JARVIS: Disjoint **La**rge Language Models on **R**adio **V**LANs for **I**ntelligent **S**ervices

Miquel Sirera Perelló, Joshua Groen, Wan Liu, Stratis Ioannidis, Kaushik Chowdhury
Institute for the Wireless Internet of Things, Northeastern University, Boston, MA, USA
{sirera.m, groen.j, liu.wan1, e.ioannidis, k.chowdhury}@northeastern.edu

***Abstract*—Large Language Models (LLMs) have changed the way we access and interpret information, communicate with each other and even operate computer systems through autonomous code generation. Typically, these billion-parameter models rely on cloud storage and execution due to their computational demands. In this paper, we challenge this status quo by proposing JARVIS, a distributed LLM framework that splits model layers across edge devices with limited compute resources, trading off computation for increased peer-level communication. JARVIS is robust to individual node failures, including recovery methods for lost layers via peer-level duplication. We evaluate JARVIS using Google's open-source Gemma LLM (2B parameters) deployed over 18 software-defined radios in the NSF Colosseum RF emulator. Our evaluation explores LLM performance degradation from node losses, providing insights into node prioritization in tactical environments. The JARVIS software code is released for community exploration and adoption.**

***Index Terms*—Artificial Intelligence, Generative AI, Large Language Models, Network Slicing, Distributed Computing**

I. Introduction

Generative AI, particularly large language models (LLMs), is driving innovations in language translation, text generation, and automated reasoning [1]. However, their significant computational demands require centralized cloud-based inference. Models like GPT-4 [2], Gemini [3], and Llama 3 [4] range from billions to over a trillion parameters, requiring substantial storage. In this paper, we propose JARVIS, a novel approach that splits a trained LLM across multiple distributed nodes, eliminating the need for powerful remote clouds and enhancing resilience. JARVIS supports distributed intelligence, is robust to node failure, and is capable of self-repairing and self-organizing across various network topologies and constraints.

Need for Distributed AI for Inference, Spotlight on LLMs: Traditionally, LLMs execute queries on remote servers, but this is impractical in tactical scenarios lacking reliable cloud connections and infeasible on handheld devices [5]. Techniques like model pruning [6] and quantization [7] have limits and can degrade inference capability. Centralized points of failure are also undesirable due to potential adversarial actions. Thus, we believe a distributed LLM on cooperating nodes offers significant advantages. By incorporating resilience-by-design, JARVIS overcomes the limitations of centralized LLMs.

Networking and AI are tightly integrated. AI has primarily optimized network operations, but there is an opportunity to design networks specifically for AI tasks. JARVIS exemplifies this vision by allocating LLM components to nodes based on connectivity, storage, and computational ability. This balances the computational power of cloud architectures with the benefits of distributed networks, such as eliminating single points of failure, enabling localized inference, and enhancing privacy, despite the overhead of inter-node communication.

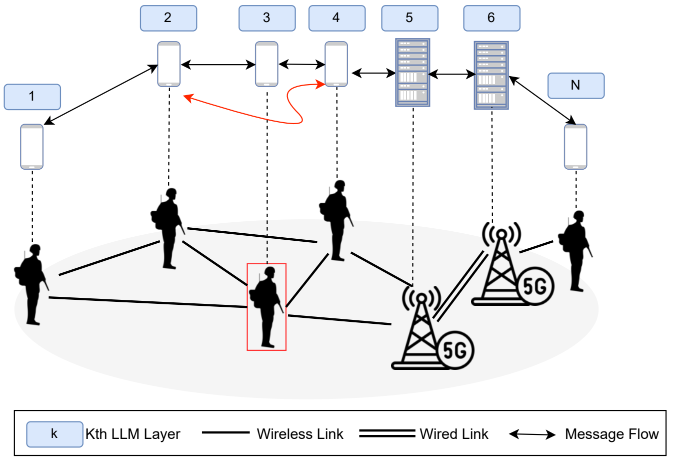


Fig. 1. Overview of the JARVIS framework. The numbers represent different layer-wise splits of the LLM, with the first and last parts running at the UE. JARVIS supports distributed nodes with both wireless and wired links and is resilient to intermediate layer/node losses by re-routing the connections to the next hop, as shown in the case of the node/layer 3.

