park[1], and Jong-Chan Kim[1]

***Abstract*— Vision-Language Models (VLMs) are increasingly used in robotics for natural language understanding and executable plan generation, yet integrating them into real-time control pipelines remains challenging. Many existing systems rely on HTTP/JSON-based inference interfaces that require repeated Base64 serialization, introducing unnecessary overhead and increasing end-to-end latency. At the execution level, waiting for a full plan leads to stalls where no valid actions are available, while naive streaming of partial plans produces stop-and-go behavior due to token arrival gaps. To address these issues, we extend llama-ros with Stream-to-Act, a ROS 2-native execution mechanism that begins acting once sufficient tokens arrive while ensuring continuous execution through an optimal start-time policy. Our open-source implementation is evaluated on RTX GPUs and NVIDIA Jetson platforms through end-to-end latency analysis, token generation throughput measurements, and execution timeline visualization. In addition, a Carla-based driving scenario illustrates how the proposed execution policy eliminates stop-and-go behavior and maintains continuous control, even when the total plan generation time remains unchanged.**

## I. Introduction

Vision-Language Models (VLMs) have recently gained significant traction in robotics for their ability to interpret natural language instructions and generate executable plans [1]. Leveraging large-scale multimodal pretraining, researchers have explored planning, reasoning, and human–robot interaction in applications such as household robotics and autonomous driving [1, 2]. However, integrating these models into real-time robotic control requires not only accuracy but also system-level properties such as responsiveness and smooth execution [3, 4].

### A. Existing Systems and Contributions

Researchers typically rely on three categories of tools to access VLMs.

(1) **API-based services** (e.g., OpenAI API [5]) provide convenient access to state-of-the-art models but impose restrictions on local deployment and model selection. Moreover, they always require stable Internet connectivity [6], which is unsuitable for embedded robotics platforms or environments with limited network availability.

(2) **General-purpose serving frameworks** (e.g., llama.cpp [7], vLLM [8]) enable efficient deployment of multiple open-source models, yet they primarily follow HTTP/JSON batch inference pipelines. In such designs, streaming inputs (e.g., video frames) must be Base64-encoded and transmitted in JSON payloads. This transformation introduces redundant encoding and communication overhead, especially during Base64 conversion, making these frameworks poorly suited for embedded boards and real-time control loops.

(3) **Robotics-oriented open-source projects** have also made important contributions. OpenVLA [9] demonstrated the feasibility of end-to-end robotic policies with VLMs, while llama_ros [10] integrated llama.cpp [7] into ROS 2, lowering the barrier for robotics researchers to conduct local experiments. These systems have been crucial in enabling VLM experimentation within the robotics community.

### B. Problem: Streaming Execution and Stalls

Despite these advances, the challenge of streaming execution has not been sufficiently addressed from a robotics perspective [11, 12]. Waiting for a full plan to be generated leads to excessive end-to-end latency, while naive execution of partial plans often introduces *stalls*—control gaps where no valid action is available—resulting in undesirable stop-and-go behavior [1, 4]. Such discontinuity reduces safety margins, increases jerk, and may trigger fail-safe behaviors in robotic controllers [9]. Therefore, beyond reducing inference latency, it is critical to ensure execution *continuity*.

### C. Contributions of This Work

In this paper, we analyze the system pipeline of existing serving frameworks and identify why their HTTP/JSON + Base64 design is ill-suited for robotic streaming. Based on this analysis, we extend llama_ros [10] with a *Stream-to-Act* mechanism:

- We design a ROS-native streaming pipeline that eliminates redundant Base64 encoding and JSON packaging, directly delivering tokens and perceptual inputs to the control loop.
- We introduce a partial-plan execution strategy, committing actions as soon as they are generated, and analyze from which partial step execution should begin to maintain smooth motion.
- Instead of reporting stall ratios alone, we evaluate execution quality through **per-board output-format and policy-specific timeline graphs**, which visualize end-to-end delay and stop-and-go occurrences.
- We implement the system on ROS 2 with llama.cpp [7] and evaluate it on RTX GPUs and NVIDIA Jetson AGX Orin devices.

[1]Graduate School of Automobile and Mobility, Kookmin University, Korea. J.-C. Kim is the corresponding author {kd4hyun95, kb0316, hongkerous2619, jongchank}@kookmin.ac.kr
[2]Vehicle Solution Company, LG Electronics, Korea

979-8-3315-8160-2/26/$31.00 ©2026 IEEE

TABLE I: Comparison of Related Systems

| System | ROS 2 | Streaming | Comm. Method |
|---|---|---|---|
| SayCan [3], RT-2 [4], Gemini [2] | No | No | Other (Cloud API) |
| llama.cpp [7], vLLM [8], TGI [15], Ollama [16] | No | Limited | HTTP/JSON |
| OpenVLA [9] | No | No | Other (Custom) |
| llama_ros [10] | Yes | No | ROS 2 native (DDS) |
| LiteVLM [17], StreamVLN [11], VLM-TSI/TGLG [12] | No | Yes | Other (Task-specific) |
| **Our System (Stream-to-Act)** | Yes | Yes | ROS 2 native (DDS) |

Finally, in a Carla-based driving scenario [13], we show that while naïve streaming results in control gaps and stop-and-go behavior, our approach maintains continuous and smooth execution, thereby improving safety margins, driving stability, and control reliability—even without reducing the total plan generation time [14]. To facilitate reproducibility, the implementation is available at `https://github.com/DHKing7/stream-to-act`, which will be made publicly accessible upon publication.

