RL-PARETO: Performance-Aware Routing and Hybrid PPO–DQN Orchestration for Parallelized Service Function Chains

Akshit Kumar[$], Venkatarami Reddy Chintapalli[⋆], Bheemarjuna Reddy Tamma[•], and C. Siva Ram Murthy[•]
[$]National Institute of Technology Calicut, India [⋆]National Institute of Technology Warangal, India
[•] Indian Institute of Technology Hyderabad, India
e-mail:akshit_b210779cs@nitc.ac.in, venkatarami@nitw.ac.in, tbr@iith.ac.in, murthy@cse.iith.ac.in

***Abstract*—Emerging latency-critical applications such as cloud gaming and industrial automation demand agile and ultra-low-latency service delivery, which traditional network appliances struggle to support. Network Function Virtualization (NFV) addresses this by chaining Virtual Network Functions (VNFs) into Service Function Chains (SFCs). Parallelized SFCs (PSFCs) reduce service delay by executing independent VNFs concurrently, but introduce significant copy/merge and buffering overheads due to synchronization delays across branches. Moreover, dynamic PSFC arrival rates complicate efficient VNF placement decisions. This paper presents *RL-PARETO*, a hybrid deep reinforcement learning approach that adaptively orchestrates parallel VNFs while minimizing parallelization overheads and satisfying SLA constraints. *RL-PARETO* uses a graph transformer encoder with dual pointer-network heads to jointly generate PSFC partitions and VNF placements in a single pass. Training integrates Proximal Policy Optimization (PPO) for stable exploration with a Double-DQN critic for efficient value estimation. A fallback heuristic ensures feasible deployments under resource constraints. Extensive evaluations across diverse network topologies demonstrate that *RL-PARETO* achieves up to 10% higher acceptance rate and 15% reduction in merge buffer overhead, while maintaining robust performance under dynamic conditions.**

***Index Terms*—Network Function Virtualization, Parallelized Service Function Chain Placement, Reinforcement Learning, Graph Transformer, Proximal Policy Optimization, Double DQN**

I. Introduction

TODAY'S networked applications—cloud gaming, augmented reality, and automated industrial control—demand two critical qualities: *ultra-low latency* and *on-demand adaptability*. Traditional hardware appliances for firewalls, intrusion detection, and load balancing struggle to meet these requirements. *Network Function Virtualization* (NFV) addresses this challenge by implementing such functions as software modules (as Virtual Network Functions (VNFs)) on commodity servers. NFV substantially reduces capital and operational expenses and enables service deployment in minutes rather than months [1]. In an NFV environment, traffic traverses a *Service Function Chain* (SFC), visiting each VNF sequentially. Each hop adds processing and forwarding delay, potentially violating stringent Service-Level Agreements (SLAs) for real-time flows. To mitigate this, independent VNFs of an SFC can process packets in parallel, forming a *Parallelized SFC* (PSFC) [2]. For example, packet inspection, filtering, and logging VNFs may process packets of the same flow simultaneously instead of sequentially in traditional SFC (as shown in Fig. 1) [3], [4]. Prior works show that up to 41.5% of VNFs can be parallelized (forming PSFC), reducing end-to-end delay by approximately 40% [3]. However, PSFCs introduce two new sources of overhead: *(i) Copy/Merge Overhead:* Each packet must be duplicated by the copy module for parallel processing VNFs and later re-aggregated by using the merge module, which incurs additional delay. *(ii) Buffering Overhead [5]:* In PSFCs, parallel VNF execution creates multiple branches with varying completion times, requiring buffering at the merge module to synchronize packet streams. More delay differences across branches increase buffer occupancy, making it a critical factor in parallel VNF deployment decisions.

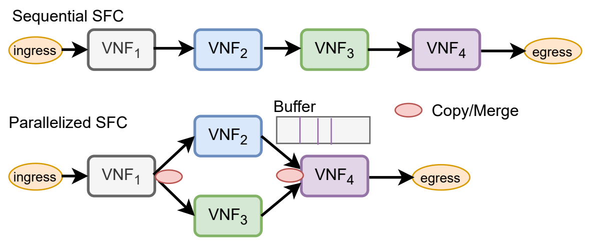


Fig. 1: Traditional sequential SFC vs. Parallelized SFC.

In recent years, several mechanisms have been proposed to enhance the performance of PSFC deployments. For instance, [6] introduces a scoring-based embedding method that clusters independent VNFs to balance workload and reduce latency, while [7] develops a flexible model that captures interactions between delay, resources and traffic, leveraging VNF sharing to cut costs. Reinforcement Learning (RL) has also been applied: NFVdeep [8] uses a Markov Decision Process to adapt to network variations, and [9] proposes an A3C-based strategy for latency-aware VNF parallelization under uncertain resource demands. Despite these advances, two major issues remain largely unaddressed: (i) the additional overheads of VNF parallelization (copy/merge delays and buffering for branch synchronization), and (ii) the dynamic nature of service arrivals, which requires adaptive, real-time placement decisions.

To address these gaps, we propose *RL-PARETO*, a deep reinforcement learning framework for adaptive orchestration of parallel VNFs. It aims to meet SLA constraints while minimizing copy/merge and buffering overheads under dynamic PSFC arrivals. Unlike heuristics that struggle with changing conditions, *RL-PARETO* offers a self-tuning approach. For each PSFC request, it evaluates available CPU, memory, and bandwidth along with delay deadlines, then (i) partitions the chain into parallel branches, and (ii) places VNFs while avoiding resource-deficient servers. If no feasible placement exists, a fallback heuristic—Service Instance Placement Algorithm (*SIPA*)—is used. *SIPA* first minimizes the delay in the critical branch and then balances other branches to satisfy the SLA constraints. *RL-PARETO* employs a Graph Transformer encoder to jointly capture node capacities, link delays, and PSFC structures. Dual pointer-network heads generate partitioning and placement decisions in one step. Training combines Proximal Policy Optimization (PPO) for stable exploration with a Double-DQN critic for efficient value refinement. This hybrid RL design enables real-time adaptation to network dynamics, with SIPA ensuring robust performance under strict SLAs. The main contributions of this paper are as follows.