Challenges in Designing JARVIS: Overlaying an LLM on a physical network is complex. We must identify optimal split points to maintain performance and map each split within the network, considering nodes' CPU/GPU resources and inter-node link interference. A node with better GPU may be ineffective if its links have high bit error rate. To address potential node failures, we replicate and distribute LLM splits across the network, ensuring redundancy and recovery opportunities. Fig. 1 shows JARVIS' distributed inference and recovery mechanisms.

Our main contributions in the design and deployment of JARVIS are:

- Demonstrating how an open-source LLM, such as Google's Gemma, can be split into over a dozen layers, each **executed on separate, resource-constrained devices**.
- **Novel layer skipping and recovery mechanisms** for resilient operation, allowing safe skipping of layers and retrieval of LLM splits from peer devices.

TABLE I
COMPARISON OF ARCHITECTURAL FEATURES AND PERFORMANCE METRICS OF FIVE OPEN-SOURCE LLMS

Model	Release Time	# Params (B)	# Layers	Model Size (GB)	Token Embedder (MB)	Layer size (MB)	Max. length context window	O/P size per token (MB) [8k tokens]	MMLU ↑
LLaMA3 8B	Apr-24	8	32	29.92	2004	832.02	8k	0.0156 [125]	66.6
Mistral 7B v0.1	Sep-23	7	32	26.49	500	832.04	8k	0.0156 [125]	60.1
Gemma 7B	Feb-24	7	28	31.81	3000	1056.02	8k	0.0117 [93.75]	64.3
Gemma 2B	Feb-24	2	18	9.34	2000	320.02	8k	0.0078 [62.5]	42.3
Phi-3 mini 128k	Apr-24	3.8	32	14.23	375.75	432.02	128k	0.0117 [93.75]	68.1

- **Automated orchestration framework** that configures and adapts communication between LLM layers across heterogeneous devices, optimizing performance and flexibility.
- **First time deployment at scale** on 18 radio nodes in the NSF Colosseum emulator, rigorously evaluating network traffic and computational overhead.

The remainder of the paper is organized as follows: Section II provides a background on LLMs, split Deep Neural Networks (DNNs) and distributed networks for AI; Section III describes the detailed design and implementation of JARVIS; Section IV presents a real deployment and evaluation of the system's performance; and Section V concludes the paper.

II. BACKGROUND AND RELATED WORKS

A. *State-of-the-Art in LLMs*

The Transformer architecture from [8] revolutionized natural language processing by introducing the attention mechanism. The latter enables models to generate effective representations from the input data by contextualizing it within sentences. This advancement led to the development of LLMs, known for their exceptional generalization abilities. Table I compares five popular open-source models, highlighting their size, features and performance on the Massive Multitask Language Understanding (MMLU) [9] benchmark, a widely used and established measure in natural language processing for evaluating the general knowledge and reasoning abilities of LLMs.

The computational demands of state-of-the-art LLMs, such as LLaMA3 8B and Mistral 7B, which require over 20 GB of storage and substantial RAM pose significant storage and memory challenges, making them impractical for devices with limited resources. These requirements often necessitate specialized hardware and cloud-based solutions for execution. Consequently, splitting and distributing the model for inference allows the utilization of constrained devices that cannot run the entire model but can efficiently manage smaller parts.

B. *Distributed Computing for AI Workloads on Edge Networks*

Advancements in distributed computing for DNNs have enabled training and inference of large models beyond single machines' capabilities. Techniques like data, tensor, and pipeline parallelism manage these models despite communication and synchronization challenges [10]–[12]. While tensor parallelism is common for sharding models, it introduces significant synchronization overhead, which complicates deployment in resource-constrained and latency-sensitive environments like those targeted by our framework. Consequently, we favor pipeline parallelism, which reduces communication overhead and better aligns with our objectives of resilience and operational flexibility. The usual memory and compute constraints of DNNs are further amplified when we consider LLMs, which are infeasible to run without optimizations on standard user equipments (UEs) like smartphones and other edge devices [13].