## II. Related Work

### A. Vision-Language Models in Robotics

Vision-Language Models (VLMs) have recently emerged as a key component in enabling planning and interaction in robotics. Notably, Google's SayCan [3] connected a language model with a library of robot skills to ground language instructions into actions, while RT-2 [4] and Gemini Robotics [2] demonstrated that large-scale multimodal models can perform reasoning and planning in complex environments. However, these approaches primarily rely on cloud-based APIs and produce responses in the form of complete plans, which makes them difficult to apply directly to real-time, continuous control loops in robotics.

### B. Serving Frameworks and System Pipelines

Frameworks such as llama.cpp [7], vLLM [8], TGI [15], and Ollama [16] have provided the foundation for running a wide range of language and vision models efficiently. They have achieved significant progress in inference optimization and multi-model support. However, most of these frameworks follow HTTP/JSON batch inference pipelines, where large-scale data must be Base64-encoded and transmitted in JSON format. While this design is convenient for server–client environments, it is ill-suited for embedded boards and real-time control loops. In this work, we analyze such pipelines and extend them into a ROS-native streaming architecture tailored to robotics.

### C. Robotics-oriented VLM Systems

OpenVLA [9] proposed an end-to-end policy framework based on VLMs, demonstrating the feasibility of applying them to robotic tasks. In contrast, llama_ros [10] integrated llama.cpp [7] into ROS 2, significantly lowering the barrier for researchers to utilize VLMs in local environments. These studies played an essential role in expanding VLMs into robotics, and our work builds upon these contributions by newly highlighting the problem of stalls that arise during streaming execution.

### D. Streaming and Low-Latency Inference

More recently, studies have directly addressed streaming inference. LiteVLM [17] introduced model optimization techniques to accelerate VLM inference on embedded devices, while StreamVLN [11] designed a streaming architecture specialized for vision-language navigation tasks. VLM-TSI [12] and TGLG [12] proposed benchmarks for evaluating temporal alignment in streaming environments. However, these works primarily focused on task-specific improvements or model optimization, and few have explored ROS-native integration with quantitative analysis of execution continuity.

## III. System Overview

### A. Pipeline Redesign

Existing serving frameworks typically adopt an HTTP/JSON-based pipeline, where large-scale inputs such as images must be Base64-encoded and then placed into JSON payloads before being transmitted to the server. This is because JSON is a text-based format and cannot directly represent binary data. However, Base64 encoding increases the data size by approximately 33%, introduces additional CPU overhead, and requires repeated encoding and decoding on both client and server sides. While such a design is convenient in standard API environments, it is unsuitable for real-time applications such as robotic control loops. A compact comparison of related systems and their ROS 2 / streaming / communication properties is provided in Table I.

To address this issue, we designed a ROS 2-native pipeline. Sensor inputs are transmitted as ROS messages, llama.cpp [7]-based VLM inference runs within the ROS 2 environment, and generated tokens are published as ROS actions. This removes unnecessary encoding steps and integrates tightly with existing stacks via the ROS 2 middleware (DDS pub–sub).

### B. Stream-to-Act Mechanism

The ROS-native structure itself primarily simplifies data transmission and improves integration. However, it does not by itself solve the problems of stalls or improve the smoothness of execution. To address these issues, we introduce a Stream-to-Act mechanism, in which partial plans generated by the VLM are immediately committed to execution, thereby reducing control gaps. The detailed design of Stream-to-Act is presented in the Methods section.

### C. Design Goals

The design goals of the proposed system are as follows:

- **Integration**: Direct compatibility with the ROS 2 middleware, enabling seamless coupling with existing robotic software stacks.

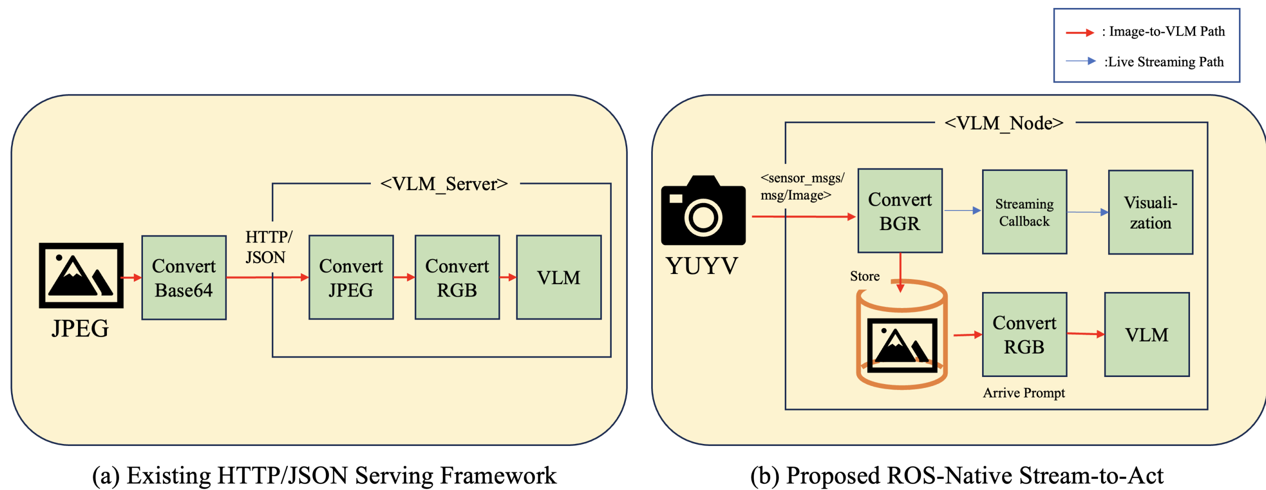



Fig. 1: Comparison of serving pipelines: (a) Existing HTTP/JSON framework with Base64/JSON overhead and stalls, (b) Proposed ROS-native Stream-to-Act with smooth control.