- We propose a graph-based encoder with dual pointer-network heads that jointly model node/link capacities and PSFC split structures to generate partitions and placements in one shot.
- We integrate PPO for stable on-policy learning with a Double-DQN critic to enhance sample efficiency, optimizing PSFC service acceptance, latency, and buffering overhead simultaneously.
- We propose a heuristic that guarantees feasible embeddings under strict SLA conditions by greedily placing VNFs on RL-generated splits, ensuring bounded worst-case regret.
- The proposed approach achieves up to 10% higher service acceptance and 15% reduction in merge buffer overhead across various real-time network topologies.

II. System Model and Problem Formulation

A. Substrate Network

We represent the physical compute–network topology as an undirected graph $G = (V, E)$, where the node set V (with $|V| = N$) comprises of servers and the link set $E \subseteq V \times V$ (with $|E| = L$) captures bidirectional fronthaul/backhaul connections. The topology is encoded by the adjacency matrix $A \in \{0,1\}^{N\times N}$, where $A_{uv} = 1$ exactly when $(u, v) \in E$. Compute resources at node n are indexed in time as $C_n(t)$, denoting available CPU units, while each link (u, v) provides residual bandwidth $\mathrm{BW}_{uv}(t)$ and can have the propagation latency ℓ_{uv}. We assume that traffic between any two VNFs can traverse one of the K precomputed shortest-latency paths $\{\mathcal{P}^k_{u\to v}\}_{k=1}^K$; selecting path $\mathcal{P}$ both reduces each link's bandwidth by the flow demand and contributes $\sum_{(i,j)\in\mathcal{P}} \ell_{ij}$ to end-to-end delay. The NFV infrastructure supports a finite set $F = \{F_1, \ldots, F_{N_f}\}$ of VNF types. Instantiating VNF F_k at time t with service rate $r_s(t)$ consumes compute effort $f^c_k(t) = \eta_k\, r_s(t)$, where η_k is a per-unit cost. A node n can host multiple VNFs so long as

$$\sum_{i\in \text{VNFs on } n} f^c_i(t) \;\le\; C_n(t). \tag{1}$$

B. PSFC Model and MDP State Representation

PSFC model is defined as the ordered tuple $\langle \mathrm{PE}_0, \mathrm{PE}_1, \ldots, \mathrm{PE}_n\rangle$ of $n + 1$ Parallel Entities (PEs) (assuming each PE has either one VNF or multiple parallelizable VNFs), with exactly one parallel stage at index $a \in \{0, \ldots, n\}$ that splits the workflow into $B_a \in \mathbb{N}$ identical branches. Fig. 2 illustrates the special case $n = 3$, $a = 1$, so the flow departs PE_0 (i.e., source (*src*)), splits at PE_1 into $B_a = B$ branches, and then reconverges at PE_3 before reaching PE_4 (i.e., destination (*Dest*)). At each time t, the state of the system s_t embeds all dynamic quantities: the residual compute capacity of each node $C_u(t)$, the remaining bandwidth of each link $\mathrm{BW}_{uv}(t)$, the external arrival rate of PSFC λ_r, the service level deadline T_r, the branch count B_a, and the traffic rate of each branch $r_{s,b}(t)$.

At each decision epoch, the agent first partitions the remaining chain of VNFs at the split stage a, assigning each function i to one of the B_a streams. Instantiating VNF i on branch b at node u consumes $f^c_{i,b}(t) = \eta_i\, r_{s,b}(t)$, where $\eta_i > 0$ is the per-unit compute cost; this placement is feasible only if $\sum_{i,b} f^c_{i,b}(t) \le C_u(t)$ at every u. Then, for each branch b the agent selects a placement node u and one of the K precomputed shortest-latency paths $P^{(k)}_{u\to v}$. Routing along $P^{(k)}$ deducts $r_{s,b}(t)$ from each link's $\mathrm{BW}_{uv}(t)$ and incurs propagation delay $\sum_{(u,v)\in P^{(k)}} \ell_{uv}$, where ℓ_{uv} is the per-link latency. Once all B_a branches complete their assigned functions and converge at PE_3, the system incurs a fixed copy delay T_{copy} at the split junction (PE_1), a fixed recombine delay $T_{\text{merge}}/T_{\text{recombine}}$ at the join node (PE_3), and any buffer overhead due to early finish branches. Key notations and symbols used in this paper are listed in Table I.

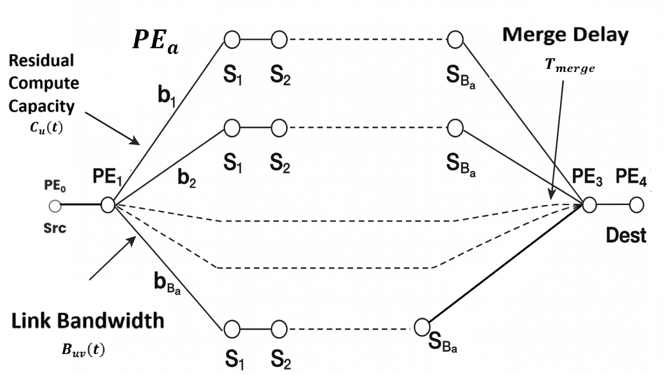


Fig. 2: PSFC model.