Recent research addresses these issues with techniques like split learning and inference [14], [15], and the use of potentially idle constrained devices [16], [17]. Our JARVIS framework extends these concepts to tactical environments by dynamically allocating model segments and addressing node failures to maintain continuous operation, as demonstrated in Section IV-A. This approach optimizes the computational footprint and enhances both resilience and flexibility.

C. *Networks for AI*

Traditional AI models trained in centralized data centers face concerns about data confidentiality, integrity, and single points of failure, especially in tactical settings [18]. Centralized models depend on single network links, leading to bandwidth bottlenecks and vulnerabilities. While deploying AI locally can help, LLMs are too large for single edge devices, necessitating a new network architecture for distributed AI across edge devices and mobile equipment [19]. Although emerging 6G networks offer virtual slices to meet AI Quality of Service requirements [20], traditional cellular architectures alone may not be viable in expeditionary environments. JARVIS realizes the vision of 'Networks for AI' by integrating cellular, Wi-Fi, and wired transport to connect edge nodes, enhancing robustness, optimizing data routing, and adapting to network conditions. This supports efficient AI deployment across diverse environments, from urban centers to remote locations, and ensures continuous AI operations.

III. SYSTEM DESIGN AND IMPLEMENTATION

Fig. 1 illustrates the JARVIS architecture, where the layers of the LLM are distributed across a network of both wireless and wired links. Each node running a layer may have different compute and storage capabilities, from handheld devices carried by soldiers to servers on portable 5G base stations. The following sections detail the implementation of this architecture.

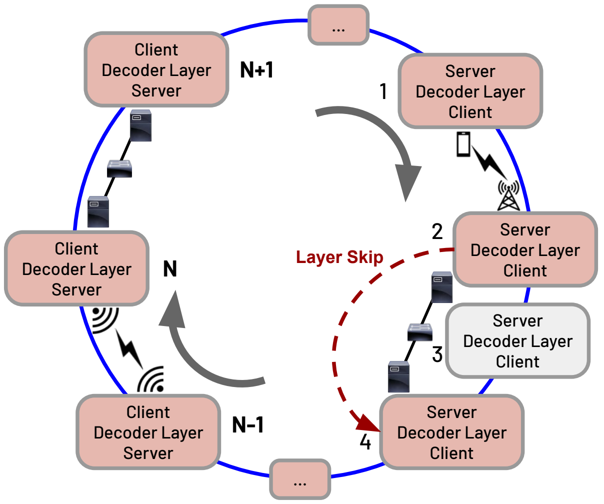


Fig. 2. High-level overview of the client-server application layer's logical architecture. The solid blue line shows a standard inference pass, forming a circular route through nodes, referred to as a *token ring*. Each layer acts as a server to the previous layer and a client to the next. Transmission can occur through various links (wireless, cellular, or wired). In the event of a node failure, such as node 3, traffic is re-routed to ensure continued operation.

A. JARVIS Framework

While various types of virtualized networks are possible, our current implementation of JARVIS distributes the LLM across multiple nodes organized in a *token ring* topology. Since each node hosts a distinct layer of the LLM, generating each output token requires a full forward pass through all nodes, forming a closed cycle. Here, a *layer* refers to an individual component of the LLM distributed across the network, while a *node* is a device or system hosting one or more layers. Nodes are connected to form a token ring topology, ensuring efficient message passing of hidden states and control information.

This framework is designed to optimize resource utilization on edge devices with limited computational capabilities. By distributing different layers of the LLM to various nodes, JARVIS trades local computation for increased peer-level communication. To address scenarios where multiple inferences are requested simultaneously, JARVIS supports batching, allowing nodes to process multiple prompts concurrently by grouping them together into a single forward pass. Overall, JARVIS can be described as a decentralized system for LLM inference on nodes connected over heterogeneous link types in a token ring-like network as illustrated in Fig. 2.

