

# ISCA 2026

## DynoPipe: Heterogeneous Edge-Cloud LLM Serving with Dynamically Orchestrated Pipeline Boundaries

**Yanying Lin**, Baicheng Chen, Xinyu Zhang, Chengzhong Xu, Kejiang Ye



- Edge hard to host full-scale LLMs (memory wall)
- Full cloud offload is not always acceptable
- Cross-domain execution creates hidden overhead: transfer, sync, pipeline bubbles
- Static boundaries cannot adapt to changing conditions

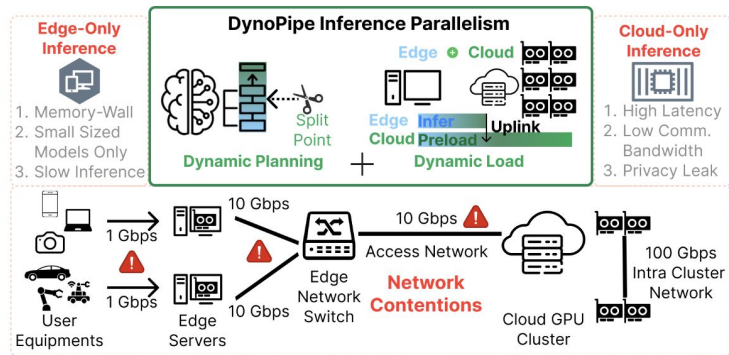
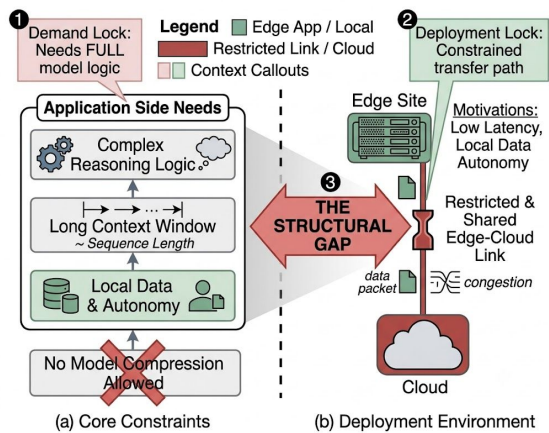
**When edge and cloud must both participate, where should the execution boundary be, and when should it move?**

# Agenda

- Problem: Edge Memory Wall + Cross-Domain Overhead
- Insight: Boundary-Shifting Pipeline Minimizes Critical Path
- Three Challenges: Heterogeneity, Volatility, Migration Cost
- DynoPipe Design: Construction · Orchestration · State
- Evaluation: Critical-Path Reduction, Overlap Efficiency, Tail Latency
- Takeaways

# Background: Edge LLM Serving

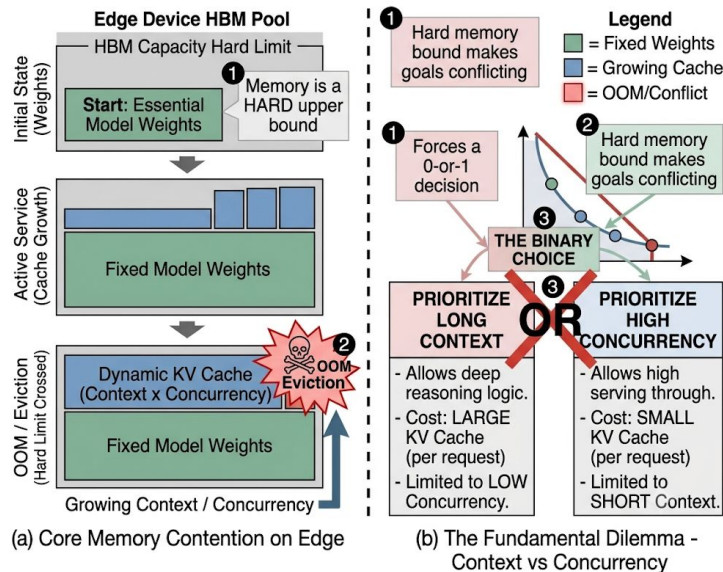
- Driven by latency, autonomy, data-locality needs
- Apps need full capability — complex reasoning, long context — not shrunken models
- Reality: an edge device + a remote cloud, joined by a thin, shared WAN link