- **Reproducibility**: Capable of running on embedded platforms such as NVIDIA Jetson AGX Orin with a simple colcon build, facilitating reproducible experiments across researchers.
- **Simplicity**: Removal of unnecessary serialization steps, relying solely on the ROS messaging model to reduce maintenance overhead.

## IV. Proposed Method

### A. HTTP/JSON Pipeline and Base64 Overview

Most existing serving frameworks adopt an HTTP/JSON-based communication pipeline. Since JSON is a text-based format, it cannot directly embed binary data such as images or video frames. Therefore, inputs must first be transformed into text strings, most commonly through Base64 encoding. As summarized in Fig. 1, such HTTP/JSON pipelines accumulate serialization overheads in streaming settings, whereas our ROS 2-native design eliminates these steps.

**How Base64 works.** Input data are grouped in chunks of 3 bytes (24 bits), which are split into four 6-bit segments. Each 6-bit value is mapped to one of 64 printable characters (A–Z, a–z, 0–9, +, /). Thus, **How Base64 works (mapping vs. wire size)**

$$\underbrace{\text{3 bytes}}_{\text{24 bits}} \xrightarrow{\text{grouping}} \underbrace{\text{4 sextets}}_{\text{6 bits}\times 4} \xrightarrow{\text{alphabet map}} \underbrace{\text{4 chars}}_{\text{8 bits}\times 4}$$

$$\Rightarrow \text{ payload expansion} = \frac{4 \times 8 \text{ bits}}{3 \times 8 \text{ bits}} = \frac{4}{3} \approx 1.33$$

*Padding:* If the input length is not a multiple of 3, the output is padded with one or two “=” symbols so that its length becomes a multiple of 4. For large image streams, the overall overhead still converges to $\sim$33%.

**Problem summary.** For every frame, the pipeline introduces (i) Base64 encoding cost on the client, (ii) larger payloads for network transmission, (iii) Base64 decoding cost on the server, and (iv) JSON string processing. These accumulate in streaming scenarios, leading to control stalls where actuators wait for commands.

### B. Latency Breakdown of HTTP/JSON Streaming

For each streamed frame (or chunk), the end-to-end latency can be decomposed as:

$$\begin{aligned} T_{\text{e2e}}^{\text{HTTP}} &= T_{\text{capture}} + T_{\text{encode}}^{\text{B64}} + T_{\text{pack}}^{\text{JSON}} \\ &+ T_{\text{network}}\big(\alpha\,|\text{payload}|\big) \\ &+ T_{\text{decode}}^{\text{B64}} + T_{\text{parse}}^{\text{JSON}} + T_{\text{inference}} + T_{\text{post}}. \end{aligned} \tag{1}$$

where $\alpha = \frac{4}{3}$ is the Base64 expansion factor. Importantly, $T_{\text{encode}}^{\text{B64}}$, $T_{\text{pack}}^{\text{JSON}}$, $T_{\text{decode}}^{\text{B64}}$, $T_{\text{parse}}^{\text{JSON}}$ are pure serialization overheads, independent of the model’s performance, and become bottlenecks in real-time robotic control loops.

### C. ROS-native Pipeline with v4l2 YUYV Pass-Through

In our implementation, the USB camera is connected via the v4l2 driver and configured to output YUYV. This mode is pass-through (uncompressed), so frames are delivered without an extra compression/encoding stage—hence no added encoding latency from the camera path.

On the ROS side, incoming YUYV frames are converted to BGR to interoperate with the streaming/computer-vision library (OpenCV). The latest BGR frame is continuously cached during streaming. When a user prompt arrives, the cached frame is converted to RGB and fed to the VLM.

- **Why BGR $\rightarrow$ RGB.** OpenCV’s image routines conventionally use BGR ordering, while most deep models expect RGB. The BGR$\rightarrow$RGB step is only a channel reorder (no re-sampling or compression), so the cost is negligible compared to serialization or Base64.
- Color conversions in this pipeline
  - YUYV $\rightarrow$ BGR: colorspace conversion (local CPU/GPU op).
  - BGR $\rightarrow$ RGB: channel swap (very low overhead).

### D. Stream-to-Act Mechanism

The ROS-native design improves efficiency by removing serialization overheads, but does not by itself address stalls or smooth execution. To solve this, we introduce a Stream-to-Act mechanism.

- The token stream produced by the VLM is parsed into partial plans.
- Once a valid plan fragment is detected (e.g., delimiter- or template-based), it is immediately committed to execution.
- If a short gap occurs before the next fragment arrives, each skill $s_i$ can be safely extended within its temporal bounds $\big(T_{\min}(s_i),\, T_{\text{nom}}(s_i),\, T_{\max}(s_i)\big)$, preventing control gaps.

This policy ensures that even if the total inference time $T_{\text{inference}}$ is not shortened, control gaps are minimized, enabling continuous and smooth execution.

### E. Output Format and Prompt Example

The proposed Stream-to-Act mechanism defines model outputs in the form of a **skill-based domain-specific language (DSL)**. For example, when a robot is equipped with primitive skills such as `go_to(location)`, `open(object)`, `pick(object)`, and `place(object, location)`, the model output can be represented as a numbered sequence of skill commands:

```
1. go_to(kitchen)
2. pick(cup)
3. go_to(table)
4. place(cup, table)
```

In a streaming environment, such a plan is delivered in *fragments*, and its **granularity** directly influences both smoothness and responsiveness of execution. We parameterize the output granularity as the number of tokens per skill ({6, 4, 2, 1} tokens/skill).