B. Splitting an LLM

We split the LLM into two components: layers at the UE (text tokenizer, embedder, first and last transformer decoder layer, and sampler) and hidden layers on constrained devices over a local network. The UE handles initial feature extraction and final output prediction. A single forward pass requires round-trip communication across the network, as shown in Fig. 3.

In JARVIS, text generation begins at the UE, where the tokenizer converts the input prompt into tokens. These tokens are processed by the embedder and the first transformer decoder layer, producing initial hidden states. These states are then distributed to various nodes for further contextual refinement. The refined states are sent back to the UE, where the sampler selects the next token, repeating until the output sequence is complete. Finally, the tokenizer converts the tokens back into words.

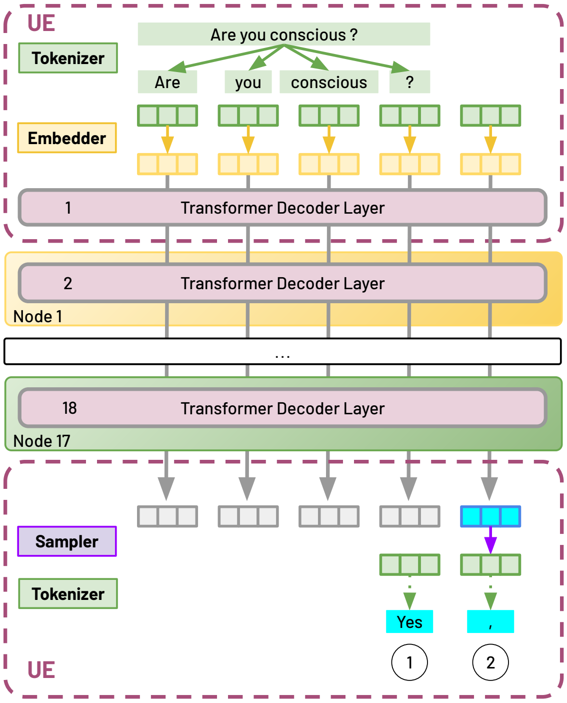


Fig. 3. Forward pass overview of the split LLM architecture in JARVIS. The UE handles initial text processing and final token generation, while intermediate layers are distributed across devices. The figure shows the round trip communication, with hidden states sent from the UE to nodes and refined states returned for final output.

Key structures like caches, positional embeddings, and masks are initialized on both the UE and nodes. The UE sends an initial message to each node with necessary details. The first forward pass involves transmitting the entire input prompt to establish foundational embeddings. Subsequent passes only transmit the last token generated, reducing computational load and data transfer. Each token generation requires a forward pass through all layers, including passing token indexes to update caches and apply the correct positional embeddings and masks.

C. Network design

We designed a simple client-server application layer protocol for transferring information between devices. Each node operates as both a server and a client: node N serves node $N-1$ and sends updates to node $N+1$, as shown in Fig. 2. There are several key systems-level challenges at the transport and application layers that we highlight in this section.

Transport Layer Selection: We chose TCP for reliable, in-order packet delivery, despite its higher round-trip latency, to ensure reliable delivery of hidden states.

TCP Connection Design: We considered two approaches: creating a new TCP connection for each token transfer or maintaining a continuous connection. Although maintaining a continuous connection reduces handshake overhead and optimizes bandwidth through TCP's congestion and flow control algorithms, we opted for the simpler design of creating a new connection for each round trip of hidden states. This choice avoids long-term port reservations and allows for a more granular analysis of network traffic during inter-layer messaging of the LLM, as discussed in Sec. IV-B.