Serving full-scale LLMs is moving from the cloud toward the edge

# Problem: The Edge Memory Wall

- Weights + KV cache exceed edge GPU memory → large models undeployable
- Throughput wants big batches & long context; memory forces context-vs-concurrency trade-off
- KV cache grows with context → eviction → stalls to rebuild cache
- Multi-model co-location makes the pressure worse

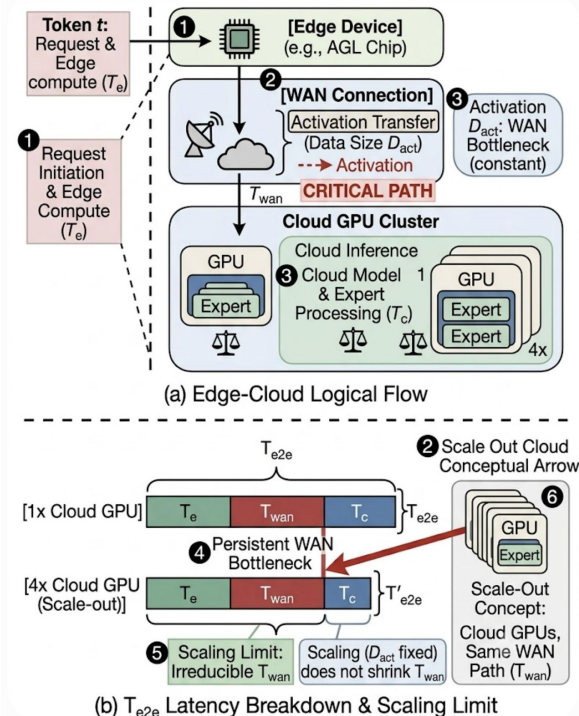


**Edge-only serving cannot fit the model and scale concurrency at the same time.**

# Problem: Cloud Offloading Is Not Free



- Privacy / data locality: raw prompts and context often must stay on the edge
- Full offload exposes user inputs on the public network (healthcare, finance, autonomy)
- When edge must participate, the question is **where** to place execution — not "all cloud vs all edge"
- Fixed full-offload = edge as passive client → no cross-domain pipeline overlap

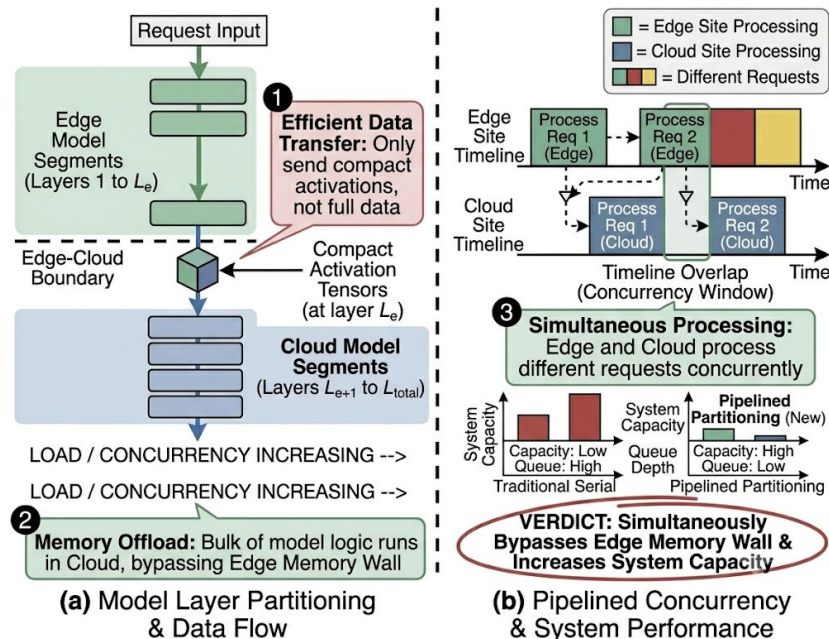


Cloud-only is a valid baseline — but many edge deployments cannot send everything remotely.