Larger granularities produce stable but slower responses, while smaller granularities allow more aggressive streaming. Under the naive policy this increases stall risk as $\Delta = L/R$ grows, whereas Stream-to-Act mitigates stalls by selecting $S_0$ and extending skills within $(T_{\min}, T_{\text{nom}}, T_{\max})$.

Accordingly, the system treats granularity as a tunable parameter, enabling a trade-off between *aggressive streaming* and *conservative streaming*. This design demonstrates that Stream-to-Act is not only capable of executing partial plans, but also of controlling the output granularity to balance smoothness and responsiveness in robotic control.

**Prompt Example.** The following is a representative prompt used for robotic planning:

```
You are a robot task planner.
Skills: go_to(location), open(object),
        close(object), pick(object),
        place(object, location).
Rules: Output only numbered skill calls,
        one per line.
Do not output any explanation.
```

This prompt leads the model to generate structured plans in the DSL format, which can then be consumed directly by the Stream-to-Act executor. During inference, each skill fragment arrives incrementally, and the executor immediately integrates partial plans into execution according to the chosen output granularity.

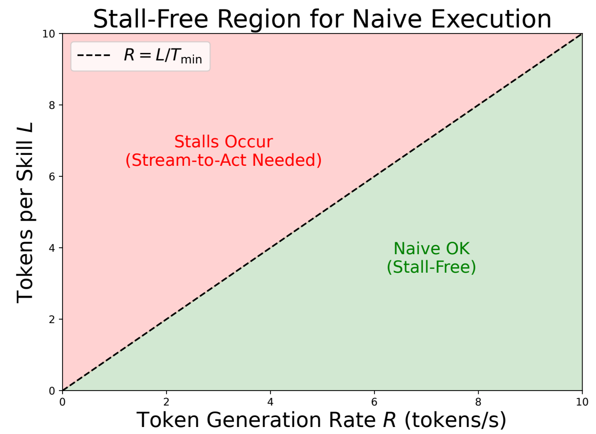



Fig. 2: Stall-free boundary of naive execution. When $R \geq L/T_{\min}$ (green region), naive execution suffices. Otherwise (red region), stalls occur and Stream-to-Act is required.

**Skill-Scoped Prompting and Hallucination Control.** In this work, we do not add a separate runtime hallucination filter; instead, we control the VLM strictly through prompt design and a fixed set of skills. The DSL enumerates a closed vocabulary of actions and arguments, and the prompt constrains the model to produce only these skills within a predefined template. The proposed Stream-to-Act formulation is thus built under the assumption that the underlying model can be steered to stay within this skill space. From an engineering perspective, preventing hallucinated or out-of-scope skills is part of the system designer's responsibility, and our contribution focuses on how to exploit such controllable models in real-time control loops.

### F. Stream-to-Act Formulation

Let $N$ be the number of skills in a plan, and each skill $s_i$ requires a nominal execution time $T_{\text{nom}}(s_i)$. The VLM generates fragments (plan tokens) sequentially, with the $i$-th fragment arriving at absolute time $t_i$. The inter-arrival interval is $\Delta = L/R$, where $L$ is the number of tokens required per skill and $R$ is the token generation rate (tokens/s). The time-to-first-token is denoted as $T_{\text{TFT}}$.

**Naive execution.** In the naive policy, execution of $s_i$ begins immediately once fragment $i$ is available. This can cause stall durations

$$\text{stall}_i = \max\big(0,\ t_{i+1} - e_i\big),$$

where $e_i$ denotes the end time of skill $s_i$. The high-level behaviors of *wait-then-act*, *naive stream*, and *Stream-to-Act* in the CCRB scenario are illustrated in Fig. 3.

[Stall-Free Condition for Naive Execution] Naive execution is stall-free if the minimal execution time of each skill is no shorter than the fragment inter-arrival interval:

$$T_{\min}(s_i) \geq \Delta = \frac{L}{R}, \quad \forall i.$$

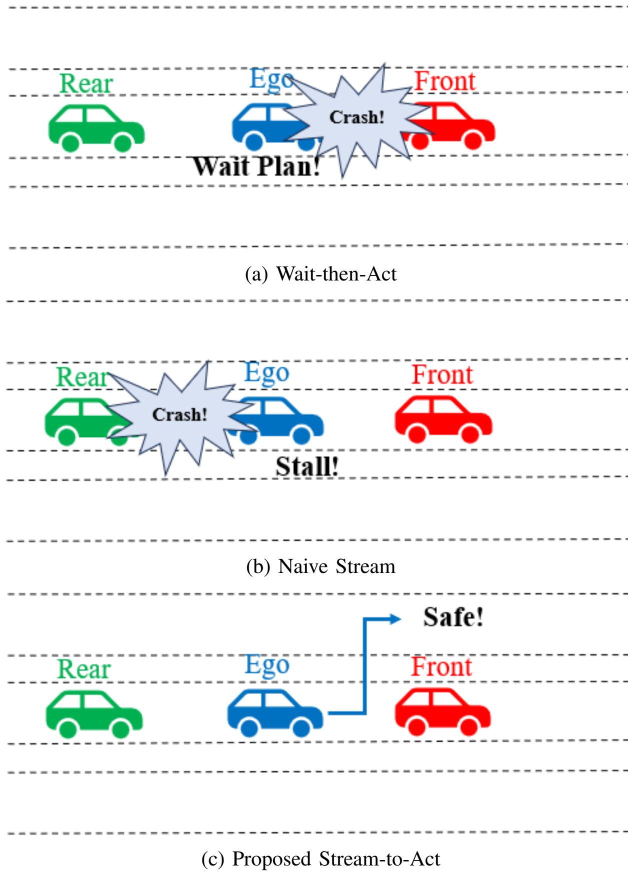



Fig. 3: Carla scenario comparison: (a) wait-then-act, (b) naive streaming with stop-and-go, (c) proposed stream-to-act.