C. MDP Formalization

We model PSFC placement as a finite-horizon MDP $M = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of actions, P is the transition function, R is the reward function, and $\gamma = 1$ is the discount factor. The aim of this MDP is to decide both how to split the flow at a designated stage and where to place each VNF so that the PSFC meets resource limits and delay requirements. At each decision step t, the current state is defined by

$$s_t = \big(G_t,\ [PE_0, \dots, PE_n],\ \lambda_r,\ T_r,\ i_t,\ b_t,\ \sigma_t\big). \quad (2)$$

Here, $G_t = (V, E, \{c_u(t)\}, \{\mathrm{BW}_{uv}(t)\})$ is the residual substrate graph. Each node u retains the compute capacity $c_u(t)$, and each link (u, v) retains the bandwidth $\mathrm{BW}_{uv}(t)$. The ordered list $[PE_0, \dots, PE_n]$ represents the sequence of VNFs, with the index a marking the split stage (for parallel VNFs). At stage a, the packets from a PSFC flow are copied into B_a parallel branches. Parameters λ_r and T_r denote the arrival rate of PSFC (in requests per second) and the SLA of end-to-end delay, respectively. The tuple (i_t, b_t, σ_t) is the placement cursor: i_t is the index of the next VNF to place, b_t is the branch number when $i_t = a$ (otherwise $b_t = 1$), and σ_t is the slot within that branch. From state s_t, an action a_t is either a partition decision or a placement decision. A partition decision assigns each of the PE_a instances to one of the B_a branches, ensuring that the compute load $\sum_i \eta_i \lambda_r$ for each branch fits within the chosen nodes. A placement decision selects a node $u \in V$ and one of its K precomputed shortest-latency paths $\mathcal{P}$, and places VNF PE_{i_t} for branch b_t provided that

$$c_u(t) \ \geq\ \eta_{i_t} \lambda_r \quad \text{and} \quad \exists\, \mathcal{P} \text{ s.t. } \mathrm{BW}_{xy}(t) \geq \lambda_r\ \forall\, (x, y) \in \mathcal{P}. \quad (3)$$

If neither a partition nor a placement is feasible, the SIPA heuristic (explained in the next section) is invoked as a fallback, and the episode immediately terminates with a large negative penalty. When an action is executed, the transition is deterministic. If the action is a partition, the substrate graph does not change ($G_{t+1} = G_t$) and the cursor resets to ($i_{t+1} = a$, $b_{t+1} = 1$, $\sigma_{t+1} = 1$). If the action is a placement on node u, then for every link (x, y) on the chosen path $\mathcal{P}$ the resources are updated as

$$\begin{aligned} C_u(t+1) &= C_u(t) - \eta_{i_t}\, r_{s,b}(t), \\ \mathrm{BW}_{xy}(t+1) &= \mathrm{BW}_{xy}(t) - r_{s,b}(t). \end{aligned} \quad (4)$$

After the update, the cursor advances to the next VNF or branch. The episode ends when all VNFs across all branches have been placed or when the SIPA fallback is triggered. The reward function gives zero reward at every non-terminal step. At termination, the reward is the following.

$$\begin{aligned} R = -\Big[(B_a - 1)\,(T_{\mathrm{copy}} + T_{\mathrm{merge}}) + \max_b D_b \\ + \sum_{b=1}^{B_a}\big(\max_b D_b - D_b\big)\Big] - M\, \mathbf{1}_{\mathrm{infeasible}}. \end{aligned} \quad (5)$$

where D_b is the total delay (including processing and path delay) experienced by branch b, T_{copy} and T_{merge} are fixed copy and merge delays, and M is a large constant penalty applied if any action was infeasible or a SIPA fallback occurred.

TABLE I: Notation and Symbol Definitions

Symbol	Definition
$[PE_0, \dots, PE_n]$	Ordered PSFC of $n+1$ VNFs; index a is the split stage.
B_a	Number of parallel branches at stage a.
$r_{s,b}(t)$	Service rate of branch b at time t.
η_i	Per-unit compute cost of VNF PE_i.
$f^c_{i,b}(t)$	Compute demand of PE_i on branch b: $\eta_i\, r_{s,b}(t)$.
(i_t, b_t, σ_t)	Placement cursor: VNF index i_t, branch b_t, slot σ_t.
D_b	Total delay of branch b (processing + path).
$x^{(0)}_u, e^{(0)}_{uv}$	Initial node/edge features for Graph-Transformer.
$h^{(\ell)}_u, q^{(\ell)}_u, k^{(\ell)}_u, v^{(\ell)}_u$	Query/key/value embeddings at layer ℓ.
$\alpha^{(\ell)}_{uv}$	Attention coefficient at layer ℓ.
$z^{(0)}, W_m, b_m$	Meta-token embedding and projection.
$H^{(0)}$	Pooled graph embedding after L layers.
ℓ_{part}	Raw logits for partition choices.
$\pi_{\mathrm{part}}(\pi_k \mid s_t)$	Softmax probability over partitions.
q_u	Pointer score for node u.
$M_t[u]$	Feasibility mask for node u.
$\pi(u \mid s_t)$	Masked-softmax for placement.
θ, ϕ	Actor (policy) and critic parameters in PPO.
$V_\phi(s_t)$	Critic's value estimate.
A_t	Advantage estimate.
$\epsilon, c_1, c_2, N_{\mathrm{ppo}}$	PPO hyperparameters: clip, coefficients, epochs.
ψ^+, ψ^-	Online and target Q-network parameters for Double-DQN.
$y_i, \delta_i, L_{\mathrm{DQN}}$	TD target, error, and DQN loss.
$\alpha, \beta, \varepsilon$	PER exponents and small constant.
$N_{\mathrm{DQN}}, U_Q, \tau$	Replay size, update interval, soft-update factor.
D, T, G_0	Replay buffer, episode horizon, return $G_0 = \sum r_t$.
$\mathrm{lr}_q, \mathrm{lr}_p$, batch_size,grad_clip	Learning rates, batch size, gradient clipping.
sync_every	Target-network synchronization frequency.

