

DistServe: Splitting Prefill and Decode onto Different GPUs

DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Zhong, Liu, Chen, Hu, Zhu, Liu, Jin, Zhang · Peking University + UC San Diego + StepFun · OSDI '24 · arXiv:2401.09670 · Day 3

Words you need first (read this once, then the rest is easy)

Token

A chunk of text, roughly $\frac{3}{4}$ of a word. The model reads and writes tokens, not letters.

Prefill

Phase 1. The model reads your whole question at once and produces the **first** word of the answer. Reading 500 tokens happens in **one** big step, all in parallel.

Decode

Phase 2. The model writes the rest of the answer **one word at a time**. Each word needs its own trip through the model. 200 output words = 200 separate steps.

KV cache

Scratch notes the model keeps about every token it has already seen, so it doesn't re-read the whole conversation for every new word. It lives in GPU memory and it is **big** — about 1.13 GB for a single 512-token request on a 66-billion-parameter model.

TTFT — Time To First Token

How long from you hitting Enter until the **first** word appears. This is prefill's report card. You feel it as "did it hang?"

TPOT — Time Per Output Token

Average gap between one word and the next while the answer streams. This is decode's report card. You feel it as "is it typing smoothly?"

Careful: the Sarathi-Serve paper calls the exact same thing **TBT** (time-between-tokens). Same metric, two names — interviewers use both.

SLO — Service Level Objective

A promise you make about speed. Example: "TTFT under 0.25 seconds and TPOT under 0.1 seconds."

SLO attainment

The **percentage** of requests that actually kept the promise. This paper targets 90%.

Throughput vs. Goodput ← the most important distinction in this paper

Throughput = how many tokens per second you produce, counting everything.

Goodput = how many requests per second you finish **while keeping the promise**.

A pizza shop making 100 pizzas/hour has great throughput. If 60 arrive cold and late, its goodput is 40. Customers only pay for goodput.

Per-GPU goodput = goodput ÷ number of GPUs, which is really just "cost per query" upside down.

Compute-bound

The GPU's math units are the bottleneck — it's doing arithmetic as fast as it physically can. Prefill is like this.

Memory-bound

The GPU's math units are mostly idle, waiting for data to arrive from memory. Decode is like this: it drags the entire multi-gigabyte model out of memory just to produce one token.

Intra-op parallelism (a.k.a. tensor parallelism / TP)

"Intra" = *inside*. Split each individual math operation across several GPUs, so 4 GPUs each do a quarter of the same multiplication. **Makes one request finish faster**, but the GPUs must constantly talk to each other, so it needs a very fast cable between them.

Inter-op parallelism (a.k.a. pipeline parallelism / PP)

"Inter" = *between*. Give GPU 1 the first 20 layers, GPU 2 the next 20, and pass work down the line like an assembly line. **One request isn't faster** (it still visits every station), but you can have several requests in flight at different stations, so you handle **more requests per second**. Barely any chatter between GPUs, so it's fine over slow networks.

Instance

One complete copy of the model plus the GPUs holding it. Could be 1 GPU or 12.

Replication

Making more identical copies of an instance to handle more traffic. The boring way to scale.

Execution time vs. queuing delay

Execution time = how long your job takes once it's actually running. Queuing delay = how long it sat in line waiting for a turn. **Latency = both added together**, and people forget the second one constantly.

NVLink / InfiniBand

Fast cables. NVLink connects GPUs **inside one machine** (~600 GB/s between A100s — extremely fast). InfiniBand connects **separate machines** (~800 Gbps — fast, but roughly 6× slower than NVLink, and many clusters have far worse). Ordinary ethernet is much slower still: this paper's own test cluster only had 25 Gbps between machines.

Continuous batching

The standard trick today: don't wait for a whole batch to finish; let requests join and leave the batch after every single step. It's what vLLM does, and it's what mixes prefill and decode work into the same batch — which is exactly the thing this paper attacks.

Where this sits in the series. Day 1 (PagedAttention) fixed *memory*. Day 2 (Sarathi-Serve) fixed *scheduling*, by chopping prefill into small pieces so it stops freezing decode — but it kept both jobs on the same GPU. Day 3 asks: **why are they on the same GPU at all?** Put prefill on some GPUs and decode on others. This is the rival answer to Day 2, and knowing both sides of the argument is the actual interview asset.

