# Dynamo

INTRODUCTION TO LLM
INFERENCE SERVING SYSTEMS

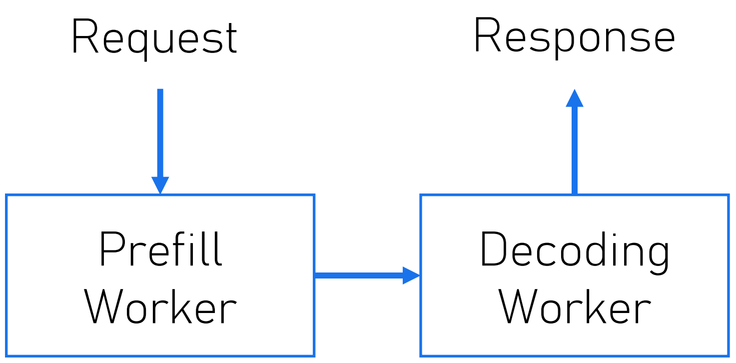
CHUHONG YUAN

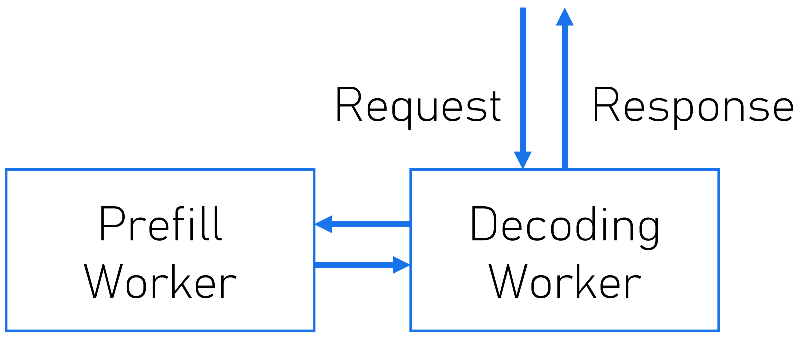
# P-D Disaggregation

- Usually, one request performs prefill and decoding on the same GPU
- What about separate these two phases to different workers?

# P-D Disaggregation

- Usually, one request performs prefill and decoding on the same GPU
- What about separate these two phases to different workers?



# P-D Disaggregation – Why?

- Discussion: what are your opinions?

# P-D Disaggregation – Why?

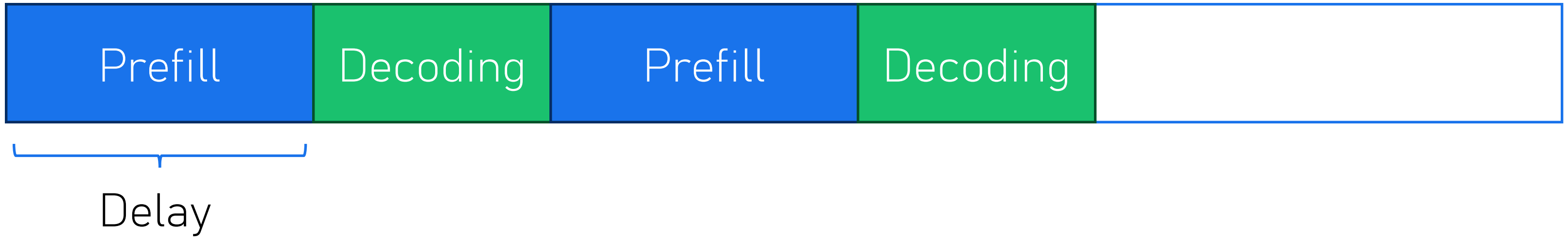
- Discussion: what are your opinions?
- Mitigating interference between the two phases

# P-D Disaggregation – Why?

- Discussion: what are your opinions?
- Mitigating interference between the two phases

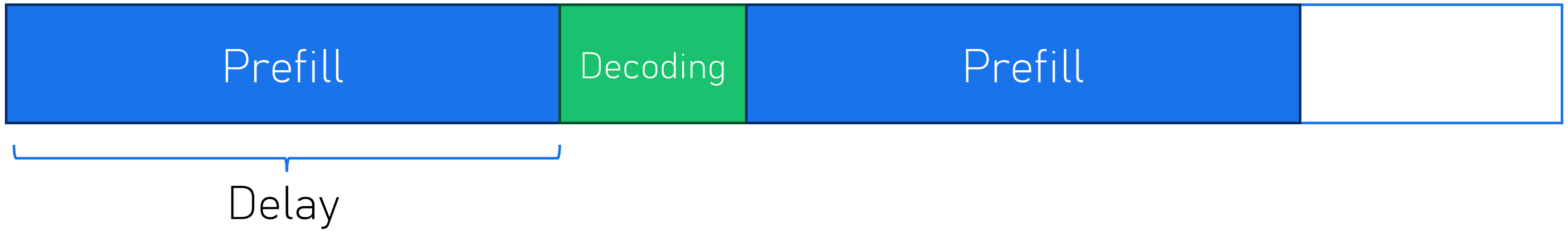
# P-D Disaggregation – Why?

- Discussion: what are your opinions?
- Mitigating interference between the two phases



# P-D Disaggregation – Why?

- Discussion: what are your opinions?
- Mitigating interference between the two phases



# P-D Disaggregation – Why?

- Discussion: what are your opinions?
- Mitigating interference between the two phases
- Opportunities for phase-tailored optimizations
  - Heterogenous serving
    - Rubin CPX
  - Batching/parallelism strategies

# Dynamo Features

**Features**

## Explore the Features of NVIDIA Dynamo

### Disaggregated Serving

Separates LLM context (prefill) and generation (decode) phases across distinct GPUs, enabling tailored model parallelism and independent GPU allocation to increase requests served per GPU.

### GPU Planner

Monitors GPU capacity in distributed inference environments and dynamically allocates GPU workers across context and generation phases  to resolve bottlenecks and optimize performance.