Passing Model Hidden States: The application layer message format was designed to include the tensor meta-data and values. We use a multi-level Python dictionary to transmit information types such as 'KV_index' and 'hidden_state' for a normal inference call or 'batch_size' and 'max_seq_len' for the message to initialize the generation process. Tensors are converted to numpy arrays, then to bytes, and encoded in base64 using utf-8 before serialization using JSON. Data is sent in chunks with an 'END' message signaling completion. On reception, the data is reconstructed, and the tensor is deserialized back to its original type and shape.

D. Node allocation

To demonstrate JARVIS' capabilities, we built an initial framework for deploying different LLM layers to multiple nodes in Colosseum [21], the world's largest wireless network emulator with hardware in-the-loop. JARVIS supports three network transmission types (Fig. 2): a cellular (LTE) link, using the SCOPE framework [22]; Wi-Fi using an 802.11 a/g/p modem with a GNU Radio-based protocol stack [23]; and wired 10 Gbps Ethernet connections.

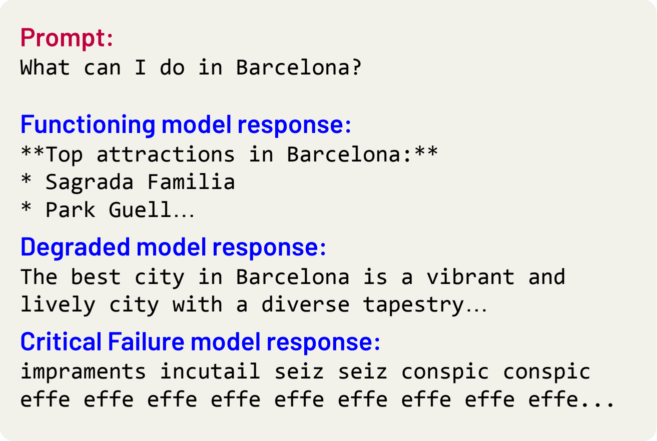
Colosseum provides various wireless channel scenarios for JARVIS deployment, replicating RF conditions of cities like Rome, Boston, and Salt Lake City based on real cellular base station locations [22]. We also use the Alleys of Austin scenario, set in downtown Austin, TX, supporting 50 nodes divided into 5 squads, each with 9 pedestrian users in column formation and a UAV circling above [21].

We created automated tools for rapid deployment of new topologies. Users provide a configuration file listing nodes, network connections, and desired layers. Our setup script deploys each node's networking stack and initial layer weights, allowing flexibility in layer assignment based on computing capabilities. The route-building script detects network connections between nodes and configures interfaces and routing for logical connections.

E. Node failure control

We can test the impact of a node failure using our route-building script. Initially, users can remove a node or layer from the configuration file and re-run the script, which automatically reconfigures JARVIS to route information through the available layers.

Future implementations will automate this process with a centralized node monitoring system that updates the node-layer list and re-runs the route-building tool. Additionally, we are developing a decentralized approach where each layer, L, knows the next two hops. If layer $L+1$ fails to respond, layer L will automatically skip to layer $L+2$.



Prompt:
`What can I do in Barcelona?`

Functioning model response:
```
**Top attractions in Barcelona:**
* Sagrada Familia
* Park Guell...
```

Degraded model response:
```
The best city in Barcelona is a vibrant and
lively city with a diverse tapestry...
```

Critical Failure model response:
```
impraments incutail seiz seiz conspic conspic
effe effe effe effe effe effe effe effe effe...
```

Fig. 4. Example input prompt and model responses with different layer skips. Skipping layer 2 yields a relevant answer, while skipping layers 1 and 8 produces a coherent but incorrect response ('the best city in Barcelona'). Skipping layers 11 and 18 results in repetitive, meaningless output, showing that only certain layer-skipping combinations cause model failure.

IV. Evaluation and Results

A. Model Performance

We successfully split two different LLMs, Gemma 2B and Gemma 7B, and provide results in this section. As described in III-B, the split consists of running the head and tail parts of the model in the UE and the remaining hidden layers on distributed nodes. This adds privacy to the processing of the user prompts at the cost of computing overhead at the edge and added network traffic. Fig. 4 shows various recorded responses of the 2B model to the same prompt with different layers skipped, illustrating varying levels of generation degradation depending on the specific layers skipped.