III. Reinforcement Learning Policy and Training

A. From MDP to RL: Motivation

The PSFC placement MDP must decide, at each step, whether to partition a split stage into B_a parallel branches or to place a specific VNF instance on one of the N compute nodes, subject to residual compute capacities $C_u(t)$, link bandwidths $\mathrm{BW}_{uv}(t)$, and a strict end-to-end latency SLA T_r. This yields an extremely large, structured action space where naive enumeration or scalar function approximation fails to capture both the local resource constraints and the global graph topology. Moreover, the sequential nature of split–recombine chains imposes long-horizon dependencies: an early placement choice can dramatically affect downstream feasibility and cumulative delay. Reinforcement learning offers a natural remedy, but only if the policy network can (a) embed heterogeneous node/link features, (b) respect action feasibility masks, and (c) generalize across varying PSFC lengths and substrate graphs. These requirements motivate our choice of a Graph Transformer encoder, capable of learning high-dimensional node embeddings through multi-head self-attention, and a dual pointer network decoder that cleanly issues either partition or placement decisions under dynamic feasibility constraints. In the next subsection, we detail how this architecture concretely implements $\pi(a \mid s)$ for our MDP.

B. Graph-Transformer & Pointer-Network Policy

The PSFC placement MDP defined demands a policy that can jointly reason about local resource constraints: residual computation $C_u(t)$ at each node and bandwidth $\mathrm{BW}_{uv}(t)$ at each link and the global network topology to satisfy a strict end-to-end delay limit T_r. To this end, we propose a single neural module, POLICY-NETWORK, which comprises a multilayer Graph Transformer encoder and a dual-head pointer network decoder. This architecture naturally mirrors the two action types in our MDP, partitioning the split stage and placing individual VNFs, while embedding all dynamic state information into attention-based representations. Algorithm 1 provides the pseudocode.

Algorithm 1 PolicyNetwork

```
procedure POLICYNETWORK($s_t$)
    $\{c_u(t), \{\mathrm{BW}_{uv}(t)\}, PE\_cursor\} \leftarrow s_t$
    // 1. Node  edge feature projection
    for all nodes $u$ do
        $x_u \leftarrow [\mathrm{onehot}(PE\_mask_u) \,\|\, c_u(t)/C_{\max}]$
        $h_u^{(0)} \leftarrow W_x x_u + b_x$
    end for
    for all edges $(u, v)$ do
        $e_{uv}^{(0)} \leftarrow W_e[\ell_{uv}/\ell_{\max} \,\|\, \mathrm{BW}_{uv}(t)/\mathrm{BW}_{\max}] + b_e$
    end for
    $m^{(0)} \leftarrow W_m[PE\_cursor, \lambda_r/T_{\max}, T_r/T_{\max}] + b_m$
    // 2. Graph Transformer encoder
    for $\ell = 0$ to $L - 1$ do
        for all nodes $u$ do
            $q_u, k_u, v_u \leftarrow W_Q h_u^{(\ell)},\ W_K h_u^{(\ell)},\ W_V h_u^{(\ell)}$
        end for
        for all nodes $u$ do
            $h_u^{(\ell+1)} \leftarrow \mathrm{LayerNorm}\big(h_u^{(\ell)} + \sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(\ell)} v_v\big)$
        end for
    end for
    // 3. Pooling and context
    $g \leftarrow \frac{1}{|V|} \sum_u h_u^{(L)},\ c \leftarrow [g \| m^{(0)}]$
    if $i_t = a$ then                                          ▷ split-stage
        // 4a. Partition head
        $\ell_{\text{part}} \leftarrow W_p c + b_p$
        $\pi_{\text{part}}[i] = \dfrac{\exp(\ell_{\text{part}}[i])}{\sum_j \exp(\ell_{\text{part}}[j])}$        ▷ Partition head softmax
        return sample($\pi_{\text{part}}$)
    else
        // 4b. Pointer head
        for all nodes $u$ do
            if $c_u(t) \geq \eta_{i_t} \lambda_r$ and feasible path then
                $s_u \leftarrow h_u^{(L)\top}(W_{ptr} c + b_{ptr})$
            else
                $s_u \leftarrow -\infty$
            end if
        end for
        $\pi(u) = \dfrac{\exp(s_u)}{\sum_v \exp(s_v)}$        ▷ Placement head softmax
        if $\sum_u \pi(u) > 0$ then
            return sample($\pi$)
        else
            return SIPA_Fallback($s_t$)
        end if
    end if
end procedure
```

In the first stage, we embed the substrate and SLA context. Each node u is represented by a one-hot mask of remaining VNF types and its normalized residual compute $c_u(t)/C_{\max}$. Each edge (u, v) is featurized by normalized propagation latency $\ell_{uv}/\ell_{\max}$ and bandwidth $\mathrm{BW}_{uv}(t)/\mathrm{BW}_{\max}$. We also construct a global meta-token from the placement cursor (i_t, b_t, σ_t) and the request parameters (λ_r, T_r). By projecting these inputs into initial embeddings, the network attains precise knowledge of available resources and remaining delay budget. The second stage employs L layers of self-attention on the substrate graph. At each layer ℓ, every node computes query, key and value vectors from its current embedding and attends over its neighbors with an additive bias derived from the edge features. This mechanism enables the policy to trade off, for instance, choosing a slightly more distant node if its links offer higher bandwidth, or avoiding a nearby node whose links are congested. Skip-connections and layer normalization ensure stable training, and inclusion of the meta-token in every attention layer propagates SLA and placement progress information across the entire graph.