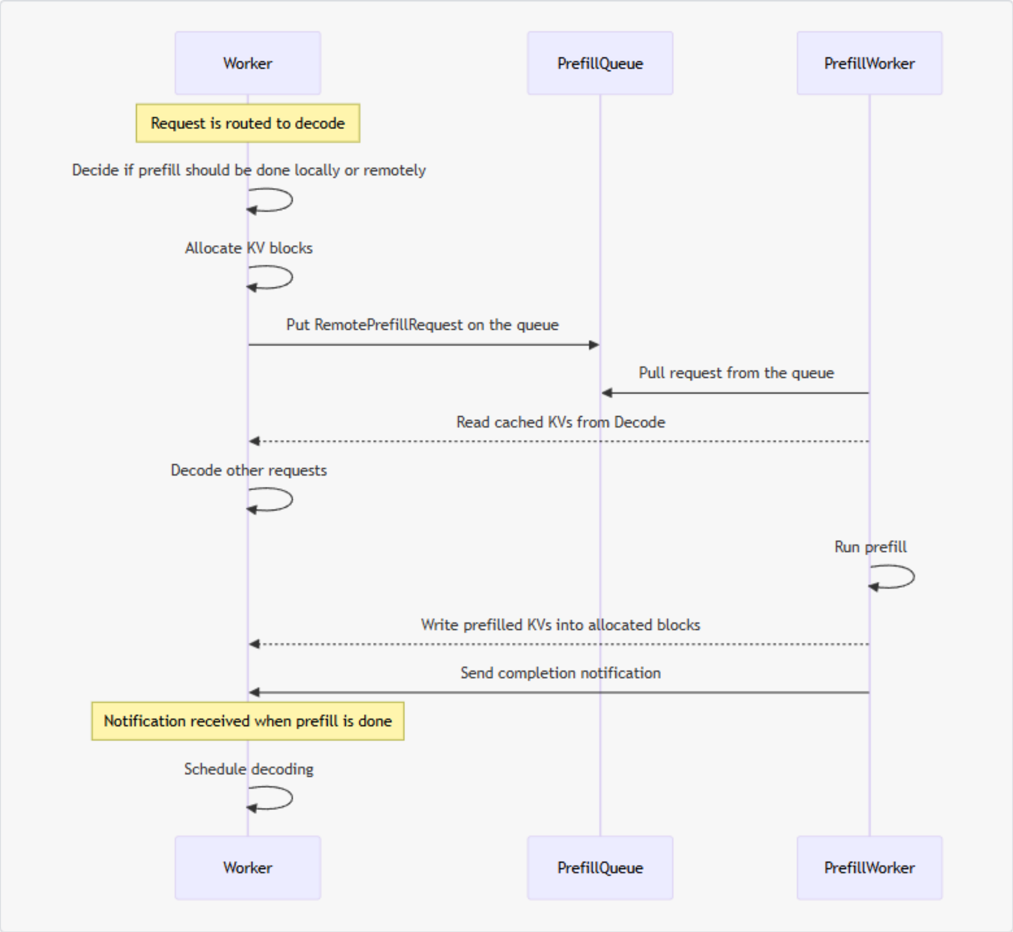
### Smart Router

Routes inference traffic efficiently, minimizing costly recomputation of repeat or overlapping requests to preserve compute resources while ensuring balanced load distribution across large GPU fleets.

### NIXL Low-Latency Communication Library

Accelerates data movement in distributed inference settings while simplifying transfer complexities across diverse hardware, including GPUs, CPUs, networks, and storage.

# Remote Prefill



# Remote Prefill

```python
async def init(runtime: DistributedRuntime, config: Config):
    """
    Instantiate and serve
    """

    component = runtime.namespace(config.namespace).component(config.component)
    await component.create_service()

    generate_endpoint = component.endpoint(config.endpoint)
    clear_endpoint = component.endpoint("clear_kv_blocks")

    prefill_router_client = (
        await runtime.namespace(config.namespace) ···
    )

    prefill_worker_client = (
        await runtime.namespace(config.namespace) ···
    )

    factory = StatLoggerFactory( ···
    engine_client, vllm_config, default_sampling_params = setup_vllm_engine( ···

    # TODO Hack to get data, move this to registering in TBD
    factory.set_num_gpu_blocks_all(vllm_config.cache_config.num_gpu_blocks)
    factory.set_request_total_slots_all(vllm_config.scheduler_config.max_num_seqs)
    factory.init_publish()

    logger.info(f"VllmWorker for {config.model} has been initialized")

    handler = DecodeWorkerHandler(
        runtime,
        component,
        engine_client,
        default_sampling_params,
        prefill_worker_client,
        prefill_router_client,
    )
```

```python
try:
    logger.debug("Starting serve_endpoint for decode worker")
    await asyncio.gather(
        # for decode, we want to transfer the in-flight requests to other decode engines,
        # because waiting them to finish can take a long time for long OSLs
        generate_endpoint.serve_endpoint(
            handler.generate,
            graceful_shutdown=config.migration_limit <= 0,
            metrics_labels=[("model", config.model)],
            health_check_payload=health_check_payload,
        ),
        clear_endpoint.serve_endpoint(
            handler.clear_kv_blocks, metrics_labels=[("model", config.model)]
        ),
    )
    logger.debug("serve_endpoint completed for decode worker")
except Exception as e:
    logger.error(f"Failed to serve endpoints: {e}")
    raise
finally:
    logger.debug("Cleaning up decode worker")
    # Cleanup background tasks
    handler.cleanup()
```

components/src/dynamo/vllm/main.py

# Remote Prefill

```python
async def generate(self, request, context):
    for key, value in request["sampling_options"].items():...

    for key, value in request["stop_conditions"].items():...

    # Use prefill router or worker if available
    can_prefill = (
        self.prefill_worker_client is not None
    ) and self.prefill_worker_client.instance_ids()

    if can_prefill:
        # Create prefill sampling params with modifications
        prefill_sampling_params = deepcopy(sampling_params)
        if prefill_sampling_params.extra_args is None:...
        prefill_sampling_params.extra_args["kv_transfer_params"] = {...
        prefill_sampling_params.max_tokens = 1
        prefill_sampling_params.min_tokens = 1

        try:
            # Send request with sampling_params and request_id in extra_args
            prefill_request = request.copy()
            # TODO (PeaBrane): this smells a bit bad as not we have two nestings
            # of extra_args (an inner one again in sampling_params)
            prefill_request["extra_args"] = {...

            # Try router first if available, fallback to worker
            if (
                self.prefill_router_client is not None
                and self.prefill_router_client.instance_ids()
            ):
                # Call router's generate endpoint which returns LLMEngineOutput
                prefill_response = await anext(...
            else:
                # Fallback to direct worker with same format
                prefill_response = await anext(...

            prefill_output = prefill_response.data()

            # Extract kv_transfer_params from response
            kv_transfer_params = prefill_output.get("extra_args", {}).get(
                "kv_transfer_params"
            )
            if kv_transfer_params:...

        except Exception as e:
            if context.is_stopped() or context.is_killed():
                logger.debug(f"Aborted Remote Prefill Request ID: {request_id}")
                return
            logger.warning(f"Prefill error: {e}, falling back to local prefill")

    async with self._abort_monitor(context, request_id):
        try:
            async for tok in self.generate_tokens(
                prompt, sampling_params, request_id
            ):
                yield tok
        except EngineDeadError as e:...
```