To test the resilience of LLMs to the layer-skipping phenomenon, we evaluated the accuracy of both Gemma LLMs on the MMLU benchmark [9] by progressively removing one or two layers. The models were tested on 5 example multiple-choice questions and a final question, with the correct answer determined by selecting the highest-scoring option (A, B, C, or D) based on the calculated logits for the next token probability. We first measured the full model's accuracy, then assessed the impact of removing layers individually and in pairs, as shown in Fig. 5. The relative accuracy degradation was calculated by dividing the accuracy reduction after removing specific layers by the full model's accuracy.

The heatmap shows performance degradation for different layer removals, with darker nodes indicating complete failures. Removing the first layer causes a significant accuracy drop, but because it processes the initial prompt, it is assumed always to be available. Single-layer removals, represented along the diagonal, generally cause minimal degradation, with later layers having less impact. However, some middle layers, like

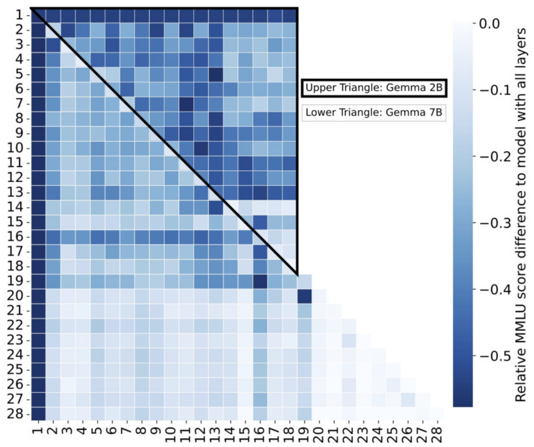


Fig. 5. Heatmap illustrating the relative accuracy difference of the Gemma 2B and 7B LLMs on the MMLU benchmark dataset with layer removals. Accuracy is measured by the number of correct answers the models provide to multiple-choice questions across various topics. The relative accuracy is calculated by dividing each model's performance by the accuracy of the full model. Each index (i, j) in the matrix represents the accuracy difference compared to the full model when layers i and j are removed. Darker squares, such as in column and row 1, indicate complete model failure. The diagonal shows the impact of removing individual layers. It is evident that the first layer and certain middle layers are more critical. Additionally, the larger model (Gemma 7B) appears to be less affected by layer skipping, particularly in the later layers.

11 and 13 in Gemma 2B, are more critical. Two-layer removals reveal non-critical interactions among later layers, while the most severe effects are seen in the middle layers. Notably, the 7B model shows fewer critical interactions from skipping layers than the 2B model, suggesting that the resilience to layer skipping is likely due to the overparameterization of LLMs. Larger models could potentially enhance this resilience further, allowing even more layers to be skipped.

We also assessed the model's text generation for Gemma 2B with layer removals. Removing the first or last layer alone led to critical failure responses, while two-layer removals varied in impact. This indicates system resilience to specific layer failures. Consequently, even with simultaneous node failures, the system may remain functional depending on the layers involved. Important layers should be allocated to secure nodes, and recovery protocols initiated upon node failure. Interestingly, in some cases, letting the model generate the next token freely with certain layer removals even improved the model's performance.

B. Network Performance

To analyze model traffic, we conducted experiments using the Gemma 2B model with input prompts from 10 to 1000 tokens, limiting the generated tokens to 200. This setup ensures consistency with up to 200 forward passes and exchanges between the UE and other layers. The initial context length impacts the initial information transfer, and is followed by the transmission of hidden states for the remaining generated tokens.