In the third stage, we combine the final node embeddings into a summary vector g and concatenate the meta-token to form a context vector c. If the next VNF index i_t corresponds to the split stage, a partition head projects c into logits over all valid splits of the split-stage VNFs into B_a branches, applies a masked softmax to enforce capacity constraints, and samples the chosen assignment. Otherwise, a pointer head computes a score for each node by taking the dot product of its final embedding with a learned query vector, masks out nodes that violate $c_u(t) \geq \eta_{i_t}\lambda_r$ or lack any feasible K-shortest path, and then samples or greedily selects the host. In the rare case that no placement is feasible, we invoke the deterministic *SIPA* fallback (Section III-C) to guarantee a valid embedding or graceful reject. By embedding every dynamic quantity from the MDP state into the self-attention mechanism and structuring the decoder to mirror precisely the MDP's action space, this Graph-Transformer + Pointer-Network policy achieves both expressive global reasoning and strict adherence to the hard resource and latency constraints of the PSFC placement problem.

C. Deterministic SIPA Fallback

When the learned graph transformer policy cannot find any feasible placement for a required VNF instance, i.e. no node u satisfies both $c_u(t) \geq \eta_{i_t}\lambda_r$ and a K-shortest path with $\mathrm{BW} \geq \lambda_r$ – we invoke the *SIPA* heuristic (Algorithm 2) as a guaranteed safety net. This fallback solves the same PSFC embedding problem under the assumption that a finite pool of VNF instances already exists on the substrate and that each instance can be selected exactly once. The *SIPA* procedure begins by decomposing the PSFC into a critical branch and non-critical branches (*phase I*), where the critical branch is the one whose earliest failure would most likely cause an SLA violation. During *phase IIa*, it greedily embeds the critical branch: for each VNF in sequence, it scans all candidate instances and their zero-hop or precomputed K-shortest paths, selects the combination minimizing processing and link delays, and then updates the residual bandwidth of each traversed link. This preserves the exact resource update semantics of our MDP transitions. In *phase IIb*, *SIPA* handles each non-critical branch by building a layered graph instance path (via the subroutine INST_ASSIGN), performing a depth-first enumeration of all feasible placement sequences, and filtering out any whose total delay plus the already accumulated critical branch delay would exceed the SLA T_r. Among the remaining candidates, it chooses the one whose delay is best synchronized with the critical path (minimizing the buffer overhead) and updates the link resources accordingly.

Algorithm 2 SIPA Heuristic Fallback

```
Input: PSFC chain delays $r$, instance map $D$, precomputed $k\_paths$, substrate $G$,
  source/sink ($src$, $dst$), rate $\lambda_r$, SLA $\Delta$, overheads $d_{cm}, d_{dep}$
Output: Assignments or REQUESTREJECTED
(branches, $crit\_idx$) $\leftarrow$ PSFC_STRUCT($r$); $te2e \leftarrow 0$; $prev \leftarrow src$
for all vnf in branches[crit_idx] do                ▷ Phase I: critical branch
    $best \leftarrow (\infty, \text{None}, \emptyset)$
    for all inst in $D[vnf]$ do
        consider zero-hop: $d \leftarrow inst.proc\_delay$
        update $best$ if $d$ smaller
        for all path in $k\_paths[(prev, inst.host)]$ do
            $d \leftarrow inst.proc\_delay + \sum_{(u,v)\in path} G[u][v].delay$
            update $best$ if $d$ smaller
        end for
    end for
    if $best.inst$ is None then throw REQUESTREJECTED
    end if
    reserve $\lambda_r$ on $best.path$ and on $best.inst.capacity$
    record $crit\_assign[vnf] \leftarrow best$
    $prev \leftarrow best.path[\text{end}],\ te2e \mathrel{+}= best.delay$
end for
for all $b \neq crit\_idx$ in branches do             ▷ Phase II: noncritical
    $best\_delta \leftarrow \infty,\ best\_seq \leftarrow$ None
    for all seq in INST_ASSIGN(k_paths,b,G) do
        $d \leftarrow \sum_{(inst,p)\in seq}\big(inst.proc\_delay + \sum_{(u,v)\in p} G[u][v].delay\big)$
        if $d \leq \Delta - d_{dep}$ then
            $\delta \leftarrow |(te2e + d_{cm}) - d|$
            if $\delta < best\_delta$ then update $(best\_delta, best\_seq) \leftarrow (\delta, seq)$
            end if
        end if
    end for
    if $best\_seq$ is None then throw REQUESTREJECTED
    end if
    reserve $\lambda_r$ on all instances and links in $best\_seq$
    $noncrit\_assign[b] \leftarrow best\_seq$
end for
// Phase III: finalize
$theta \leftarrow \max(\max delay(crit\_assign), \max_b delay(noncrit\_assign[b]))$
$te2e \mathrel{+}= (B_a - 1)d_{cm} + \frac{\lambda_r}{\lambda_{\max}} \sum_b (theta - delay)$
if $te2e > \Delta$ then throw REQUESTREJECTED
end if
return $(crit\_assign, noncrit\_assign, te2e)$
```

Finally, *SIPA* computes the split recombine copy / merge overhead $(B_a - 1)\, d_{cm}$ and the stochastic deposition delay term, adds them to the branch delay of the worst case and rejects the request if the resulting end-to-end delay exceeds T_r. By mirroring the MDP's reward-termination checks and resource constraints, this deterministic heuristic ensures that every PSFC either receives a valid embedding—albeit possibly suboptimal—or is gracefully rejected with the same infeasibility penalty that the learned policy would incur. This fallback enables one to maintain strict correctness: the graph transformer policy handles the vast majority of embeddings, and only in rare corner cases does *SIPA* intervene, leveraging its efficient two-phase greedy strategy to guarantee feasibility under the assumptions of the original problem.