# Insight: Pipeline Across the Boundary



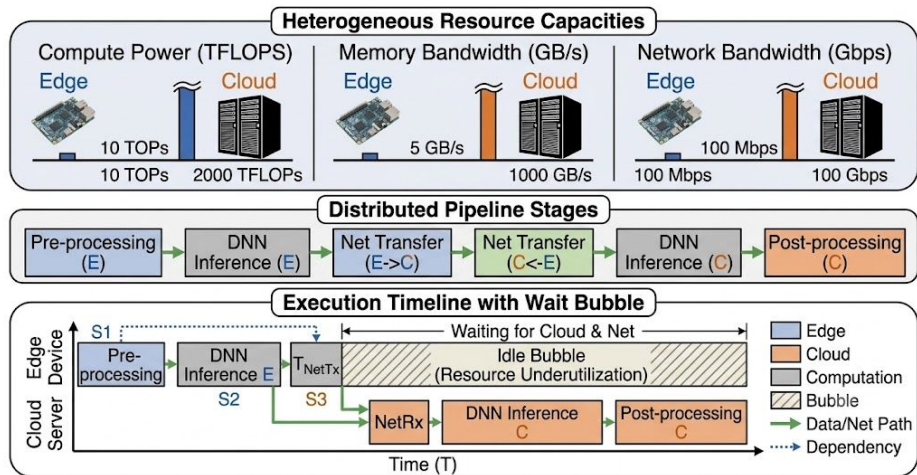
- Assign early layers to edge → no full-model memory requirement
- Assign heavy layers to cloud → leverage elastic compute
- Overlap edge compute and cloud compute for different requests
- Adapt boundary to minimize:  
 $\max(T_{\text{edge}}, T_{\text{cloud}}) + T_{\text{transfer}}$



**Don't just offload — split: keep sensitive inputs local, overlap edge and cloud execution.**

# Challenge 1: Structural Heterogeneity

- Edge: slower compute, narrower memory bandwidth, weaker & higher-latency network
- Constraints are coupled: compute imbalance amplifies memory pressure, which kills comm overlap
- Operators hit unevenly (attention vs FFN) → irregular bubbles that simple balancing can't fix



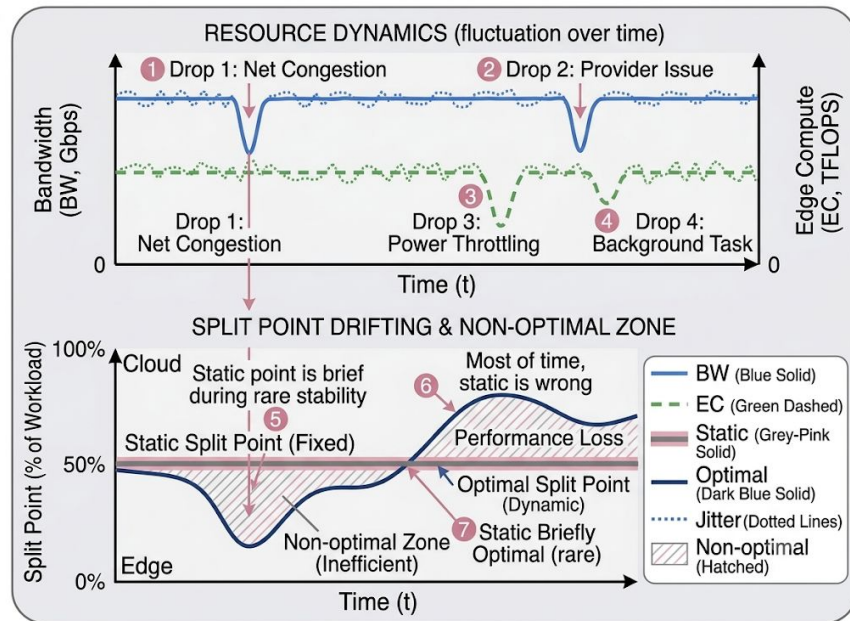
**Edge vs cloud are asymmetric across compute, memory, and network — and the asymmetries compound**