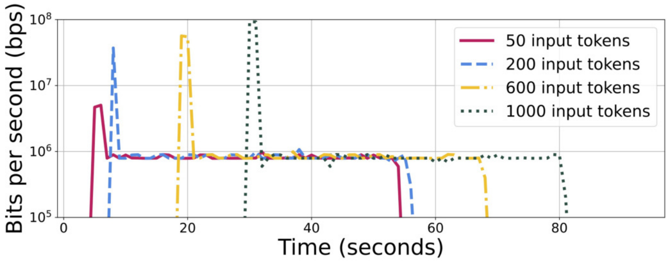


Fig. 6. Throughput for the duration of 4 different experiments, with different prompt lenghts (input tokens). For these generation processes, we only split the UE from a powerful server running the rest of the layers and observe the traffic between them. We observe a large peak when sending the input tokens. As more tokens are sent, the peak is higher and happens later due to more demanding computational effort at the UE. Consequently, the overall inference takes longer and ends later.

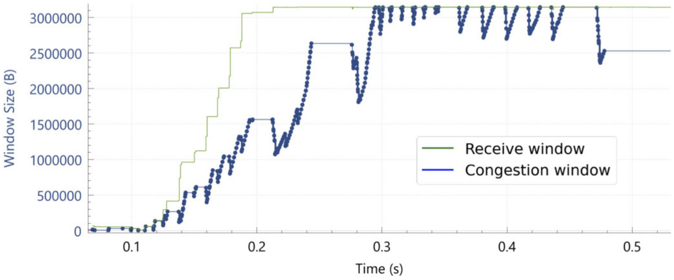


Fig. 7. This figure shows TCP receive and congestion window sizes in bytes during the initial exchange of hidden states for a prompt length of 1000 tokens. TCP window scaling exhibits standard congestion control behavior, with the receive window ultimately limiting throughput at nearly 3,000 kilobytes. This demonstrates that the transmission speed between layers is constrained by the receiving edge device's buffer size.

We examined network traffic between the UE's layer 1 (client) and the next layer (server). The initial hidden state round trip causes a significant spike in network utilization, while subsequent updates maintain consistent throughput (Fig. 6). The input context length affects the initial peak timing and magnitude, as larger input prompts result in larger hidden state dimensions. Larger structures also take longer to process, extending the overall inference time.

Initial Hidden States Update: The initial update of hidden states requires significant data transfer, averaging 11.6 KB per token. For instance, 50 tokens generate an initial message of about 583 KB, causing a brief spike of 5 Mbps, while 1000 tokens result in 11.6 MB and a 90 Mbps spike. TCP window scaling during this transfer shows typical congestion control behavior, with the receive window limiting throughput, seen in Fig. 7.

Incremental Model Updates: After the initial exchange, the client sends the server 199 incremental updates (since output tokens are capped at 200) with a mean size of 12.1 KB, resulting in an average throughput of just under 800 Kbps, shown in Fig. 6. The processing time for the incremental updates will be the same regardless of the context length.

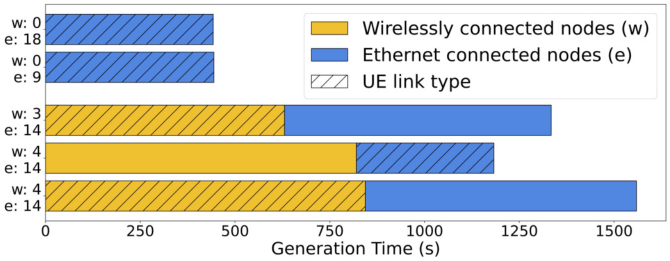


Fig. 8. Comparison of generation times across different configurations in the JARVIS framework, measured from prompt submission to generation of 100 tokens. The y-axis shows various combinations of wireless (w) and Ethernet (e) links. The upper group of bars shows minimal impact from running two layers per device. The lower group highlights that wireless connections significantly increase generation time. Additionally, placing the UE on a wired connection optimizes the overall process. Skipping a layer on a wireless link also notably reduces the total time required for generation.

With a wired connection, we calculated round trip statistics, finding a median exchange time of 54.4 ms, excluding client processing time before the next exchange.