Problem

Prefill and decode are two totally different jobs sharing one machine

Plain version: Imagine one kitchen with one oven. Prefill is baking a wedding cake — it fills the oven completely, takes a long time, and baking two cakes at once helps nobody. Decode is toasting bread — each slice takes seconds and barely uses the oven, so you should toast 200 slices at once. Today we do both in the same oven, alternating. Every time a cake goes in, all the toast waits. Every time you cram toast in beside the cake, the cake bakes slower.

	Prefill	Decode
Work per step	All prompt tokens at once	Exactly 1 token per request
Bottleneck	Compute-bound (math units maxed)	Memory-bound (math units idle)
Does batching help?	No. One 512-token request already saturates an A100 on a 13B model	Yes, enormously. It's the only way to get any utilization
Graded on	TTFT	TPOT
Wants which parallelism?	Depends on load — see below	Depends on batch size — see below

The number that motivates the entire paper

13B model, one A100, prompts of 512 tokens, outputs of 64 tokens, 90% SLO attainment:

Setup	Goodput (requests/sec)
Today's systems, both phases on one GPU	1.6
A GPU doing <i>only</i> prefill	5.6
A GPU doing <i>only</i> decode	10.0
2 GPUs for prefill + 1 GPU for decode	10 total = 3.3 per GPU → 2.1× better

Same hardware. Same model. The only change is who does what. **Mixing the two jobs was costing more than half the machine.**

Why mixing hurts: two separate damages

Damage 1 — interference

Add a **single** prefill job to a batch of decode jobs and both get measurably slower. The decodes must wait for the longer prefill to finish (TPOT goes up), and the prefill is slowed by carrying the decodes (TTFT goes up). The longer the prompt, the worse it gets. And you can't dodge it by running them one after another instead of together — then decode jobs just sit in line behind prefill jobs, and the delay reappears as queuing instead of execution. **Prioritizing either phase simply moves the pain to the other one.**

Damage 2 — coupling (the subtler one, and the one interviewers like)

If both phases live on the same GPUs, they are forced to share **the same number of GPUs** and **the same parallelism plan**. But they want different things. So you end up configuring for whichever SLO is harder to hit, and over-buying hardware to drag the other one along. Disaggregating doesn't just remove interference — it lets you **tune each phase separately**, which turns out to matter just as much.

Their direct attack on chunked prefill (Day 2's answer)

DistServe argues Sarathi-style chunking **reduces** interference but cannot **eliminate** it, and that tuning the chunk size is a trap:

- **Make chunks small** → the prefill never gets the GPU to itself, so each prefill takes longer. TTFT suffers.
- **Make chunks big** → the batch fills up with prefill tokens and there's no room left to piggyback decode tokens. The whole benefit evaporates.
- **Either way you pay extra memory traffic.** Chunk 3 must re-read the KV cache of chunks 1 and 2. Split a prefill into N chunks and you read $N + (N-1) + \dots + 1 = O(N^2)$ chunks of KV cache instead of $O(N)$. This gets worse as prompts get longer.

Their summary: chunked prefill **trades TTFT for TPOT**. It doesn't create anything new.

Intuition

The whole idea in one line: stop trying to schedule two incompatible jobs onto one GPU, and give each job its own GPUs — sized differently, configured differently, scaled independently.

The obvious objection is: *if prefill happens on GPU A and decode on GPU B, you have to ship the KV cache from A to B. Isn't that fatal?* The paper's central empirical claim is **no, and it's not even close**:

- KV cache for a 512-token request on a 66B model $\approx$ **1.13 GB**.
- At 10 requests/second that's 11.3 GB/s $\approx$ **90 Gbps** of network needed to make the transfer invisible.
- InfiniBand clusters do $\sim$ 800 Gbps. NVLink inside a node does $\sim$ 600 GB/s.
- Measured result on their cluster: KV transfer was **under 0.1% of total request latency**, and 95% of requests transferred in **under 30 ms** — even though their cross-node link was only 25 Gbps, because their placement algorithm keeps transfers on NVLink.

Plain version: Moving the scratch notes between two GPUs sounds expensive, but modern GPU cables are so fast that it's like handing someone a piece of paper across a desk. The cost of *copying* is nothing compared to the cost of two jobs elbowing each other for hours.