# Challenge 2: Temporal Drift of the Critical Path



- Bandwidth contention, thermal throttling, network-compute coupling
- Non-linear: small fluctuations trigger threshold effects → sudden shifts
- Optimal split moves by several stages within minutes
- Any offline-fixed partition becomes mismatched

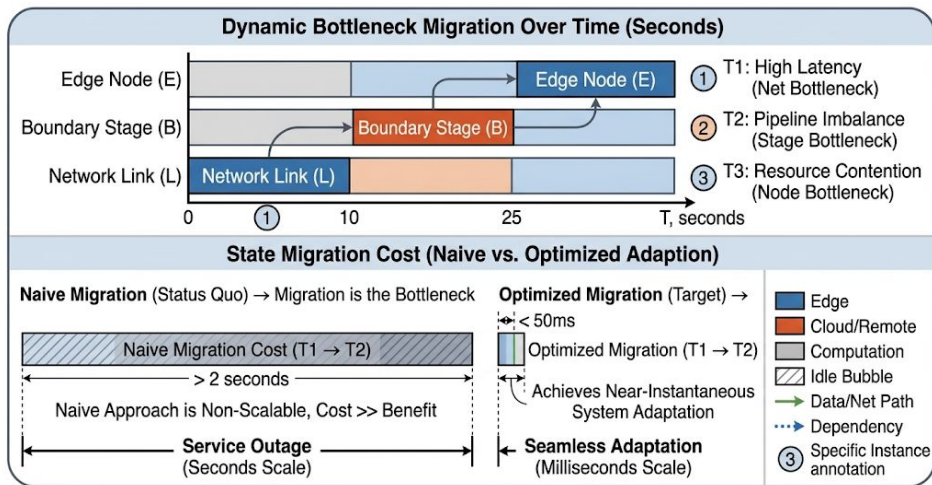
**The dominant stage on the critical path shifts over time — any fixed boundary becomes suboptimal**



# Challenge 3: Migration Overhead



- Boundary change transfers execution ownership between domains
- KV cache must follow execution: cross-domain transfer of accumulated state
- Naive migration takes seconds → service disruption
- Adaptation only helps when migration cost is amortized



**Adapting the boundary means migrating state — and migration overhead can exceed the benefit**

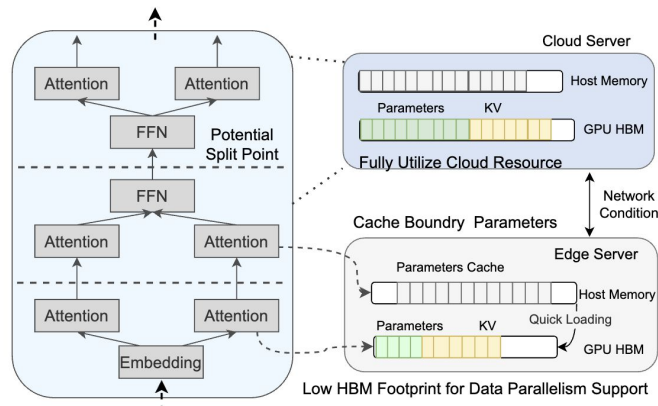
- Structural heterogeneity → Boundary-Constrained Construction
- Temporal volatility → Proactive Orchestration
- Migration overhead → Hierarchical State Management