Equivalently, $L \le R T_{\min}(s_i)$ or $R \ge L/T_{\min}(s_i)$.
*Sketch.* In naive execution, $s_i$ ends at $e_i = t_i + T(s_i)$. Stall-free requires $t_{i+1} \le e_i$, i.e., $t_{i+1} - t_i \le T(s_i)$. With regular arrivals, $t_{i+1} - t_i = \Delta = L/R$. Hence, if $T_{\min}(s_i) \ge \Delta$, the inequality holds for all $i$. If arrivals are subject to bounded jitter with $\Delta_{\max}$, the condition $T_{\min}(s_i) \ge \Delta_{\max}$ suffices.
**Stream-to-Act (minimal safe start at $S_0$).** When the above condition does not hold, stalls may occur under naive execution. We therefore define the minimal safe start time $S_0$ that guarantees contiguous execution (no stalls):

$$S_0 \;=\; \max_i \bigl(t_i - (i-1)\,T_{\text{nom}}\bigr).$$

[Stall-Free Guarantee of Stream-to-Act] Starting execution at $S_0$ ensures that, for all $i$,

$$t_i \;\le\; S_0 + (i-1)\,T_{\text{nom}},$$

so each skill is available by its start time and no control gaps occur. Moreover, $S_0$ is minimal: any $S < S_0$ violates the inequality for some $i$, resulting in at least one stall.
*Sketch.* By definition, $t_i \le S_0 + (i-1)\,T_{\text{nom}}$ for all $i$. With back-to-back execution, $s_i$ runs on $[\,S_0 + (i-1)T_{\text{nom}},\; S_0 + iT_{\text{nom}}\,]$, so fragment $i$ is available by the start of $s_i$, implying continuity. Minimality: if $S < S_0$, then for some $i^\star$ we have $t_{i^\star} > S + (i^\star - 1)T_{\text{nom}}$, meaning $s_{i^\star}$ is not ready at its start, causing a stall.

**Remark (buffer extension).** If fragments arrive slightly later, each skill $s_i$ can be extended within its temporal bounds

$$\bigl(T_{\min}(s_i),\; T_{\text{nom}}(s_i),\; T_{\max}(s_i)\bigr),$$

effectively absorbing micro-delays. In practice, if the inter-arrival time is bounded by $\Delta_{\max}$ and $T_{\max}(s_i) \ge \Delta_{\max}$, continuity can still be maintained without stalls. The resulting decision boundary for naive execution ($R \ge L/T_{\min}$) is visualized in Fig. 2.

## V. Experiments

### A. Experimental Setup

We use the MiniCPM-2.6 model. Visual inputs are divided into $14 \times 14$ patches to produce visual tokens, and text inputs follow the prompt templates described earlier.
**Hardware/Software.**

- Desktop GPU: NVIDIA GeForce RTX 4080
- Embedded Board 1: NVIDIA Jetson AGX Orin (JetPack 6.0.2, CUDA 12.6)
- Embedded Board 2: NVIDIA Jetson Orin Nano (JetPack 6.0.2, CUDA 12.6)
- Common Software: Ubuntu 22.04, ROS 2 Humble

Our system is implemented as a ROS 2-native node by extending `llama_ros` [10], which wraps `llama.cpp` [7]. We further extend it with a *stream-to-act* execution mechanism that commits partial plans during token generation.
**Binary Transport Discussion.** Binary transport protocols such as DDS zero-copy, WebSocket binary frames, or ZeroMQ may reduce serialization overhead. However, our goal in this work is not to benchmark all transport mechanisms, but to analyze how widely used HTTP/JSON-based VLM serving pipelines affect real-time robotic execution. Many VLM-based robotic systems still rely on HTTP/JSON interfaces inherited from cloud-oriented APIs. Therefore, our evaluation focuses on this common baseline to highlight the impact of serialization overhead and execution policies on control continuity. A systematic comparison with additional binary transports remains an interesting direction for future system-level work.

### B. E2E Latency Measurement

We compare end-to-end (e2e) latency between an HTTP/JSON baseline and our ROS 2-native pipeline on the Orin AGX. Figure 4 reports averages over 50 runs. The measured latency is 101.8 ± 8.6 ms for HTTP/JSON and 15.5 ± 1.0 ms for ROS-native. Although the HTTP pipeline appears to have lower "encoding" time, this is because a pre-encoded JPEG is provided to the client (only Base64 is performed). In contrast, the ROS 2-native path includes the realistic YUYV→BGR conversion present in streaming. Overall, the HTTP pipeline incurs substantial serialization/decoding overheads, whereas the ROS 2-native pipeline avoids these redundant steps.

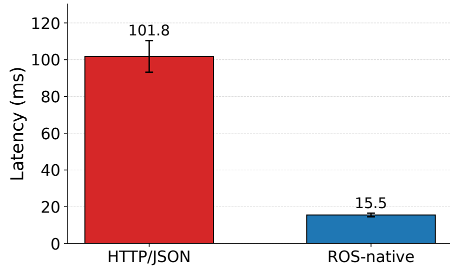



Fig. 4: Comparison of mean e2e latency on Orin AGX. HTTP/JSON shows higher overall latency due to serialization and decoding overhead, while ROS-native avoids these steps despite additional YUYV→BGR conversion.