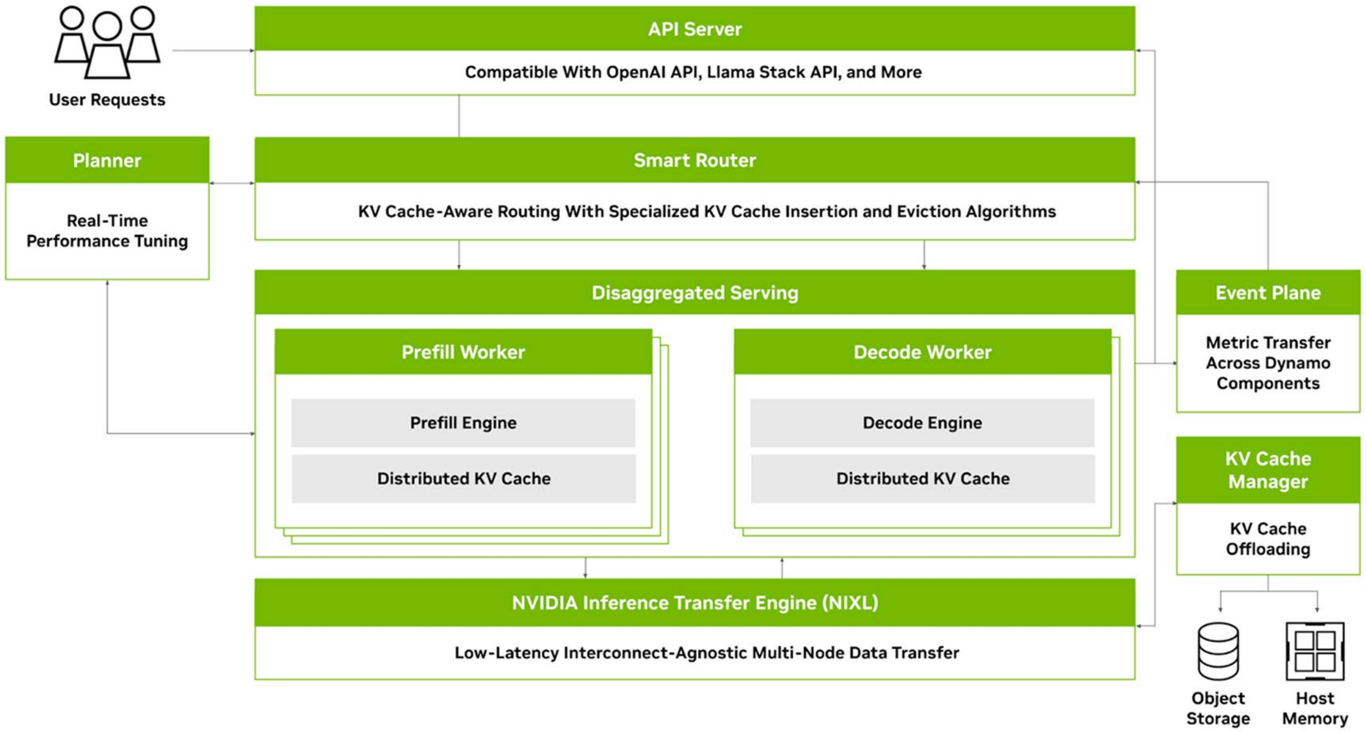
components/src/dynamo/vllm/handlers.py

# Remote Prefill

```python
# Receive KV cache in distributed KV cache transfer setting
# In disagg prefill setting, it will also recv hidden states and bypass
# model forwarding
# In KV cache database setting, it will change the model input so that
# we can skip prefilling on tokens that successfully received KV caches
# NOTE: The receive operation is blocking
bypass_model_exec = False
if self.need_recv_kv(model_input, kv_caches):
    hidden_or_intermediate_states, bypass_model_exec, model_input = \
        get_kv_transfer_group().recv_kv_caches_and_hidden_states(
            # model is used to know which layer the current worker
            # is working on, so that we can receive KV for only those
            # layers.
            model_executable,
            model_input,
            kv_caches=kv_caches
        )
```

vllm/worker/model_runner.py in vLLM

# GPU Planner


User Requests
API Server
Compatible With OpenAI API, Llama Stack API, and More
Planner
Real-Time
Performance Tuning
Smart Router
KV Cache-Aware Routing With Specialized KV Cache Insertion and Eviction Algorithms
Disaggregated Serving
Prefill Worker
Prefill Engine
Distributed KV Cache
Decode Worker
Decode Engine
Distributed KV Cache
Event Plane
Metric Transfer
Across Dynamo
Components
KV Cache
Manager
KV Cache
Offloading
NVIDIA Inference Transfer Engine (NIXL)
Low-Latency Interconnect-Agnostic Multi-Node Data Transfer
Object
Storage
Host
Memory

# GPU Planner

```python
class Planner:
    def __init__(
        self,
        runtime: Optional[DistributedRuntime],
        args: argparse.Namespace,
        dryrun: bool = False,
    ):
        self.args = args
        self.dryrun = dryrun

        # Rely on getting model name from connector
        self.model_name: Optional[str] = None

        if not self.dryrun: ...

        self.num_req_predictor = LOAD_PREDICTORS[args.load_predictor](
            window_size=args.load_prediction_window_size,
        )
        self.isl_predictor = LOAD_PREDICTORS[args.load_predictor](
            window_size=args.load_prediction_window_size,
        )
        self.osl_predictor = LOAD_PREDICTORS[args.load_predictor](
            window_size=args.load_prediction_window_size,
        )

        if "use-pre-swept-results" in args.profile_results_dir: ...
        else:
            self.prefill_interpolator = PrefillInterpolator(args.profile_results_dir)
            self.decode_interpolator = DecodeInterpolator(args.profile_results_dir)

        self.prefill_component_name = WORKER_COMPONENT_NAMES[
            self.args.backend
        ].prefill_worker_k8s_name
        self.decode_component_name = WORKER_COMPONENT_NAMES[
            self.args.backend
        ].decode_worker_k8s_name

        if not self.dryrun: ...

        self.p_correction_factor = 1.0
        self.d_correction_factor = 1.0
        if self.dryrun:
            self.no_correction = True
        else:
            self.no_correction = args.no_correction
```