# Design 1: Boundary-Aware Construction



- Objective: min over stages of (exec + boundary overhead)
- Boundary overhead = serialize + cross-domain transfer + state sync
- Split points between transformer blocks; DP over the search space
- Output: configs for bandwidth/ compute/ memory-bound regimes

**Model cross-boundary cost explicitly; solve splits by DP**

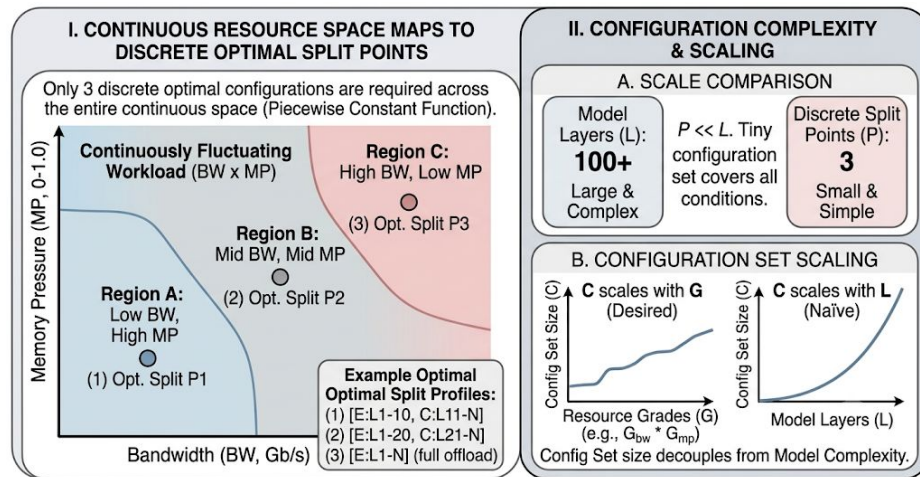


# Design 1 (cont.): Why a Tiny Portfolio



- Resources continuous, optimal split discrete  $\rightarrow$  few cells
- Portfolio size grows with resource regimes, not model depth
- Uniform blocks  $\rightarrow$  strong collapse  $\rightarrow$  few configs suffice

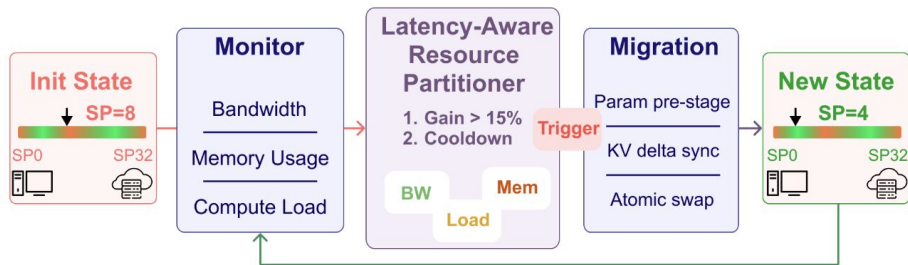
**Optimal split is  
piecewise-constant in resource  
space**



## Design 2: Proactive Orchestration (LRP)

- LRP objective:  $\min L = \max(T_{\text{edge}}, T_{\text{cloud}}) + T_{\text{transfer}} + T_{\text{sync}}$
- Bottleneck-aware selection: transfer-bound  $\rightarrow$  minimize activation size; edge-bound  $\rightarrow$  earlier offload; sync-bound  $\rightarrow$  minimize state transfer
- Hysteresis + cooldown  $\rightarrow$  no oscillation
- $O(|K|)$  lookup, not re-solve  $\rightarrow$  sub-ms decision