TABLE II: Token generation throughput (10-run average). Stream-to-Act benefits are most evident on embedded boards with low tokens/s.

| Platform | Token Generation Speed (tokens/s) |
|---|---|
| NVIDIA GeForce RTX 4080 | 34.2 |
| NVIDIA Jetson AGX Orin | 4.8 |
| NVIDIA Jetson Orin Nano | 0.58 |

### C. Latency Breakdown

To analyze bottlenecks, we decompose the latency into individual stages. For HTTP/JSON, JSON packing and the HTTP POST are merged into one category. For ROS 2-native, we measure YUYV→BGR conversion and BGR→RGB channel swapping separately. Figure 6 shows that most HTTP overhead stems from repeated serialization (Base64, JSON) and server-side decoding, while ROS 2-native latency is dominated only by lightweight color conversions.

### D. Token Generation Speed

We measure tokens per second (*tokens/s*):

$$\text{Throughput (tokens/s)} = \frac{N_{\text{tokens}}}{T_{\text{eval}}}.$$

Per-platform throughput (10-run averages) is summarized in Table II. High-end GPUs achieve near real-time responses, whereas throughput drops markedly on embedded boards, making *stream-to-act* particularly valuable to maintain smooth control.

### E. Execution Timelines and Policy Analysis

**Protocol.** For each hardware (NVIDIA GeForce RTX 4080, NVIDIA Jetson AGX Orin, NVIDIA Jetson Orin Nano), we evaluate three policies—*naive*, *wait-then-act*, and *stream-to-act*—under four output granularities $\{6, 4, 2, 1\}$ tokens/skill (see Fig. 5). We log token arrival timestamps and executed skills to produce per-board, per-policy timelines, reporting (i) qualitative continuity via timelines and stop-and-go events and (ii) end-to-end latency.

**Findings by hardware.**

- **NVIDIA GeForce RTX 4080 (Fig. 5a):** With 34.2 tokens/s, all policies run smoothly with no noticeable stalls; the advantage of *stream-to-act* is less pronounced.
- **NVIDIA Jetson AGX Orin (Fig. 5b):** At 4.8 tokens/s, *naive* shows intermittent stalls; *wait-then-act* avoids stalls but increases e2e latency; *stream-to-act* eliminates stalls by choosing an appropriate start time $S_0$ without a significant latency penalty.
- **NVIDIA Jetson Orin Nano (Fig. 5c):** At 0.58 tokens/s, *naive* exhibits frequent stop-and-go and *wait-then-act* induces excessive delays; *stream-to-act* preserves continuity by starting at $S_0$.

**Effect of output granularity and hardware speed.** Let $R$ be the token generation rate and $L \in \{6, 4, 2, 1\}$ the tokens/skill. In *naive* execution, the fragment interval $\Delta = L/R$ increases as hardware performance degrades (smaller $R$) or plans become more verbose/specific (larger $L$), raising stall likelihood; this trend is visible in the timelines. In contrast, *stream-to-act* maintains continuity across granularities by selecting $S_0$ and extending skills within $(T_{\min}, T_{\text{nom}}, T_{\max})$, making it preferable for fast and smooth control, especially on NVIDIA Jetson AGX Orin compared to NVIDIA Jetson Orin Nano.

### F. Demonstration

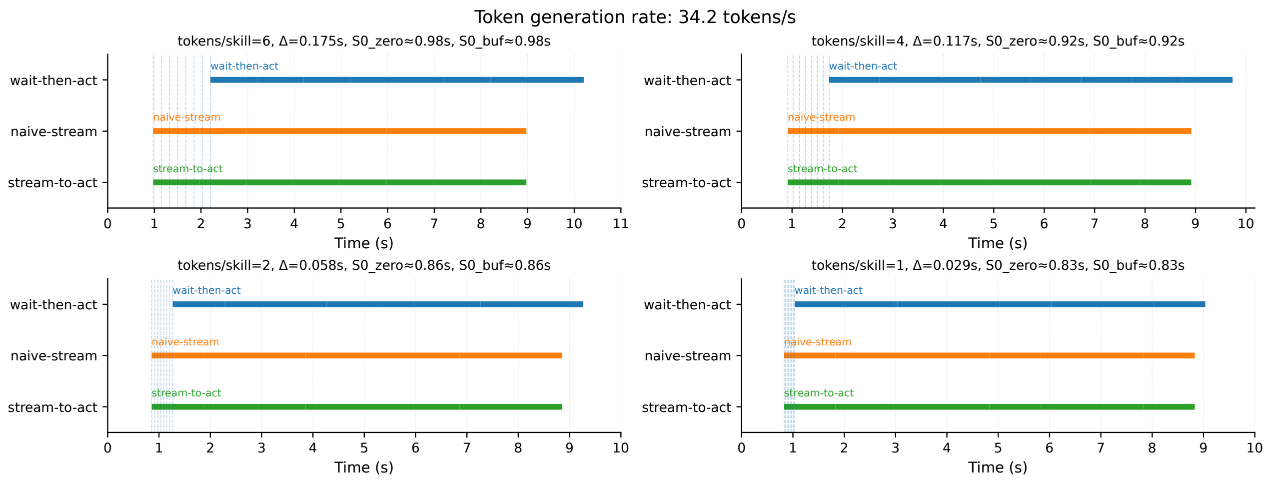
**Purpose.** This demo illustrates how the start-time policy affects execution continuity in streaming VLM control. In particular, it shows that starting the plan slightly later—at the minimal safe start time $S_0$—is crucial for smooth and safe control. Even when the total inference time remains unchanged, an appropriate start-time policy eliminates stop-and-go behavior. The scenario is simplified to highlight behavioral differences between execution policies. Our Carla configuration (lead/ego/trailing vehicles with ROS 2 integration) is shown in Fig. 7.