D. Hybrid PPO & Double-DQN Training

To learn a robust policy for PSFC placement MDP, we combine Proximal Policy Optimization (PPO) on-policy with off-policy Double DQN updates in a unified training loop (Algorithm 3). PPO provides stable and monotonic policy improvement by maximizing a clipped surrogate objective over the trajectories collected under the current policy, while DoubleDQN leverages a replay buffer and temporal-difference bootstrapping to accelerate value learning and reduce sample complexity. This hybrid design takes advantage of the complementary strengths of the policy gradient and value-based methods to address the high-dimensional, sequential nature of our placement problem.

Algorithm 3 Hybrid PPO & Double-DQN Training

```
procedure HYBRIDTRAINING($\theta, \phi, \psi^+, \psi^-$)
    Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$, $E \leftarrow 0$
    while not converged do
        // Collect on-policy trajectories
        $\mathcal{T} \leftarrow \emptyset$
        for $e = 1$ to $N_{\text{env}}$ do
            $s \leftarrow$ env.reset()
            for $t = 0$ to $T - 1$ do
                $a \sim \pi_\theta(\cdot \mid s)$ or FALLBACK
                $(s', r, done) \leftarrow$ env.step($a$)
                Append $(s, a, r, s')$ to $\mathcal{T}$
                if $a \neq$ FALLBACK then
                    Compute TD-error $\delta = r + \gamma\, Q_{\psi^-}(s', a') - Q_{\psi^+}(s, a)$
                    Store $(s, a, r, s')$ in $\mathcal{D}$ with priority $|\delta| + \varepsilon$
                end if
                if done then break
                end if
                $s \leftarrow s'$
            end for
            $E \mathrel{+}= 1$
        end for
        // PPO updates
        Compute advantages $A_t = R_T - V_\phi(s_t)$ for all $(s_t, a_t)$ in $\mathcal{T}$
        for $k = 1$ to $K_{\text{ppo}}$ do
            Sample batch from $\mathcal{T}$
            Compute clipped loss $L_{\text{PPO}}$ and update $\theta, \phi$
        end for
        // DQN updates every $U_Q$ episodes
        if $E \bmod U_Q = 0$ and $|\mathcal{D}| \geq N_{\text{dqn}}$ then
            for $k = 1$ to $K_{\text{dqn}}$ do
                Sample $(s_i, a_i, r_i, s_i')$ from $\mathcal{D}$ with PER weights
                $y_i = r_i + \gamma\, Q_{\psi^-}(s_i', \arg\max Q_{\psi^+}(s_i', \cdot))$
                Update $\psi^+$ by minimizing $\big(Q_{\psi^+}(s_i, a_i) - y_i\big)^2$
                Update priorities $p_i \leftarrow |Q_{\psi^+}(s_i, a_i) - y_i| + \varepsilon$
            end for
            $\psi^- \leftarrow \tau\, \psi^+ + (1 - \tau)\, \psi^-$
        end if
    end while
end procedure
```

Concretely, in each outer iteration we first collect N_{env} on-policy episodes by rolling out π_θ in the PSFC environment. At each step, the actor either emits a partition or placement action via the Graph-Transformer policy, or if no feasible host exists, triggers the SIPA fallback. All transitions (s_t, a_t, r_t, s_{t+1}) are stored in a temporary trajectory buffer $\mathcal{T}$. In parallel, any non-fallback step is added to a prioritized replay buffer $\mathcal{D}$ with priority proportional to its TD-error under the current online Q-network Q_{ψ^+}. Next, we compute advantages $A_t = R_T - V_\phi(s_t)$ for each on-policy transition (where R_T is the terminal reward) and perform K_{ppo} epochs of PPO updates. Each minibatch from $\mathcal{T}$ is used to optimize the clipped surrogate loss

$$L_{\text{clip}} = \mathbb{E}\big[\min(\rho_t A_t,\ \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)A_t)\big],$$

Together with a regression term for the function of value and an entropy bonus, we refine the parameters of the actor θ and the parameters of the critic ϕ. In every U_Q episodes, we interleave double-DQN updates using transitions sampled from $\mathcal{D}$ via Prioritized Experience Replay. For each sampled tuple (s_i, a_i, r_i, s_i'), we compute the target

$$y_i = r_i + \gamma\, Q_{\psi^-}\big(s_i', \arg\max_v Q_{\psi^+}(s_i', v)\big)$$

and minimize squared TD-error $\big(Q_{\psi^+}(s_i, a_i) - y_i\big)^2$, updating the online Q-network ψ^+ and periodically soft-syncing the target network $\psi^- \leftarrow \tau\,\psi^+ + (1-\tau)\,\psi^-$. The prioritized sampling weights and PER-exponents α, β further focus learning on transitions with high informational content.

By interleaving PPO's on-policy gradients with off-policy DQN bootstrapping, our hybrid learner achieves both the stability required for policy improvement in long-horizon, constrained decision processes and the sample efficiency needed to explore a vast placement action space. This training directly addresses the combinatorial and sequential challenges of the PSFC problem, producing policies that consistently satisfy latency SLAs and resource constraints.