Pick a config at runtime by  
bottleneck-aware lookup — fast,  
stable

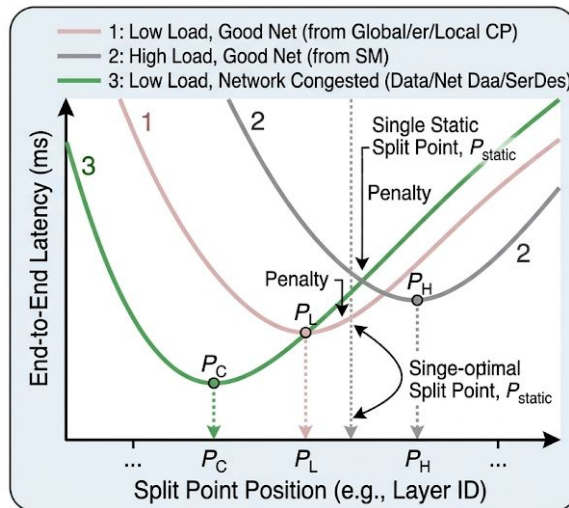


# Why the Optimal Split Moves

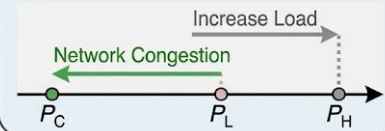


- Latency = queue + edge + network + cloud
- Low load  $\rightarrow$  cloud-heavy split
- Near capacity  $\rightarrow$  balanced split
- Network contention  $\rightarrow$  earlier split

**No static split wins across load and network**



## Optimal Split Point Drift Indicator



## Optimal Configuration Table

| Workload Condition      | Optimal Split ( $P^*$ ) | Reason                 |
|-------------------------|-------------------------|------------------------|
| Low Load, Good Net      | Balanced ( $P_L$ )      | Utilizes HBM & NoC     |
| High Load, Good Net     | Cloud-leaning ( $P_H$ ) | Cloud GPU needed       |
| Low Load, Congested Net | Edge-leaning ( $P_C$ )  | Minimize Data Transfer |

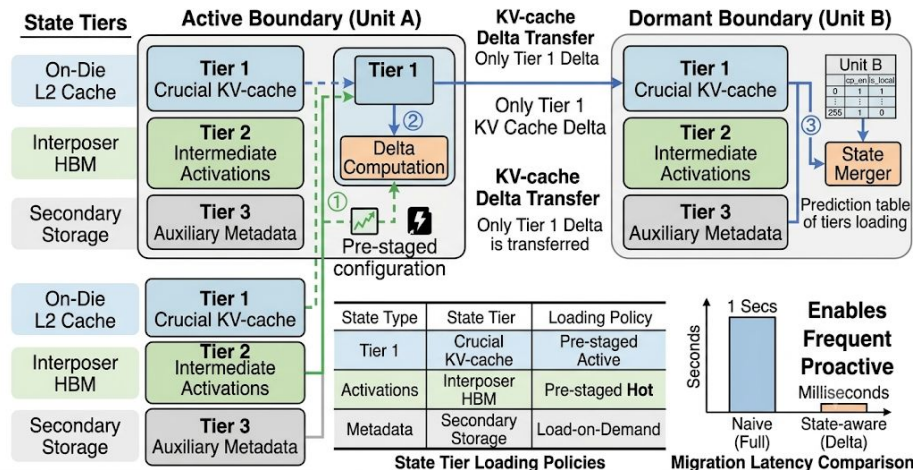
**No Single Optimal.** Static points cause severe penalties under changing conditions.