C. Deployment Results

In our deployment of the JARVIS framework, we evaluated the performance and resilience of our distributed LLM system using various configurations, as shown in Fig. 8. The tests aimed to analyze the impact of different node types, network links, and layer distributions on processing time and system resilience. Each configuration generated 100 tokens per inference, roughly equivalent to a third of a standard A4 page of single-spaced text.

The network transmission type of the UE significantly affected generation time. Moving the UE from a wired to a wireless node increased generation time by approximately 377 seconds. We also assessed the effect of layer skipping on generation time, as discussed in Section III-E. Re-routing traffic to the next node saved 225 seconds in generation time, highlighting the overhead introduced by wireless links.

Additionally, we experimented with assigning multiple layers to a single node to reduce the number of devices required. While this approach optimized resource use, it compromised resilience to single-point failures by increasing dependency on fewer nodes. Interestingly, as we were using only wired connections it did not result in significant time savings, as shown in the upper bars of Fig. 8.

V. Conclusion

In this paper we present JARVIS, a novel framework for deploying large language models (LLMs) by distributing them across multiple network nodes and leveraging trusted edge devices with limited computational resources. This approach enhances resilience to node failures through layer skipping, demonstrated using the Gemma LLM on the NSF Colosseum RF emulator. It also enables further recovery methods, such as peer-level communication and layer redundancy. JARVIS allows efficient local execution of LLMs without relying on centralized cloud resources, ensuring robust operation in tactical environments.

Acknowledgment

This work was supported in part by the U.S. National Science Foundation under grants CNS-1925601 and CNS-2112471.

References

[1] W. X. Zhao *et al.*, "A survey of large language models," 2023.

[2] OpenAI *et al.*, "GPT-4 technical report," 2024.

[3] Gemini Team *et al.*, "Gemini: A family of highly capable multimodal models," 2024.

[4] Meta AI, "Introducing llama 3: The most capable openly available llm to date," https://ai.meta.com/blog/meta-llama-3/, 2024, accessed: 2024-06-06.

[5] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," 2023.

[6] Z. Wang, J. Wohlwend, and T. Lei, "Structured pruning of large language models," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[7] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[8] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[9] D. Hendrycks *et al.*, "Measuring massive multitask language understanding," in *ICLR*, 2021.

[10] P. Goyal *et al.*, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 06 2017.

[11] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[12] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[13] Z. Lin, G. Qu, Q. Chen, X. Chen, Z. Chen, and K. Huang, "Pushing large language models to the 6g edge: Vision, challenges, and opportunities," 2024.

[14] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled split deep neural networks for edge-assisted real-time systems," 2019.

[15] P. Li, E. Koyuncu, and H. Seferoglu, "Adaptive and resilient model-distributed inference in edge computing systems," *IEEE Open Journal of the Communications Society*, 2023.

[16] A. Borzunov *et al.*, "Petals: Collaborative inference and fine-tuning of large models," 2023.

[17] S. Ye *et al.*, "Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference," in *IEEE INFOCOM*. IEEE, 2024.

[18] L. Song, X. Hu, G. Zhang, P. Spachos, K. N. Plataniotis, and H. Wu, "Networking systems of AI: On the convergence of computing and communications," *IEEE Internet of Things Journal*, 2022.

[19] J. Pan, L. Cai, S. Yan, and X. S. Shen, "Network for AI and AI for network: Challenges and opportunities for learning-oriented networks," *IEEE Network*, vol. 35, no. 6, 2021.

[20] W. Wu *et al.*, "AI-native network slicing for 6G networks," *IEEE Wireless Communications*, 2022.

[21] L. Bonati *et al.*, "Colosseum: Large-scale wireless experimentation through hardware-in-the-loop network emulation," in *IEEE DySPAN*, 2021.

[22] L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "SCOPE: An open and softwarized prototyping platform for NextG systems," in *MobiSys, Applications, and Services*, 2021.

[23] B. Bloessl, "IEEE 802.11 a/g/p Transceiver," https://github.com/bastibl/gr-ieee802-11, 2024, accessed: 2024-05-31.