components/src/dynamo/planner/utils/planner_core.py

# GPU Planner

```python
async def make_adjustments(self):
    # Skip adjustment if no traffic
    if not self.last_metrics.is_valid():...

    if not self.no_correction:
        try:
            self.p_endpoints, self.d_endpoints = await self.get_workers_info()
            logger.info(
                f"Number of prefill workers: {len(self.p_endpoints)}, number of decode workers: {len(self.d_endpoints)}"
            )

            # first correct the prediction correction factor
            # for TTFT, we expect the correction factor to be << 1 due to queuing delay
            expect_ttft = self.prefill_interpolator.interpolate_ttft(
                self.last_metrics.isl
            )
            self.p_correction_factor = self.last_metrics.ttft / expect_ttft
            # for ITL, we expect the correction factor to be close to 1
            expect_itl = self.decode_interpolator.interpolate_itl(
                concurrency=self.last_metrics.num_req  # type: ignore
                / len(self.d_endpoints)
                * self.last_metrics.request_duration  # type: ignore
                / self.args.adjustment_interval,
                context_length=self.last_metrics.isl + self.last_metrics.osl / 2,  # type: ignore
            )
            self.d_correction_factor = self.last_metrics.itl / expect_itl
            logger.info(
                f"Correction factors: TTFT: {self.p_correction_factor:.3f}, ITL: {self.d_correction_factor:.3f}"
            )
        except Exception as e:
            logger.error(f"Failed to correct prediction factors: {e}")
            return

    next_num_req, next_isl, next_osl = self.predict_load()

    if next_num_req is not None and next_isl is not None and next_osl is not None:
        try:
            next_num_p, next_num_d = self._compute_replica_requirements(
                next_num_req, next_isl, next_osl
            )
        except Exception as e:
            logger.error(f"Failed to compute number of replicas: {e}")
            return

    if not self.args.no_operation:
        target_replicas = [
            TargetReplica(
                sub_component_type=SubComponentType.PREFILL,
                component_name=self.prefill_component_name,
                desired_replicas=next_num_p,
            ),
            TargetReplica(
                sub_component_type=SubComponentType.DECODE,
                component_name=self.decode_component_name,
                desired_replicas=next_num_d,
            ),
        ]
        await self.connector.set_component_replicas(target_replicas, blocking=False)
```

components/src/dynamo/planner/utils/planner_core.py

# GPU Planner

```python
# Auto ARIMA model from pmdarima
Hongkuan Zhou, 2 months ago | 1 author (Hongkuan Zhou)
class ARIMAPredictor(BasePredictor):
    def __init__(self, window_size=100, minimum_data_points=5):
        super().__init__(minimum_data_points=minimum_data_points)
        self.window_size = window_size  # How many past points to use
        self.model = None

    def add_data_point(self, value):
        super().add_data_point(value)
        # Keep only the last window_size points
        if len(self.data_buffer) > self.window_size:
            self.data_buffer = self.data_buffer[-self.window_size :]

    def predict_next(self):
        """Predict the next value(s)"""
        if len(self.data_buffer) < self.minimum_data_points:
            return self.get_last_value()

        # Check if all values are the same (constant data)
        # pmdarima will predict 0 for constant data, we need to correct its prediction
        if len(set(self.data_buffer)) == 1:
            return self.data_buffer[0]  # Return the constant value

        try:
            # Fit auto ARIMA model
            self.model = pmdarima.auto_arima(
                self.data_buffer,
                suppress_warnings=True,
                error_action="ignore",
            )

            # Make prediction
            forecast = self.model.predict(n_periods=1)
            return forecast[0]
        except Exception as e:
            # Log the specific error for debugging
            logger.warning(f"ARIMA prediction failed: {e}, using last value")
            return self.get_last_value()
```

```python
def _compute_replica_requirements(
    self, next_num_req: float, next_isl: float, next_osl: float
) -> tuple[int, int]:
    """Compute the number of prefill and decode replicas needed based on predicted load.

    Args:
        next_num_req: Predicted number of requests
        next_isl: Predicted input sequence length
        next_osl: Predicted output sequence length

    Returns:
        tuple[int, int]: Number of prefill and decode replicas needed
    """
    # compute how many replicas are needed for prefill
    # here we assume the prefill bias is purely due to request queueing
    # and we increase the number of prefill replicas linearly to account for the queueing delay
    pred_prefill_throughput = (
        next_num_req
        * next_isl
        / self.args.adjustment_interval
        * min(1, self.p_correction_factor)
    )
    next_num_p = math.ceil(
        pred_prefill_throughput
        / self.prefill_interpolator.interpolate_thpt_per_gpu(next_isl)
        / self.args.prefill_engine_num_gpu
    )

    logger.info(...

    # compute how many replicas are needed for decode
    # 1. apply d_correction_factor to the ITL SLA
    # Prevent divide by zero when d_correction_factor is 0 (no metrics yet)
    if self.d_correction_factor <= 0:...
    else:
        corrected_itl = self.args.itl / self.d_correction_factor
    # 2. reversely find out what is best throughput/gpu that can achieve corrected_itl under the predicted context length
    (...
        itl=corrected_itl, context_length=next_isl + next_osl / 2
    )
    # 3. compute number of decode replicas needed
    pred_decode_throughput = next_num_req * next_osl / self.args.adjustment_interval
    next_num_d = math.ceil(
        pred_decode_throughput
        / pred_decode_thpt_per_gpu
        / self.args.decode_engine_num_gpu
    )

    logger.info(
        f"Decode calculation: {pred_decode_throughput:.2f}(d_thpt) / "
        f"{pred_decode_thpt_per_gpu * self.args.decode_engine_num_gpu:.2f}(d_engine_cap) = "
        f"{next_num_d}(num_d)"
    )

    # correct num_p and num_d based on the gpu budget
    next_num_p = max(next_num_p, self.args.min_endpoint)
    next_num_d = max(next_num_d, self.args.min_endpoint)
    logger.info(...
```

components/src/dynamo/planner/utils/planner_core.py
components/src/dynamo/planner/utils/load_predictor.py