# Design 3: Hierarchical State Management



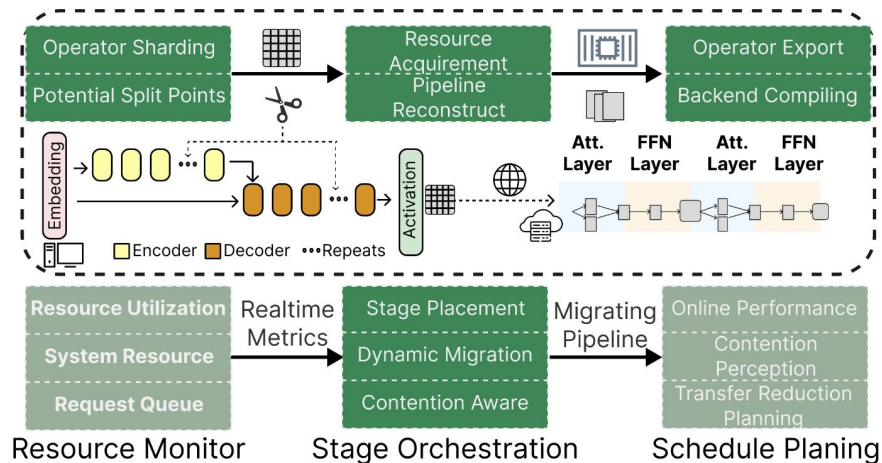
- State tiers: KV cache / activations / metadata
- 3-level memory: prestage hot configs, kill cold-start
- Transfer KV deltas only; recomputation fallback at low BW

**Predictive staging + differential transfer → migration in milliseconds**



# System Overview

- Offline: profile  $\rightarrow$  DP  $\rightarrow$  small config portfolio
- Online: telemetry  $\rightarrow$  pick config by table lookup
- State layer: keep KV cache continuity across switches
- Flow: edge stage  $\rightarrow$  activation  $\rightarrow$  cloud stage



**Precompute the hard part offline;  
cheap lookups online.**

- Cloud: 100 Gbps IB
- Edge: constrained WAN uplink
- Workload: real Azure Functions trace (bursty) + Splitwise params
- Baselines: FlexNN (edge), EdgeShard (static), Cloud-only
- Metrics: TTFT, throughput, E2E latency (mean / P50 / P99)

# Result 1: System Performance



- FlexNN unusable beyond moderate QPS; DynoPipe sub-second throughout
- vs FlexNN: 98.5% TTFT cut, 4.4x throughput
- Scales to 70B (edge-only cannot deploy at all — memory wall)
- 10.1x over edge-only, 1.6x over cloud-only, 99.2% wait-latency cut

TABLE I: Performance of DYNOPipe and FlexNN in LLaMA3.1-8B.

| QPS | Queue (s) |        | TTFT (s) |        | Tput (tok/s) |        |
|-----|-----------|--------|----------|--------|--------------|--------|
|     | DynoPipe  | FlexNN | DynoPipe | FlexNN | DynoPipe     | FlexNN |
| 2   | 0.30      | 43.86  | 0.10     | 19.68  | 922          | 277    |
| 4   | 0.40      | 55.78  | 0.47     | 27.03  | 1118         | 291    |
| 8   | 0.91      | 110.06 | 0.74     | 68.53  | 1327         | 304    |
| 16  | 1.25      | -      | 1.56     | -      | 1857         | -      |
| 24  | 1.49      | -      | 2.56     | -      | 2471         | -      |

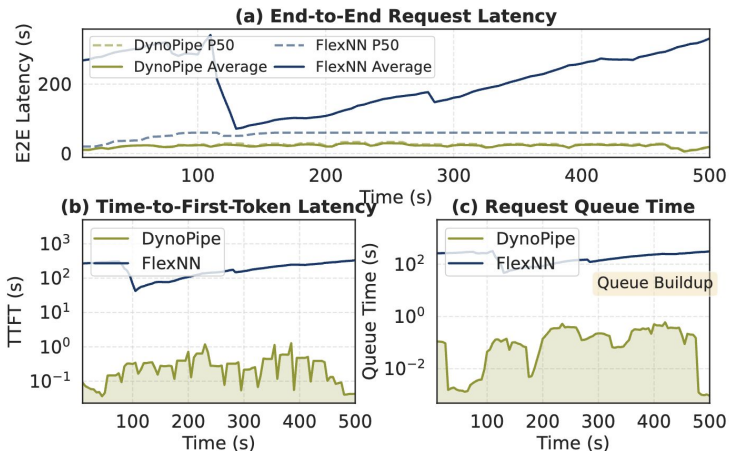
QPS: query/second.