The second half of the intuition is **optimize goodput, not throughput**. Throughput says "make lots of tokens." Goodput says "finish lots of requests *on time*." The whole design follows from that switch: once deadlines are the objective, interference stops being an inefficiency and becomes an SLO violation.

Solution

1. Split into prefill instances and decode instances

Each type holds its own full copy of the model weights. A request goes: prefill instance computes the prompt and the first token $\rightarrow$ ships the KV cache to a decode instance $\rightarrow$ decode instance streams the rest. Because decode uses the GPU so poorly, you typically point **several prefill instances at one decode instance**, so the decode instance always has a fat batch to work on.

2. Different batching rules per phase

Phase	Rule	Why
Prefill	Profile the model+GPU to find L_m — the prompt length that already saturates the GPU ($\approx$ 512 tokens for 13B on A100). Batch multiple requests only if they're shorter than L_m . Anything longer runs alone.	Past L_m , adding requests adds zero efficiency and simply makes everyone in the batch wait longer. Real prompts average hundreds of tokens, so prefill batches stay small.
Decode	Batch as large as GPU memory will allow.	A single decode job wastes almost the entire GPU. Big batches are the <i>only</i> lever. Disaggregation is what finally makes big decode batches possible without wrecking TTFT.

3. Different parallelism per phase — derived from queuing theory

Plain version of the math: Think of a single checkout counter. D = how long one customer takes. R = customers arriving per second. Then $R \times D$ is how busy the counter is, from 0 (empty) to 1 (never idle). Your total wait = time being served + time standing in line. And the line term **explodes** as $R \times D$ approaches 1 — which is why a store that's 95% busy feels catastrophically worse than one that's 80% busy.

Formally, prefill after disaggregation behaves like an M/D/1 queue (random arrivals, fixed service time, one server):

```
Avg_TTFT = D + R * D^2 / (2 * (1 - R * D))
           ^^^          ^^^^^^^^^^^^^^^
           execution    queuing delay  (blows up as R * D -> 1)
```

Now plug in each parallelism style, 2 GPUs each:

```
# Intra-op / tensor parallel: makes each job FASTER (D shrinks to D/K)
# K is the speedup, 1 < K < 2 — never a full 2x, because the GPUs must talk.
Avg_TTFT_intra = D/K + R * D^2 / (2 * K * (1 - R * D))

# Inter-op / pipeline parallel: same job speed, but 2 stations means the
# bottleneck stage is only half as long, so the LINE moves twice as fast.
Avg_TTFT_inter = D + R * D^2 / (4 * (1 - R * D))
```

Read the two terms and the answer falls out:

- **Low traffic** $\rightarrow$ almost no line, so execution time dominates $\rightarrow$ **intra-op (TP) wins**, because it's the only one that makes a single job faster.
- **High traffic** $\rightarrow$ the line dominates $\rightarrow$ **inter-op (PP) wins**, because it drains the queue faster.
- **Tighter TTFT SLO** $\rightarrow$ pushes toward intra-op.

- **Worse interconnect** (smaller K, meaning TP's speedup is poor) → pushes away from intra-op. This is a real measured sensitivity, not a footnote.

For **decode** the shape is different: intra-op reduces latency but with **diminishing returns** (communication overhead eats the gains), while inter-op scales throughput **almost linearly**. So the recipe is: *use just enough intra-op to get under the TPOT SLO, then add inter-op to scale volume.*

4. Placement — letting a search algorithm pick, because humans can't

"Placement" = how many prefill instances, how many decode instances, what parallelism each uses, and which physical GPUs they sit on. Two algorithms, depending on your network:

Algorithm 1 — fast cross-node network (InfiniBand)

Transfers are free anywhere, so the two phases are independent problems. Enumerate every (inter-op, intra-op) combination that fits in memory, simulate the goodput of each, keep the one with the best **goodput per GPU** — separately for prefill and for decode. Then replicate each until you meet the target traffic rate. Complexity $O(N \cdot M^2)$ where $M \approx 8$ GPUs per node; runs in **under 1.3 minutes** even at their largest setting.