IV. Experimental Evaluation

To assess the effectiveness of the proposed scheme, we developed a C++ based simulator. The *RL-PARETO* scheme is tested on two backbone network topologies: *NSFNET* (14 nodes, 21 links) and *USNET* (24 nodes, 43 links). The simulation parameters, listed in Table II, are used as default values unless otherwise specified.

TABLE II: Simulation parameters

Parameter	Value	Parameter	Value
PSFC length	4–8	Branch count B	2–4
Arrival rate(λ_r)	30 req/s	SLA bound	network diameter
Link delay	U[4,8] ms	VNF capacity	5 Gbps
Copy/merge delay	4 ms	Shortest paths K	3
VNF Proc. Delay	DU$[1, 2]$ ms	PSFC Traffic Rate	5Gbps

A. Benchmark Approaches

We evaluate *RL-PARETO* against three benchmarks: *(i) SIPA*, a heuristic that maps VNFs in two phases—first minimizing delay on the critical branch, then balancing others to satisfy SLA deadlines; *(ii) Pure PPO*, an on-policy actor–critic model trained with the PPO clipped-surrogate objective and no fallback; *(iii) Pure DQN*, an off-policy Double-DQN with prioritized replay, trained to minimize temporal-difference error. Performance is assessed using three metrics: *(i) acceptance ratio*, the fraction of PSFC requests embedded within SLA bounds; *(ii) end-to-end delay*, the total latency including link propagation, VNF processing, and copy/merge delays; *(iii) packet deposition*, the buffering required to synchronize parallel branches before merging.

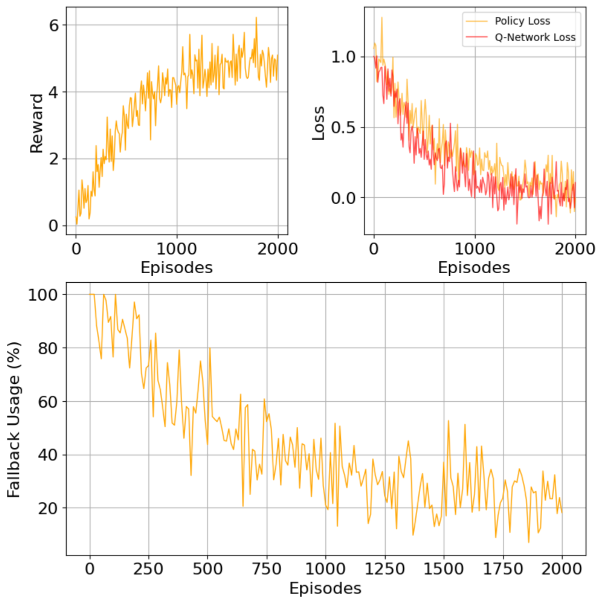


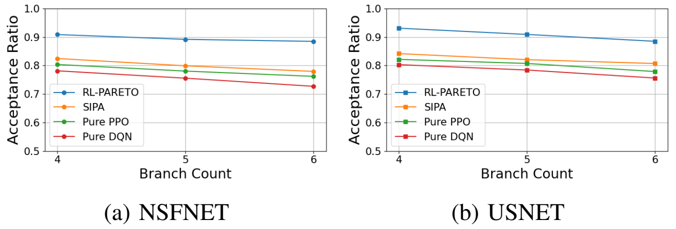
Fig. 3: Training dynamics: episodic reward (top-left), PPO surrogate loss and Double-DQN TD error (top-right), and SIPA-fallback usage fraction (bottom).

B. Training Dynamics

Fig. 3 illustrates how the *RL-PARETO* policy matures. During training, episodic rewards are low and erratic, reflecting random exploration; By episode 2000 they rise to around 5 and stabilize, showing that the agent has learned to balance latency, copy/merge overhead, and buffering. At the same time, the PPO surrogate loss and the Double-DQN temporal-difference error both decline to near zero, indicating convergence of actor and critic networks despite sampling noise. Finally, reliance on *SIPA* fallback drops from nearly 100% to below 20%, confirming that the learned policy has become sufficiently reliable to replace the heuristic in most decisions. Together, these trends demonstrate that within 2000 episodes, *RL-PARETO* acquires a high-quality placement strategy while maintaining feasibility through its safety-net fallback.

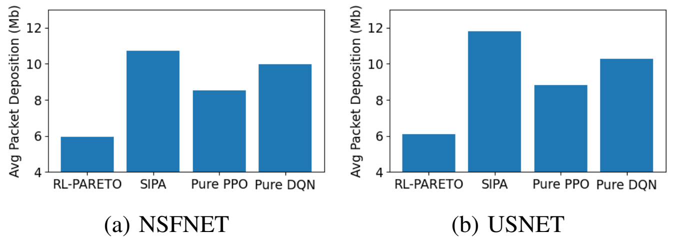
C. High Branching & Chain-Length Stress

We evaluated the impact of increasing parallelism (increasing number of parallel branches of a PFSC request) by fixing all parameters to their defaults, except for the branch count B, which we sweep over $\{4, 5, 6\}$. Each setting is tested on *NSFNET* and *USNET* network topologies with 50 PSFCs per run. *RL-PARETO*s acceptance ratio gently falls from 0.91 at $B = 4$ to 0.88 at $B = 6$, whereas *SIPA* drops from 0.82 to 0.78 and *Pure-PPO/DQN* lags by 11–14% for *NSFNET* topology (Fig. 4a). The similar trends are observed for *USNET* network topology also (Fig. 4b): *RL-PARETO* maintains 0.93→0.89, *SIPA* 0.84→0.80, and *Pure-PPO/DQN* remain 10–12% behind. These results show that the hybrid policy (i.e., *RL-PARETO*) scales gracefully with branch count, while other approaches suffer from combinatorial explosion.