# KVCache-Aware Router

lib/llm/src/kv_router.rs

```rust
impl KvRouter {
    pub async fn new(
        component: Component,
        block_size: u32,
        selector: Option<Box<dyn WorkerSelector + Send + Sync>>,
        kv_router_config: Option<KvRouterConfig>,
        consumer_uuid: String,
    ) -> Result<Self> {
        let kv_router_config: KvRouterConfig = kv_router_config.unwrap_or_default();

        let cancellation_token: CancellationToken = component Component
            .drt() &DistributedRuntime
            .primary_lease() Option<Lease>
            .expect(msg: "Cannot KV route static workers") Lease
            .primary_token();

        let generate_endpoint: Endpoint = component.endpoint("generate");
        let client: Client = generate_endpoint.client().await?;

        let instances_rx: Receiver<Vec<Instance>> = match client.instance_source.as_ref() {…
        };

        // Create runtime config watcher using the generic etcd watcher…
        let etcd_client: Client = component Component
            .drt() &DistributedRuntime
            .etcd_client() Option<Client>
            .expect(msg: "Cannot KV route without etcd client");

        let runtime_configs_watcher: TypedPrefixWatcher<i64, ModelRuntimeConfi… = watch_prefix_with_extraction(…
        .await?;
        let runtime_configs_rx: Receiver<HashMap<i64, ModelRuntimeConfig>… = runtime_configs_watcher.receiver();

        let indexer: Indexer = if kv_router_config.overlap_score_weight == 0.0 {
            // When overlap_score_weight is zero, we don't need to track prefixes
            Indexer::None
        } else if kv_router_config.use_kv_events {
            let kv_indexer_metrics: Arc<KvIndexerMetrics> = indexer::KvIndexerMetrics::from_component(&component);
            Indexer::KvIndexer(KvIndexer::new(
                cancellation_token.clone(),
                block_size,
                kv_indexer_metrics,
            ))
        } else {
            // hard code 120 seconds for now
            Indexer::ApproxKvIndexer(ApproxKvIndexer::new(
                cancellation_token.clone(),
                block_size,
                ttl: Duration::from_secs(120),
            ))
        };

        let scheduler: KvScheduler = KvScheduler::start(
            component.clone(),
            block_size,
            instances_rx,
            runtime_configs_rx,
            selector,
            kv_router_config.router_replica_sync,
            router_uuid: consumer_uuid.clone(),
        ) impl Future<Output = Result<…, …>>
        .await?;
```

# KVCache-Aware Router

lib/llm/src/kv_router/indexer.rs

```rust
impl KvIndexer {
    /// Create a new `KvIndexer`.
    ///
    /// ### Arguments
    ///
    /// * `token` - A `CancellationToken` for managing shutdown.
    /// * `expiration_duration` - The amount of time that block usage should be buffered.
    ///
    /// ### Returns
    ///
    /// A new `KvIndexer`.
    pub fn new_with_frequency(
        token: CancellationToken,
        expiration_duration: Option<Duration>,
        kv_block_size: u32,
        metrics: Arc<KvIndexerMetrics>,
    ) -> Self {
        let (event_tx: Sender<RouterEvent>, event_rx: Receiver<RouterEvent…>) = mpsc::channel::<RouterEvent>(2048);
        let (match_tx: Sender<MatchRequest>, match_rx: Receiver<MatchReque…>) = mpsc::channel::<MatchRequest>(128);
        let (remove_worker_tx: Sender<i64>, remove_worker_rx: Receiver<i64>) = mpsc::channel::<WorkerId>(buffer: 16);
        let (dump_tx: Sender<DumpRequest>, dump_rx: Receiver<DumpRequest…>) = mpsc::channel::<DumpRequest>(16);
        let cancel_clone: CancellationToken = token.clone();

        let task: JoinHandle<()> = std::thread::spawn(move || {
            // Create a single-threaded tokio runtime
            let runtime: Runtime = tokio::runtime::Builder::new_current_thread() Builder
                .enable_all() &mut Builder
                .build() Result<Runtime, Error>
                .unwrap();

            runtime.block_on(future: async move {
                let cancel: CancellationToken = cancel_clone;
                let mut match_rx: Receiver<MatchRequest> = match_rx;
                let mut event_rx: Receiver<RouterEvent> = event_rx;
                let mut remove_worker_rx: Receiver<i64> = remove_worker_rx;
                let mut dump_rx: Receiver<DumpRequest> = dump_rx;
                let mut trie: RadixTree = RadixTree::new_with_frequency(expiration_duration);
                loop {
                    tokio::select! {
                        biased;

                        _ = cancel.cancelled() => {
                            tracing::debug!("KvCacheIndexer progress loop shutting down");
                            return;
                        }

                        Some(worker) = remove_worker_rx.recv() => {
                            trie.remove_worker(worker);
                        }

                        Some(event) = event_rx.recv() => {
                            let event_type = KvIndexerMetrics::get_event_type(&event.event.data);
                            let result = trie.apply_event(event);
                            metrics.increment_event_applied(event_type, result);
                        }

                        Some(dump_req) = dump_rx.recv() => {
                            let events = trie.dump_tree_as_events();
                            let _ = dump_req.resp.send(events);
                        }

                        Some(req) = match_rx.recv() => {
                            let matches = trie.find_matches(req.sequence, req.early_exit);      Yan Ru Pei, 4 days ago
                            let _ = req.resp.send(matches);
                        }
```

# KVCache-Aware Router

```rust
#[async_trait]
impl KvIndexerInterface for KvIndexer {
    async fn find_matches(
        &self,
        sequence: Vec<LocalBlockHash>,
    ) -> Result<OverlapScores, KvRouterError> {
        let (resp_tx: Sender<OverlapScores>, resp_rx: Receiver<OverlapSc…) = oneshot::channel();
        let req: MatchRequest = MatchRequest {
            sequence,
            early_exit: false,
            resp: resp_tx,
        };

        if let Err(e: SendError<MatchRequest>) = self.match_tx.send(req).await {
            tracing::error!(
                "Failed to send match request: {:?}; the indexer maybe offline",
                e
            );
            return Err(KvRouterError::IndexerOffline);
        }

        resp_rx Receiver<OverlapScores>
            .await Result<OverlapScores, RecvError>
            .map_err(|_| KvRouterError::IndexerDroppedRequest)
    }
```

```rust
GuanLuo, 8 months ago | 1 author (GuanLuo)
/// Scores representing the overlap of workers.
#[derive(Debug, Clone, Serialize, Deserialize)]
6 implementations
pub struct OverlapScores {
    // map of worker_id to score
    pub scores: HashMap<WorkerId, u32>,
    // List of frequencies that the blocks have been accessed. Entries with value 0 are omitted.
    pub frequencies: Vec<usize>,
}
```

lib/llm/src/kv_router/indexer.rs

# KVCache-Aware Router

- `--kv-events`: Sets whether to listen to KV events for maintaining the global view of cached blocks. If true, then we use the `KvIndexer` to listen to the block creation and deletion events. If false, `ApproxKvIndexer`, which assumes the kv cache of historical prompts exists for fixed time durations (hard-coded to 120s), is used to predict the kv cache hit ratio in each engine. Set false if your backend engine does not emit KV events.