Algorithm 2 — slow cross-node network (plain ethernet)

Here you must keep KV transfers inside a machine. The clever observation: **KV cache only ever moves between matching layers** — prefill's layer-7 notes go to decode's layer 7, never to layer 30. So use inter-op parallelism to cut both instances into stages, and **put prefill-stage-*i* and decode-stage-*i* on the same physical node**. Every transfer then rides NVLink and never touches the slow network. This is why their measured transfer cost stayed under 0.1% on a 25 Gbps cluster.

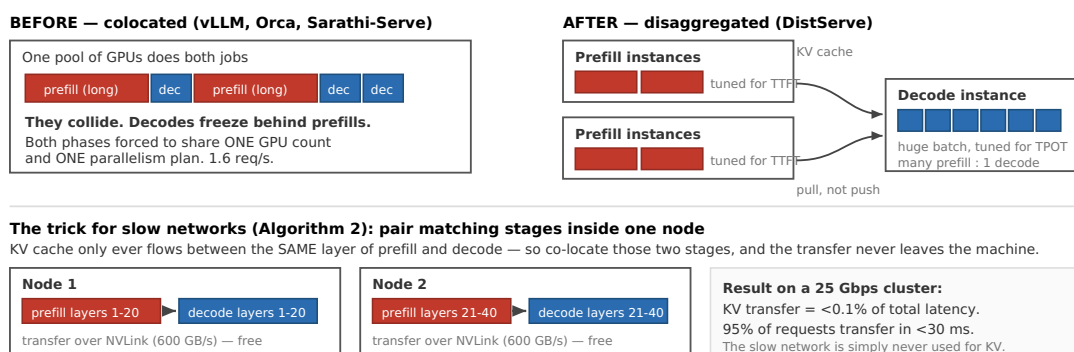
Why a simulator instead of real measurements

You can't run thousands of real configurations — it takes forever. So they built a simulator from an analytical latency model (count the FLOPs and the memory reads for each matrix multiply). Measured accuracy: **error under 2%** against the real system. This works because neural network execution time is unusually predictable.

5. Runtime scheduling — three details worth knowing

- **Dispatch:** FCFS. New request → prefill instance with the *shortest queue*; after prefill → *least loaded* decode instance.
- **Pipeline bubble control:** a "bubble" is a GPU stage sitting idle because the previous stage is still busy. Bubbles happen when consecutive batches take different amounts of time. Fix: use **token count as the proxy for batch duration**, and assemble every prefill batch to total roughly L_m tokens — group several short prompts together, run long ones alone.
- **Pull, not push.** When traffic spikes, a flood of KV caches could overwhelm the decode instance's memory. So the decode instance **pulls** KV cache when it's ready, instead of prefill **pushing** it. The prefill instance's GPU memory becomes the buffer. Each side runs at its own pace, no coordination needed. *This is a classic backpressure pattern — worth naming explicitly in an interview, since it generalizes to any producer/consumer system.*
- **Replanning:** a profiler watches average input length, output length, and arrival rate. If the workload shifts, re-run the placement search. Feasible because the search takes seconds and reloading weights takes minutes, while real workloads shift over hours.

Solution Diagram



Top: colocated vs. disaggregated. Bottom: the layer-pairing trick that makes disaggregation work without InfiniBand.