(a) NSFNET (b) USNET

Fig. 4: Acceptance ratio vs. branch count B.



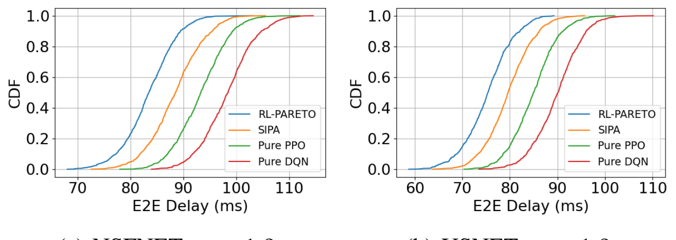
(a) NSFNET (b) USNET

Fig. 5: Average packet deposition vs. branch count B.

Packet deposition under high branching (Fig. 5) is halved by *RL-PARETO*: on *NSFNET*, buffers drop from 11.0 ± 0.3 Mb (compared to *SIPA*) to 5.5 ± 0.2 Mb; on *USNET*, from 11.5 ± 0.3 Mb to 5.8 ± 0.2 Mb. By jointly optimizing partitioning and placement, the learned policy aligns branch completion times and avoids the large skews that *SIPA*'s one-shot heuristic cannot anticipate.

D. Tight SLA Analysis

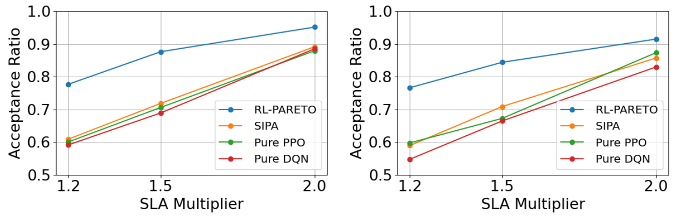
We evaluated each approach under increasing strict delay limits by varying the SLA multiplier $\alpha \in \{1.2, 1.5, 2.0\}\times$ shortest-path delay, with $L = 8$, $B = 4$, $\lambda = 30$ req/sec and other defaults as in Table II. The results presented are the average of 50 runs. At $\alpha = 1.2$, only 60% of *SIPA* placements meet the SLA versus 75% for *RL-PARETO*; *pure PPO/DQN* achieves 58% and 55%, respectively, for *NSFNET* topology (Fig. 6). For *USNET* network topology, *RL-PARETO* reaches 78% acceptance ratio compared to 62% for *SIPA* and around 69% for *pure PPO/DQN* approaches.



(a) NSFNET, $\alpha = 1.2$ (b) USNET, $\alpha = 1.2$

Fig. 6: End-to-end delay CDF at $\alpha = 1.2$.

Fig. 7 shows that in $\alpha \in \{1.2, 1.5, 2.0\}$, the acceptance ratio of *RL-PARETO* consistently outperforms *SIPA* by 8–10% on both topologies, while *pure PPO/DQN* never closes the gap. This shows that combining on-policy exploration, off-policy value refinement, and heuristic fallback yields tighter SLA compliance.



(a) USNET (b) NSFNET

Fig. 7: Acceptance ratio vs. SLA multiplier on both topologies.

V. Conclusions and Future Work

In this paper, we presented *RL-PARETO*, a hybrid deep reinforcement learning approach for the efficient deployment of PSFCs. *RL-PARETO* combines the strengths of PPO and Double-DQN for stable policy learning and value refinement, along with a SIPA-based heuristic to ensure feasible placements under strict SLA constraints. Through comprehensive evaluations in real network backbone topologies (*NSFNET*, *USNET*), *RL-PARETO* consistently outperformed benchmark approaches in both acceptance rate and merge buffer overhead, while maintaining low end-to-end latency and effectively adapting to dynamic network conditions. The framework demonstrates its robustness by invoking the *SIPA* fallback only when necessary, particularly under tight merge and link constraints. In the future, we (i) extend *RL-PARETO* to address multi-objective goals such as energy efficiency, reliability, and operational cost and (ii) focus on enhancing the model's generalization through advanced graph encoders like temporal or heterogeneous transformers and deploying *RL-PARETO* in real-time testbeds with online learning support.

Acknowledgment

This research work was supported by the Science and Engineering Research Board, New Delhi, India. Grant number: JBR/2021/000005.

References

[1] F. Xincai et al., "Paving the way for nfv acceleration: A taxonomy, survey and future directions," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–42, 2020.

[2] T. Fengsen et al., "Joint vnf parallelization and deployment in mobile edge networks," *IEEE Transactions on Wireless Communications*, vol. 22, no. 11, pp. 8185–8199, 2023.

[3] S. Chen et al., "Nfp: Enabling network function parallelism in nfv," in *Proceedings of ACM SIGCOMM*, 2017, pp. 43–56.

[4] Z. Yang et al., "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *Proceedings of ACM Symposium on SDN Research (SOSR)*, 2017.

[5] C. Jun et al., "Appm: adaptive parallel processing mechanism for service function chains," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1540–1555, 2021.

[6] L. Kate et al., "Vnf embedding and assignment for network function parallelism," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1006–1016, 2021.

[7] Z. Chenlu et al., "Service deployment for parallelized function chains considering traffic-dependent delay," *IEEE Transactions on Network and Service Management*, vol. 21, no. 2, pp. 2266–2286, 2023.

[8] X. Yikai et al., "Nfvdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *Proceedings of IEEE/ACM international symposium on quality of service*, 2019, pp. 1–10.

[9] Z. Dongliang et al., "Towards deploying parallelized service function chains under dynamic resource request in multi-access edge computing," *IEEE Transactions on Network and Service Management*, vol. 22, no. 2, pp. 1899–1910, 2025.