**Setup.** One NVIDIA Jetson AGX Orin (on-robot, ROS 2-native node with Stream-to-Act executor) and one workstation running the Carla simulator. The scenario is a Euro NCAP Car-to-Car Rear Braking (CCRB) variant where the lead vehicle decelerates, the ego vehicle follows, and a trailing vehicle is present.

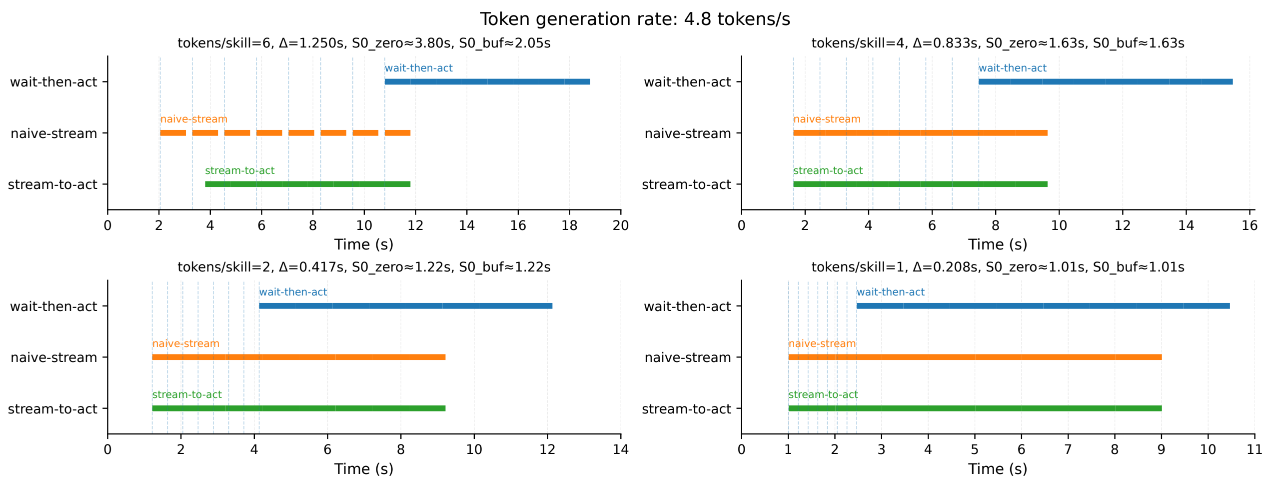
**Policies and expected outcomes.**

- *Naive stream (immediate start):* Executes as soon as tokens arrive. Speed reduction occurs promptly, but token gaps induce control stalls and stop-and-go behavior, leading to a **rear-end collision** with the trailing car.
- *Wait-then-act (full-plan wait):* Waits until the complete plan is generated. Due to planning delay, the ego car fails to decelerate in time, causing a **front collision** with the lead vehicle.
- *Stream-to-Act (delayed start at $S_0$):* Begins execution at the computed minimal safe start time, achieving smooth deceleration and lane changes while avoiding **both front and rear collisions**.

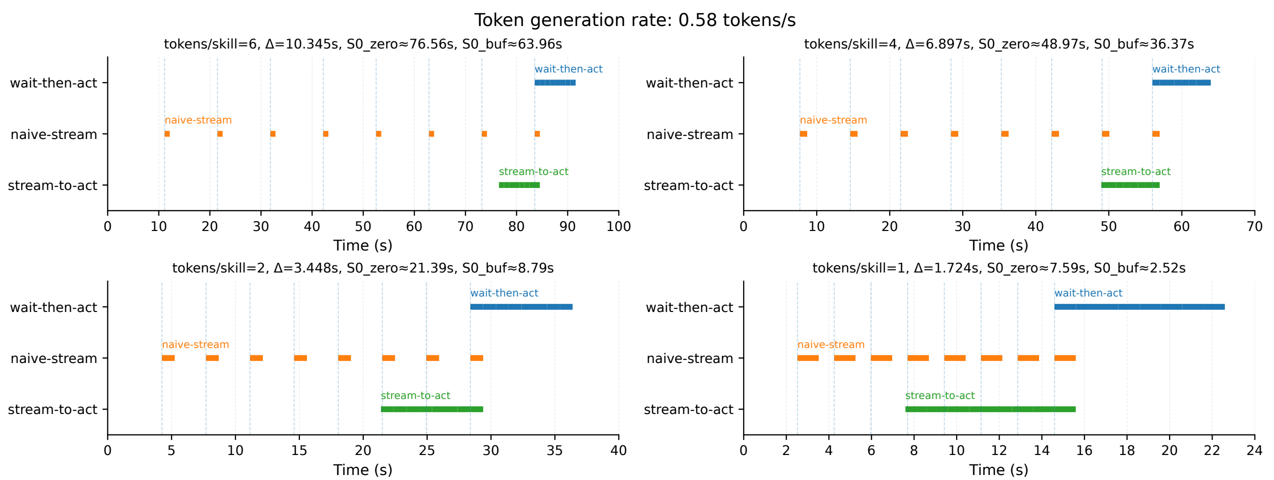
**Takeaway.** *Naive stream* reacts quickly but is unsafe due to stalls; *wait-then-act* is unsafe due to delay. Only *Stream-to-Act*, by starting slightly later at $S_0$—delivers smooth, continuous control and avoids collisions.




(a) NVIDIA GeForce RTX 4080




(b) NVIDIA Jetson AGX Orin




(c) NVIDIA Jetson Orin Nano

Fig. 5: Execution timelines under different output formats (tokens/skill = 6, 4, 2, 1) across hardware platforms: (a) NVIDIA GeForce RTX 4080 (34.2 tokens/s), (b) NVIDIA Jetson AGX Orin (4.8 tokens/s), and (c) NVIDIA Jetson Orin Nano (0.58 tokens/s). Each subplot compares three policies: wait-then-act (blue), naive (orange), and stream-to-act (green).