Config: Temp=0.7, max out=4096 toks, max in=16392 toks, timeout=5s.

TABLE II: Performance in different scale of models.

| Model       | Act.(M) | Type   | Throughput |          | TPOT (ms) |          |
|-------------|---------|--------|------------|----------|-----------|----------|
|             |         |        | FlexNN     | DynoPipe | FlexNN    | DynoPipe |
| Whisper-V2  | 0.32    | Speech | 1545       | 3900     | 30        | 36       |
| LLaMA2-7B   | 1.2     | Chat   | 263        | 2645     | 54        | 73       |
| LLaMA3.1-8B | 1.45    | Chat   | 290        | 1857     | 68        | 65       |
| Qwen-14B-VL | 2.1     | MM     | -          | 1640     | -         | 84       |
| Qwen-32B-VL | 4.8     | MM     | -          | 1309     | -         | 134      |
| LLaMA3-70B  | 8.2     | MM     | -          | 830      | -         | 160      |

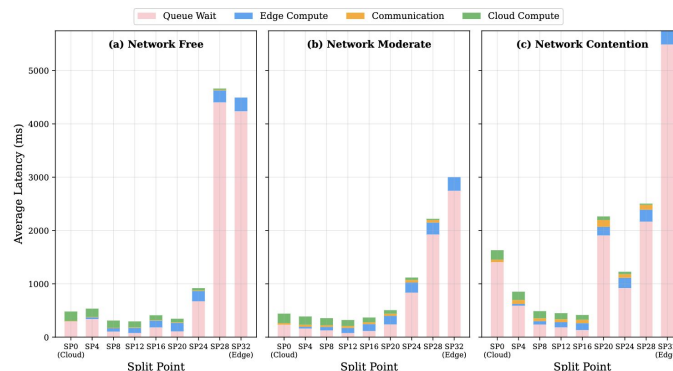
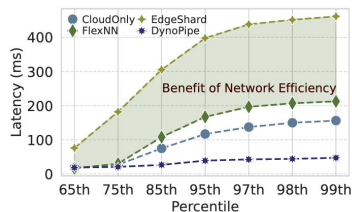
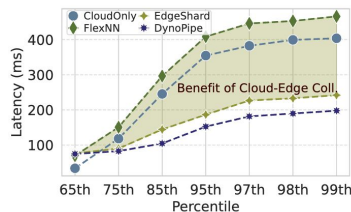
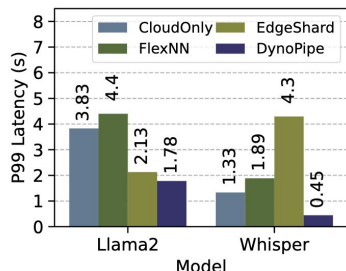
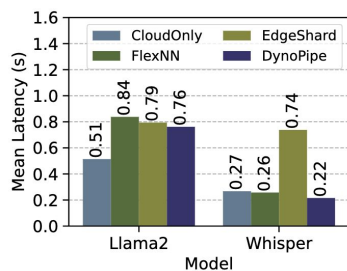
TPOT (Time Per Output Token) is calculated as the average decode time. MM refers to the multimodal model using LLM as the backbone. - indicates that the model cannot be deployed on the edge due to memory constraints.



# Result 2/3: Tail Latency + Why It Works



- Shorter critical path: overlap between edge compute and cloud compute
- Reduced transfer amplification: boundary placed at activation-minimal layers
- Dynamic adaptation: 4-config portfolio covers all regimes, zero gap
- Migration practical: tens-of-ms; output lossless (split at block boundaries)





- Three co-designed modules: construction · orchestration · state management
- Breaks edge memory wall + cloud serialization at once
- Robust under volatility; gains from capacity, not extra resources
- Move the boundary, not the model

# Q&A

yy.lin1@siat.ac.cn

<https://yanyinglin.github.io>