lib/llm/src/kv_router/approx.rs

```rust
impl ApproxKvIndexer {
    pub fn new(token: CancellationToken, kv_block_size: u32, ttl: Duration) -> Self {
        let (match_tx: Sender<MatchRequest>, mut match_rx: Receiver<MatchReque…) = mpsc::channel::<MatchRequest>(2048);
        let (route_tx: Sender<RouterResult>, mut route_rx: Receiver<RouterResu…) = mpsc::channel::<RouterResult>(2048);
        let (remove_worker_tx: Sender<i64>, mut remove_worker_rx: Receiver<i64>) = mpsc::channel::<WorkerId>(buffer: 16);
        let (dump_tx: Sender<DumpRequest>, mut dump_rx: Receiver<DumpRequest…) = mpsc::channel::<DumpRequest>(16);
        let cancel_clone: CancellationToken = token.clone();
        let task: JoinHandle<()> = std::thread::spawn(move || {
            // create a new tokio runtime which will only perform work on a single thread
            let runtime: Runtime = tokio::runtime::Builder::new_current_thread() Builder
                .enable_all() &mut Builder
                .build() Result<Runtime, Error>
                .unwrap();

            runtime.block_on(future: async move {
                let mut trie: RadixTree = RadixTree::new();
                // Use a reasonable threshold - can be made configurable if needed
                let mut timer_manager: TimerManager<TimerEntry> = TimerManager::new(ttl, threshold: 50);
                let mut event_id: u64 = 0;
                loop {
                    // Create a future that sleeps until the next expiration time.
                    let expiry_fut: Sleep = if let Some(next_expiry: Instant) = timer_manager.peek_next_expiry() {…
                    } else {…
                    };

                    tokio::select! {
                        _ = cancel_clone.cancelled() => {
                            tracing::debug!("Approximate Indexer progress loop shutting down");
                            return;
                        }

                        Some(worker) = remove_worker_rx.recv() => {…

                        Some(result) = route_rx.recv() => {
                            let hashes = result.local_hashes.iter().zip(result.sequence_hashes.iter());

                            let stored_event = KvCacheEventData::Stored(KvCacheStoreData {
                                parent_hash: None,
                                blocks: hashes.map(|(local_hash, sequence_hash)| KvCacheStoredBlockData {
                                    tokens_hash: *local_hash,
                                    block_hash: ExternalSequenceBlockHash(*sequence_hash),
                                }).collect(),
                            });
                            event_id += 1;

                            let event = RouterEvent::new(
                                result.worker_id,
                                KvCacheEvent {
                                    event_id,
                                    data: stored_event,
                                }
                            );

                            let _ = trie.apply_event(event);

                            timer_manager.insert(result.sequence_hashes.iter().map(|h| TimerEntry {
                                key: ExternalSequenceBlockHash(*h),
                                worker: result.worker_id,
                            }).collect());
                        }
```

# KVCache-Aware Router

```rust
                _ = expiry_fut => {
                    let expired = timer_manager.pop_expired();

                    expired.iter().for_each(|e| {
                        event_id += 1;

                        let event = RouterEvent::new(
                            e.worker,
                            KvCacheEvent {
                                event_id,
                                data: KvCacheEventData::Removed(KvCacheRemoveData {
                                    block_hashes: vec![e.key],
                                }),
                            }
                        );

                        let _ = trie.apply_event(event);
                    });
                }
            } tokio::select!
        }
    });
});
```

lib/llm/src/kv_router/approx.rs

```rust
impl ApproxKvIndexer {
    pub fn new(token: CancellationToken, kv_block_size: u32, ttl: Duration) -> Self {
        let (match_tx: Sender<MatchRequest>, mut match_rx: Receiver<MatchReque…) = mpsc::channel::<MatchRequest>(2048);
        let (route_tx: Sender<RouterResult>, mut route_rx: Receiver<RouterResu…) = mpsc::channel::<RouterResult>(2048);
        let (remove_worker_tx: Sender<i64>, mut remove_worker_rx: Receiver<i64>) = mpsc::channel::<WorkerId>(buffer: 16);
        let (dump_tx: Sender<DumpRequest>, mut dump_rx: Receiver<DumpRequest…) = mpsc::channel::<DumpRequest>(16);
        let cancel_clone: CancellationToken = token.clone();
        let task: JoinHandle<()> = std::thread::spawn(move || {
            // create a new tokio runtime which will only perform work on a single thread
            let runtime: Runtime = tokio::runtime::Builder::new_current_thread() Builder
                .enable_all() &mut Builder
                .build() Result<Runtime, Error>
                .unwrap();

            runtime.block_on(future: async move {
                let mut trie: RadixTree = RadixTree::new();
                // Use a reasonable threshold - can be made configurable if needed
                let mut timer_manager: TimerManager<TimerEntry> = TimerManager::new(ttl, threshold: 50);
                let mut event_id: u64 = 0;
                loop {
                    // Create a future that sleeps until the next expiration time.
                    let expiry_fut: Sleep = if let Some(next_expiry: Instant) = timer_manager.peek_next_expiry() {…
                    } else {…
                    };

                    tokio::select! {
                        _ = cancel_clone.cancelled() => {
                            tracing::debug!("Approximate Indexer progress loop shutting down");
                            return;
                        }

                        Some(worker) = remove_worker_rx.recv() => {…

                        Some(result) = route_rx.recv() => {
                            let hashes = result.local_hashes.iter().zip(result.sequence_hashes.iter());

                            let stored_event = KvCacheEventData::Stored(KvCacheStoreData {
                                parent_hash: None,
                                blocks: hashes.map(|(local_hash, sequence_hash)| KvCacheStoredBlockData {
                                    tokens_hash: *local_hash,
                                    block_hash: ExternalSequenceBlockHash(*sequence_hash),
                                }).collect(),
                            });
                            event_id += 1;

                            let event = RouterEvent::new(
                                result.worker_id,
                                KvCacheEvent {
                                    event_id,
                                    data: stored_event,
                                }
                            );

                            let _ = trie.apply_event(event);

                            timer_manager.insert(result.sequence_hashes.iter().map(|h| TimerEntry {
                                key: ExternalSequenceBlockHash(*h),
                                worker: result.worker_id,
                            }).collect());
                        }
```