Code

Two things are worth being able to write: the placement search, and the pull-based transfer.

```
# ===== PLACEMENT SEARCH (Algorithm 1: fast network) =====
# Goal: pick GPU counts + parallelism for each phase, maximizing GOODPUT PER GPU.

def find_placement(model, workload, target_rate, gpus_per_node=8, node_limit=4):
    best_prefill, best_decode = None, None

    # Try every (tensor-parallel degree, pipeline-parallel degree) that fits in memory
    for tp in range(1, gpus_per_node + 1):
        for pp in range(1, (node_limit * gpus_per_node) // tp + 1):
            if model.size / (tp * pp) >= GPU_MEMORY: continue # won't fit
            cfg = (tp, pp)
            ngpus = tp * pp
            sharded = shard(model, cfg)

            # SIMULATE, don't measure – real profiling would take days.
            # Each returns max req/s meeting the SLO for 90% of requests.
            g_pre = simulate_prefill(sharded, workload)
            g_dec = simulate_decode(sharded, workload)

            # Compare on goodput PER GPU, not raw goodput. This is the whole objective.
            if best_prefill is None or g_pre/ngpus > best_prefill.per_gpu:
                best_prefill = Cfg(cfg, g_pre, per_gpu=g_pre/ngpus)
            if best_decode is None or g_dec/ngpus > best_decode.per_gpu:
                best_decode = Cfg(cfg, g_dec, per_gpu=g_dec/ngpus)

    # Replicate each type until total capacity covers the incoming traffic.
    n_prefill = ceil(target_rate / best_prefill.goodput)
    n_decode = ceil(target_rate / best_decode.goodput)
    return n_prefill, best_prefill, n_decode, best_decode

# Found for OPT-175B on ShareGPT: prefill TP=3,PP=3 decode TP=4,PP=3
# Different for the two phases, and not something a human would guess.
```

```
# ===== PULL-BASED KV TRANSFER (backpressure) =====
# PUSH would let a traffic spike flood the decode instance's memory and OOM it.
# PULL turns the prefill instance's GPU memory into a natural buffer.

class PrefillInstance:
    def run(self, req):
        kv, first_token = self.forward_prompt(req)
        self.ready[req.id] = kv # KEEP it in local GPU memory
        controller.announce(req.id, self.id, first_token)
        # Immediately free to start the next prompt. No waiting on the consumer.

class DecodeInstance:
    def step(self):
        # Only admit new work when there is actually room. This is the backpressure.
        while self.kv_free_blocks() > THRESHOLD and controller.has_ready():
            rid, src = controller.next_ready()
            kv = src.fetch(rid) # NVLink if same node; NCCL if not
            self.admit(rid, kv)
            self.decode_one_token_for_whole_batch()

# If decode is saturated, KV simply sits on the prefill GPUs and prefill
# eventually slows down on its own. The system degrades instead of crashing.
```

Results

Setup (90% SLO attainment)	vs vLLM	vs DeepSpeed-MII (chunked prefill)
Chatbot, OPT-13B/66B/175B, ShareGPT	2.0–4.6× rate	1.6–7.4× rate
Code completion, OPT-66B (very tight TTFT)	5.7× rate	1.6× rate
Summarization, OPT-66B (long inputs, tight TPOT)	4.3× rate, 12.6× tighter SLO	1.8× rate
Headline claim	up to 7.4× more requests, or 12.6× tighter SLO	

The ablation is the most persuasive part. They built “vLLM++” — vLLM, but with an exhaustive search over parallelism settings. Result: **identical to plain vLLM**. The default was already optimal. Meaning: *you cannot tune your way out of this by adjusting parallelism while the phases remain colocated*. The interference caps you regardless. Only separating the phases unlocks the gain.

Other numbers worth carrying: KV transfer **<0.1%** of latency; simulator error **<2%**; placement search runs in **seconds to ~1.3 minutes**.

When NOT to use disaggregation (the paper says so itself)

Scenario	What to do instead, and why
Offline / batch jobs (no one is waiting)	Use chunked prefill . With no deadlines, goodput $\approx$ throughput, and chunking's ability to fill every batch right up to the compute-bound point keeps utilization higher. Day 2 wins here.
1-2 GPUs total	Use a plain colocated system. Disaggregation needs room to allocate different resources to different phases; with one GPU there's no design space at all, only added complexity.
Very long context (100K+ tokens)	Disaggregation gets better, not worse. KV cache grows linearly with prompt length but prefill compute grows quadratically — so transfer becomes relatively cheaper. Meanwhile the gap between prefill and decode widens, so interference gets worse. Both effects favor splitting. This is a counterintuitive point and a great one to raise unprompted.

Likely Interview Questions + Answers

Q1. What's the difference between throughput and goodput, and why does it change the design?

Throughput counts tokens produced per second. Goodput counts requests finished per second **that met their latency promise**. A system can max throughput by batching aggressively while every individual user waits far too long — high throughput, near-zero goodput. Once you optimize goodput, prefill-decode interference stops being "slight inefficiency" and becomes "SLO violation," which is what justifies spending whole GPUs to separate them. The paper's opener is the proof: colocated gets 1.6 req/s, split gets 3.3 req/s per GPU, same hardware.

Q2. Doesn't shipping the KV cache between GPUs kill you?