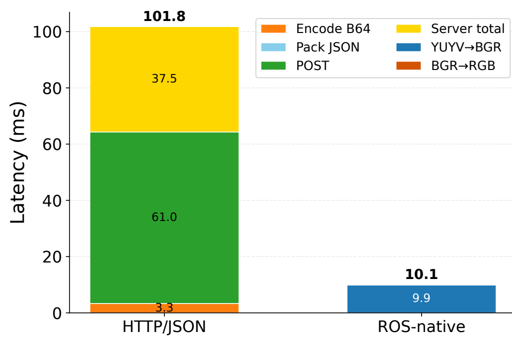



Fig. 6: Latency breakdown (mean) on Orin AGX. HTTP path suffers from multiple serialization/decoding steps, whereas ROS-native path only includes lightweight YUYV→BGR and BGR→RGB conversions.

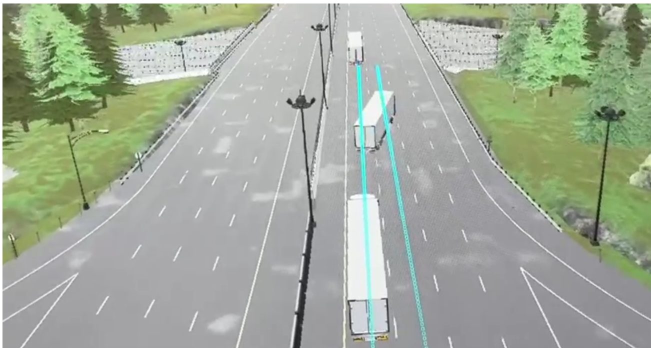


Fig. 7: Carla simulator setup for the CCRB scenario: lead, ego, and trailing vehicles; ROS 2-native integration; and control/data flow used in our experiments.

## VI. Conclusion and Future Work

We proposed a ROS-native Stream-to-Act system that extends llama_ros [10] to overcome the limitations of HTTP/JSON pipelines in robotic control. Our approach eliminates serialization overhead and executes partial plans in streaming mode, reducing stalls and ensuring smooth control. Results on RTX GPUs and NVIDIA Jetson AGX Orin confirm that even without reducing end-to-end latency, continuity metrics such as stall ratio and stop-and-go count are crucial for safe and reliable execution.

Future work includes expanding validation to more complex scenarios such as urban driving and multi-agent interaction, establishing formal boundary conditions where Stream-to-Act is most beneficial across hardware platforms, and developing policies that balance smoothness with strict safety margins (e.g., Time-to-Collision constraints). A formal theoretical characterization of when Stream-to-Act guarantees stall-free operation would further strengthen its general applicability in robotics.

## References


[1] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter *et al.*, "Palm-e: An embodied multimodal language model," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*. PMLR, 2023, pp. 8469–8488. [Online]. Available: https://proceedings.mlr.press/v202/driess23a.html

[2] G. R. Team, S. Abeyruwan, J. Ainslie, J.-B. Alayrac, M. Gonzalez Arenas, T. Armstrong *et al.*, "Gemini robotics: Bringing ai into the physical world," 2025. [Online]. Available: https://arxiv.org/abs/2503.20020

[3] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," 2022. [Online]. Available: https://arxiv.org/abs/2204.01691

[4] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski *et al.*, "Rt-2: Vision-language-action models transfer web knowledge to robotic control," 2023. [Online]. Available: https://arxiv.org/abs/2307.15818

[5] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman *et al.*, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal *et al.*, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[7] G. Gerganov *et al.*, "llama.cpp: Inference of llama model in pure c/c++," https://github.com/ggml-org/llama.cpp, 2023, accessed: 2025-09-14.

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu *et al.*, "Efficient memory management for large language model serving with pagedattention," 2023. [Online]. Available: https://arxiv.org/abs/2309.06180

[9] M. J. Kim, K. Pertsch, S. Karamcheti, T. Xiao, A. Balakrishna, S. Nair *et al.*, "Openvla: An open-source vision-language-action model," 2024. [Online]. Available: https://arxiv.org/abs/2406.09246

[10] M. González, "llama_ros: Ros 2 integration for llama.cpp," https://github.com/mgonzs13/llama_ros, 2023, accessed: 2025-09-14.

[11] M. Wei, C. Wan, X. Yu, T. Wang, Y. Yang, X. Mao *et al.*, "Streamvln: Streaming vision-and-language navigation via slowfast context modeling," 2025. [Online]. Available: https://arxiv.org/abs/2507.05240

[12] K. P. Yu and J. Chai, "Temporally-grounded language generation: A benchmark for real-time vision-language models," 2025. [Online]. Available: https://arxiv.org/abs/2505.11326

[13] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning (CoRL)*, 2017, pp. 1–16. [Online]. Available: https://arxiv.org/abs/1711.03938

[14] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, pp. 125–142, 2020.

[15] H. Face, "Text generation inference," https://github.com/huggingface/text-generation-inference, 2023, accessed: 2025-09-14.

[16] Ollama, "Ollama: Run large language models locally," https://github.com/ollama/ollama, 2023, accessed: 2025-09-14.

[17] J. Huang, Y. Jin, L. An, and J. Park, "Litevlm: A low-latency vision-language model inference pipeline for resource-constrained environments," *arXiv preprint arXiv:2506.07416*, 2025. [Online]. Available: https://arxiv.org/abs/2506.07416