# KVCache-Aware Router

```rust
pub struct KvScheduler {
    request_tx: tokio::sync::mpsc::Sender<SchedulingRequest>,
    slots: Arc<ActiveSequencesMultiWorker>,
}

impl KvScheduler {
    pub async fn start(
        component: Component,
        block_size: u32,
        instances_rx: watch::Receiver<Vec<Instance>>,
        runtime_configs_rx: watch::Receiver<HashMap<i64, ModelRuntimeConfig>>,
        selector: Option<Box<dyn WorkerSelector + Send + Sync>>,
        replica_sync: bool,
        router_uuid: String,
    ) -> Result<Self, KvSchedulerError> {
        let selector: Box<dyn WorkerSelector + Send + Sync> = selector.unwrap_or(defa… Box::new(DefaultWorkerSelector::default()));
        let instances: Vec<Instance> = instances_rx.borrow().clone();
        let runtime_configs: HashMap<i64, ModelRuntimeConfig> = runtime_configs_rx.borrow().clone();

        // Create shared workers_with_configs wrapped in Arc<RwLock>
        let workers_with_configs: Arc<RwLock<HashMap<i64, Option<ModelRuntimeConfig>>>> = {…
        };

        let worker_ids: Vec<i64> = instances Vec<Instance>
            .iter() Iter<'_, Instance>
            .map(|instance: &Instance| instance.instance_id) impl Iterator<Item = i64>
            .collect();
        let slots: Arc<ActiveSequencesMultiWorker> = Arc::new(data: ActiveSequencesMultiWorker::new(…
        ));

        // Spawn background task to monitor and update workers_with_configs
        let workers_monitor: Arc<RwLock<HashMap<i64, Option<…>>>> = workers_with_configs.clone();
        let slots_monitor: Arc<ActiveSequencesMultiWorker> = slots.clone();
        let mut instances_monitor_rx: Receiver<Vec<Instance>> = instances_rx.clone();
        let mut configs_monitor_rx: Receiver<HashMap<i64, ModelRuntimeConfig>… = runtime_configs_rx.clone();
        let monitor_cancel_token: CancellationToken = component.drt().primary_token();
        tokio::spawn(future: async move {…
        });

        let slots_clone: Arc<ActiveSequencesMultiWorker> = slots.clone();
        let workers_scheduler: Arc<RwLock<HashMap<i64, Option<…>>>> = workers_with_configs.clone();
        let (request_tx: Sender<SchedulingRequest>, request_rx: Receiver<Sched…) = tokio::sync::mpsc::channel::<SchedulingRequest>(1024);
        let scheduler_cancel_token: CancellationToken = component.drt().primary_token();
        let ns_clone: Namespace = component.namespace().clone();
```

lib/llm/src/kv_router/scheduler.rs

# KVCache-Aware Router

```rust
    // Use softmax sampling to select worker
    // Use override if provided, otherwise use default config
    let temperature: f64 = request &SchedulingRequest
        .router_config_override Option<RouterConfigOverride>
        .as_ref() Option<&RouterConfigOverride>
        .and_then(|cfg: &RouterConfigOverride| cfg.router_temperature) Option<f64>
        .unwrap_or(default: self.kv_router_config.router_temperature);
    let best_worker_id: i64 = softmax_sample(&worker_logits, temperature);
    let best_logit: f64 = worker_logits[&best_worker_id];

    let best_overlap: u32 = *overlaps.get(&best_worker_id).unwrap_or(default: &0);
    let total_blocks_info: String = workers &HashMap<i64, Option<ModelRuntimeCo…
        .get(&best_worker_id) Option<&Option<ModelRuntimeConfig>>
        .and_then(|cfg: &Option<ModelRuntimeConfig>| cfg.as_ref()) Option<&ModelR…
        .and_then(|cfg: &ModelRuntimeConfig| cfg.total_kv_blocks) Option<u64>
        .map(|blocks: u64| format!(", total blocks: {}", blocks)) Option<String>
        .unwrap_or_default();

    tracing::info!(
        "Selected worker: {}, logit: {:.3}, cached blocks: {}{}",
        best_worker_id,
        best_logit,
        best_overlap,
        total_blocks_info
    );

    Ok(WorkerSelectionResult {
        worker_id: best_worker_id,
        required_blocks: request_blocks as u64,
        overlap_blocks: overlaps.get(&best_worker_id).copied().unwrap_or(default: 0),
    })
  } fn select_worker
} impl WorkerSelector for DefaultWorkerSelect…
```

lib/llm/src/kv_router/scheduler.rs

```rust
impl WorkerSelector for DefaultWorkerSelector {
    fn select_worker(
        &self,
        workers: &HashMap<i64, Option<ModelRuntimeConfig>>,
        request: &SchedulingRequest,
        block_size: u32,
    ) -> Result<WorkerSelectionResult, KvSchedulerError> {
        assert!(request.isl_tokens > 0);

        if workers.is_empty() {…

        let isl: usize = request.isl_tokens;
        let request_blocks: usize = isl.div_ceil(block_size as usize);
        let overlaps: &HashMap<i64, u32> = &request.overlaps.scores;

        let decode_blocks: &HashMap<i64, usize> = &request.decode_blocks;
        let prefill_tokens: &HashMap<i64, usize> = &request.prefill_tokens;

        let mut worker_logits: HashMap<i64, f64> = HashMap::new();
        let mut max_logit: f64 = f64::NEG_INFINITY;

        // Calculate logits for each worker
        for worker_id: &i64 in workers.keys() {
            let overlap: u32 = *overlaps.get(worker_id).unwrap_or(default: &0);

            // this is the number of prefill tokens the worker would have if the request were scheduled there
            let prefill_token: usize = *prefill_tokens.get(worker_id).unwrap_or(default: &isl);
            let potential_prefill_block: f64 = (prefill_token as f64) / (block_size as f64);

            // this is the number of decode blocks the worker would have if the request were scheduled there
            let decode_block: f64 = *decode_blocks &HashMap<i64, usize>
                .get(worker_id) Option<&usize>
                .unwrap_or(default: &(potential_prefill_block.floor() as usize))
                as f64;

            // Use override if provided, otherwise use default config
            let overlap_weight: f64 = request &SchedulingRequest
                .router_config_override Option<RouterConfigOverride>
                .as_ref() Option<&RouterConfigOverride>
                .and_then(|cfg: &RouterConfigOverride| cfg.overlap_score_weight) Option<f64>
                .unwrap_or(default: self.kv_router_config.overlap_score_weight);

            // Calculate logit (lower is better)
            let logit: f64 = overlap_weight * potential_prefill_block + decode_block;
            max_logit = max_logit.max(logit);

            worker_logits.insert(k: *worker_id, v: logit);

            tracing::info!(
                "Formula for {worker_id} with {overlap} cached blocks: {logit:.3} \
                 = {overlap_weight:.1} * prefill_blocks + decode_blocks \
                 = {overlap_weight:.1} * {potential_prefill_block:.3} + {decode_block:.3}"      Yan Ru Pei, 2 mont
            );
        }
```