Measured: no. 1.13 GB per 512-token request on a 66B model; ~ 90 Gbps to make it invisible at 10 req/s; InfiniBand is ~ 800 Gbps and NVLink is ~ 600 GB/s. On their own cluster — with only 25 Gbps between machines — transfer was under 0.1% of request latency and 95% of requests transferred in under 30 ms. The reason it works even on a slow network is the layer-pairing trick: KV only moves between *matching* layers, so they co-locate prefill-stage-*i* with decode-stage-*i* on one machine and transfers ride NVLink. Two caveats to volunteer: they used OPT with classic multi-head attention (worst case for KV size — GQA/MQA models would transfer far less), and it degrades on genuinely bad networking.

Q3. Chunked prefill (Sarathi-Serve) vs. disaggregation (DistServe). Which is right?

Neither — it depends on the objective, and saying so is the correct answer.

Disaggregation wins when latency SLOs are strict, prompts are long, you have decent interconnect, and you have enough GPUs to allocate separately. It *eliminates* interference rather than reducing it, and lets each phase pick its own parallelism.

Chunked prefill wins when you're throughput-oriented or offline, when you're on a handful of GPUs, or when your network is genuinely bad. It keeps every batch filled to the compute-bound point and needs no KV migration, and it uses memory better since prefill machines aren't sitting on unused KV capacity.

Honest caveats both ways: DistServe never benchmarks against Sarathi-Serve directly (it uses DeepSpeed-MII as the chunked-prefill stand-in), and Sarathi-Serve explicitly declined to benchmark against disaggregation. Neither team ran the head-to-head. Anyone claiming a clean winner is over-reading the evidence. In practice the industry has largely converged on *both* — chunked prefill inside each instance, disaggregation across instances at scale.

Q4. When should prefill use tensor parallelism vs pipeline parallelism?

Split latency into execution time plus queuing delay. Tensor parallelism (intra-op) shrinks **execution time** but with imperfect speedup $K < 2$ because of communication. Pipeline parallelism (inter-op) leaves execution time roughly alone but halves the bottleneck stage, so it drains the **queue** faster. Therefore: at low arrival rates, execution dominates $\rightarrow$ tensor parallelism. At high rates, queuing dominates $\rightarrow$ pipeline parallelism. A tighter TTFT SLO pushes toward tensor parallelism; a weak interconnect (low K) pushes away from it. For decode it's different: tensor parallelism has diminishing returns while pipeline parallelism scales throughput nearly linearly — so use just enough TP to meet the TPOT SLO, then add PP for volume.

Q5. Why simulate the configurations instead of measuring them?

The search space is every (TP, PP) pair for two phases across up to 32 GPUs. Measuring each on real hardware under a realistic load would take days per candidate. So they built an analytical latency model — count FLOPs and memory accesses for each matrix multiply, calibrate constants by profiling a few points — and simulate. It works because neural network execution time is unusually predictable (fixed shapes, no data-dependent branching). Measured accuracy: **under 2% error** versus the real system. Generalizable lesson: when the workload is predictable, replace the measurement loop with a model and you turn a days-long search into a 90-second one.

Q6. Why pull the KV cache instead of pushing it?

Backpressure. If prefill instances push, a traffic burst dumps a flood of KV caches onto the decode instance and can exhaust its memory. With pull, the decode instance fetches only when it has room, and unfetched KV simply stays in the prefill instance's GPU memory, which acts as a buffer. Prefill naturally slows down when its memory fills. The system degrades gracefully instead of crashing, and neither side needs to coordinate with the other. This is the same pattern as TCP flow control or a bounded queue between producer and consumer — worth naming, because it's reusable in any pipeline design question.

Q7. What does the vLLM++ ablation prove, and why is it the strongest result in the paper?

vLLM++ is vLLM with an exhaustive search over parallelism configurations, still colocated. It performed **identically** to default vLLM — the default was already the best available. That rules out the most obvious objection ("you just tuned parallelism better"). It shows the gain comes from **separating the phases**, not from a better configuration, and that interference imposes a ceiling you cannot tune past. Whenever you claim a structural change beats tuning, this is the shape of experiment to run.

Q8. Does disaggregation get better or worse with 200K-token contexts?

Better, which surprises people. KV cache size grows *linearly* with prompt length, so transfer cost grows linearly. But prefill compute grows *quadratically*. So transfer shrinks as a fraction of total work. Simultaneously, longer prompts mean longer prefills, which means colocated systems suffer *worse* interference. Both trends favor splitting. The one thing that does get harder is decode-side memory: KV cache per request balloons, so you need PagedAttention plus GQA plus possibly quantization underneath.

Q9. Design: a coding assistant. TTFT must be under 125 ms; TPOT can be relaxed. Walk me through it.

Tight TTFT with loose TPOT means **prefill is the binding constraint**, so bias every decision toward it. (1) **Disaggregate**, and skew the ratio toward prefill instances — the opposite of a chatbot, where decode dominates. (2) **Prefill parallelism**: tight SLO plus modest rate favors tensor parallelism to cut execution time; verify K is decent, which means keeping TP inside a node on NVLink. (3) **Batching**: profile L_m ; code prompts are often short, so batch several together up to L_m rather than running them alone. (4) **Decode** can run at a large batch with fewer GPUs since TPOT is loose. (5) **Placement**: if the network is weak, use stage pairing so KV stays on NVLink. (6) **Monitor** P90 TTFT and P90 TPOT separately with per-phase queue depth, since a prefill queue backup is invisible in TPOT. (7) **Layer on** prefix caching — coding assistants resend the same file context constantly, and a cache hit removes the prefill work entirely, which directly attacks the binding constraint. The paper's own result for exactly this workload: 5.7× the request rate of vLLM.

Q10. What's the biggest operational risk of disaggregation?

Fault propagation, and the paper admits it. In a replicated colocated system, losing one instance costs you one instance's capacity — the others are independent. In DistServe, prefill and decode instances are *coupled*: one decode instance may serve many prefill instances, so losing it can strand all of them and cripple the whole cluster. They explicitly leave fault tolerance as future work. Secondary risks worth naming: FCFS causes a **convoy effect** (one long request blocks all the short ones behind it) with no preemption implemented; and the placement is fitted to a workload distribution, so a sudden traffic-shape change leaves you mis-provisioned until the next replan.

What's not covered

Admitted gaps

- **No fault tolerance.** And disaggregation actively makes this harder — the prefill→decode dependency means one failure can cascade. Named as future work.
- **No preemption.** Pure FCFS, so a long request blocks short ones behind it (the "convoy effect"). Fixable, not fixed.
- **No head-to-head against Sarathi-Serve.** DeepSpeed-MII stands in for chunked prefill. Combined with Sarathi-Serve declining the reverse comparison, the two approaches have never been properly benchmarked against each other.

Scope limits

- **Prefill machines waste memory.** They hold KV cache only briefly; the long-lived KV lives on decode machines. So a chunk of your fleet's HBM sits underused — Sarathi-Serve's specific criticism of this approach, and it isn't quantified here.
- **Placement assumes a predictable workload.** It fits a distribution from historical traces. Fine for hourly patterns, not for genuine unpredictability.
- **Static prefill:decode ratio** between replans. No fine-grained elasticity as traffic shifts mid-window.
- **OPT models only, using classic multi-head attention.** Chosen deliberately as the worst case for KV transfer size. Modern GQA/MQA models transfer much less, so results should improve — but that's an untested claim.
- **Algorithm 1's results are simulation-only**, since their real testbed lacked the fast cross-node network it assumes.
- **No prefix caching or multi-turn KV reuse.** Interacts non-trivially with disaggregation (where would a shared prefix's KV live?) and isn't addressed.

Orthogonal and untouched

- **Heterogeneous hardware** — prefill is compute-hungry and decode is bandwidth-hungry, so putting prefill on newer compute-heavy GPUs and decode on cheaper high-memory ones is the obvious next move. Not explored, and a strong thing to propose.
- Speculative decoding, quantization, MoE, LoRA multi-tenancy.
- Cluster-level autoscaling, and cost modeling across GPU types.

Where to go next: Splitwise (concurrent, independent disaggregation paper — notably explores *heterogeneous* GPU pools, the gap above) → Orca (iteration-level batching, the shared ancestor of all three papers so far) → Mooncake (KV-cache-centric disaggregation at production scale, with prefix caching folded in — the closest thing to how this actually gets deployed) → then loop back and be able to argue Day 2 vs Day 3 from either side.