# NIXL Transfer

lib > bindings > python > src > dynamo > nixl_connect > __init__.py > Device > __init__

```python
            logger.warning(
                "dynamo.nixl_connect: Failed to load CuPy for GPU acceleration, utilizing numpy
            )
        except ImportError as e:
            raise ImportError("Numpy or CuPy must be installed to use this module.") from e


J Wyman, 3 months ago | 1 author (J Wyman)
class AbstractOperation(ABC): ···


J Wyman, 3 months ago | 1 author (J Wyman)
class ActiveOperation(AbstractOperation): ···


J Wyman, 3 months ago | 1 author (J Wyman)
class Connector: ···


J Wyman, 3 months ago | 1 author (J Wyman)
class Descriptor: ···
```

```python
Didier Durand, last month | 12 authors (Nicolò Lucchesi and others)
class NixlConnectorWorker:
    """Implementation of Worker side methods"""

    def __init__(self, vllm_config: VllmConfig, engine_id: str): ···

    def __del__(self):
        """Cleanup background threads on destruction."""
        self._handshake_initiation_executor.shutdown(wait=False)
        if self._nixl_handshake_listener_t:
            self._nixl_handshake_listener_t.join(timeout=0)

    @staticmethod
    def _nixl_handshake_listener(metadata: NixlAgentMetadata, ···

    def _nixl_handshake( ···

    def initialize_host_xfer_buffer( ···

    def set_host_xfer_buffer_ops(self, copy_operation: CopyBlocksOp): ···

    def _background_nixl_handshake(self, req_id: str, ···

    def register_kv_caches(self, kv_caches: dict[str, torch.Tensor]): ···

    def add_remote_agent(self, ···

    def sync_recved_kv_to_device(self, req_id: str, meta: ReqMeta): ···

    def save_kv_to_host(self, metadata: NixlConnectorMetadata): ···

    def get_finished(self) -> tuple[set[str], set[str]]: ···

    def _get_new_notifs(self) -> set[str]: ···

    def _pop_done_transfers( ···

    def start_load_kv(self, metadata: NixlConnectorMetadata): ···

    def _read_blocks_for_req(self, req_id: str, meta: ReqMeta): ···

    def _read_blocks(self, local_block_ids: list[int], ···

    def _get_block_descs_ids(self, ···

    def get_backend_aware_kv_block_len(self): ···
```

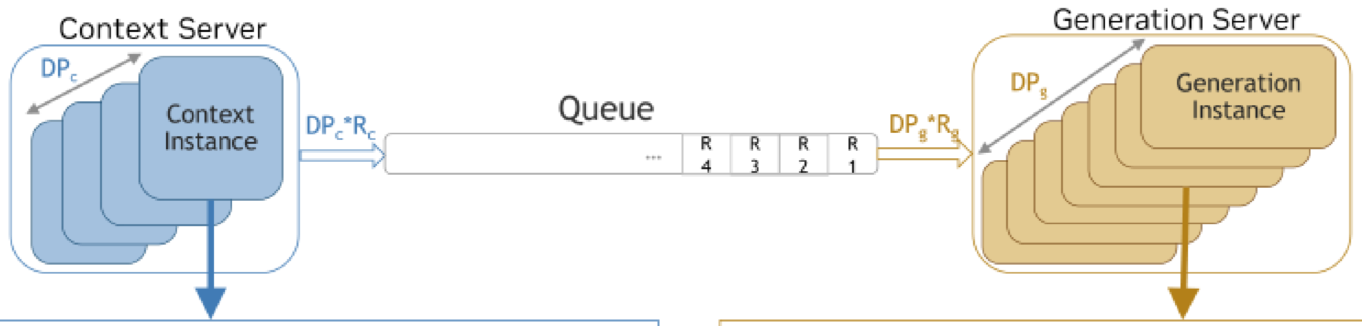
vllm/distributed/kv_transfer/kv_connector/v1/nixl_connector.py in vLLM

# Beyond The Buzz Study

- Does disaggregated serving outperform co-located serving?
- Under which scenarios should P-D disaggregation be applied?
- Terminologies: ISL (input length), OSL (output length), FTL (TTFT), TTL(TPOT)
- GPU: Blackwell
- Models: DeepSeek-R1, Llama3.1-8B, 70B, 405B, FP4 quantization
- Workloads: different combinations of ISL and OSL
- Baseline: co-located serving w/, w/o piggybacking

# Pareto Frontier

Choose $DP_c$ and $DP_g$ such that $DP_c{*}R_c \approx DP_g{*}R_g$ (while maintaining $DP_c{*}R_c \leq DP_g{*}R_g$ )




Rc: Request processing rate per Context Instance
- Throughput for best {TP, EP, PP, ...} mapping within FTL SLA
- Depends on ISL, model dims and hardware characteristics
- Usually prefers PP (with context chunking for long ISL)

Rg: Request processing rate per Generation Instance
- Throughput for best {TP, EP, PP, ...} mapping within TTL SLA
- Depends on ISL, OSL, model dims and hardware characteristics
- Usually prefers high TP at lower latencies

# Evaluation – Model

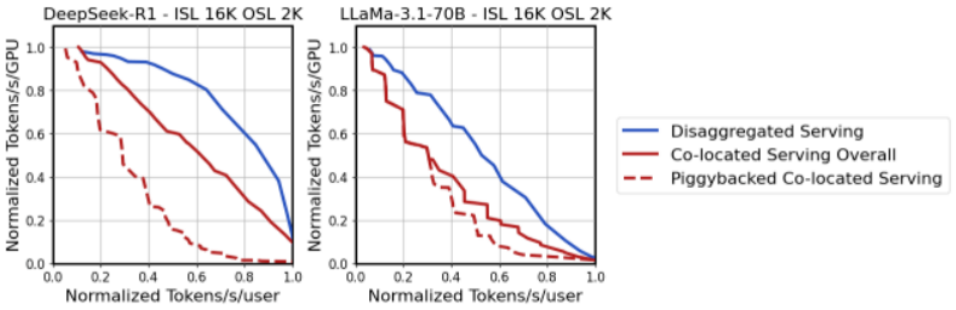



Figure 6: Disaggregated vs. co-located serving. Co-located serving overall (red-solid) is the superposition of piggybacked (red-dottec
and non-piggybacked configurations.

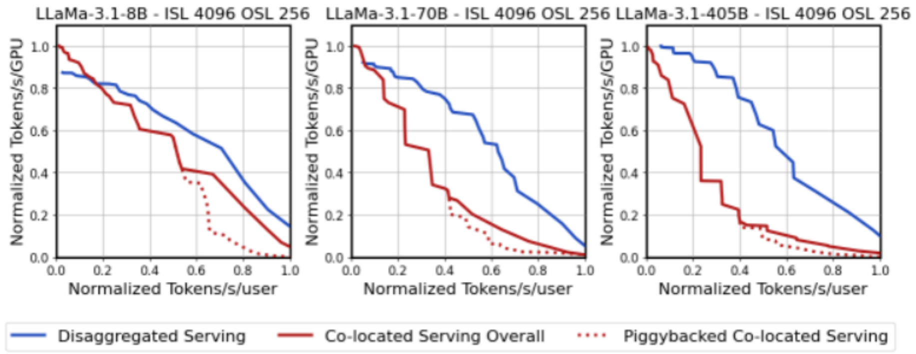
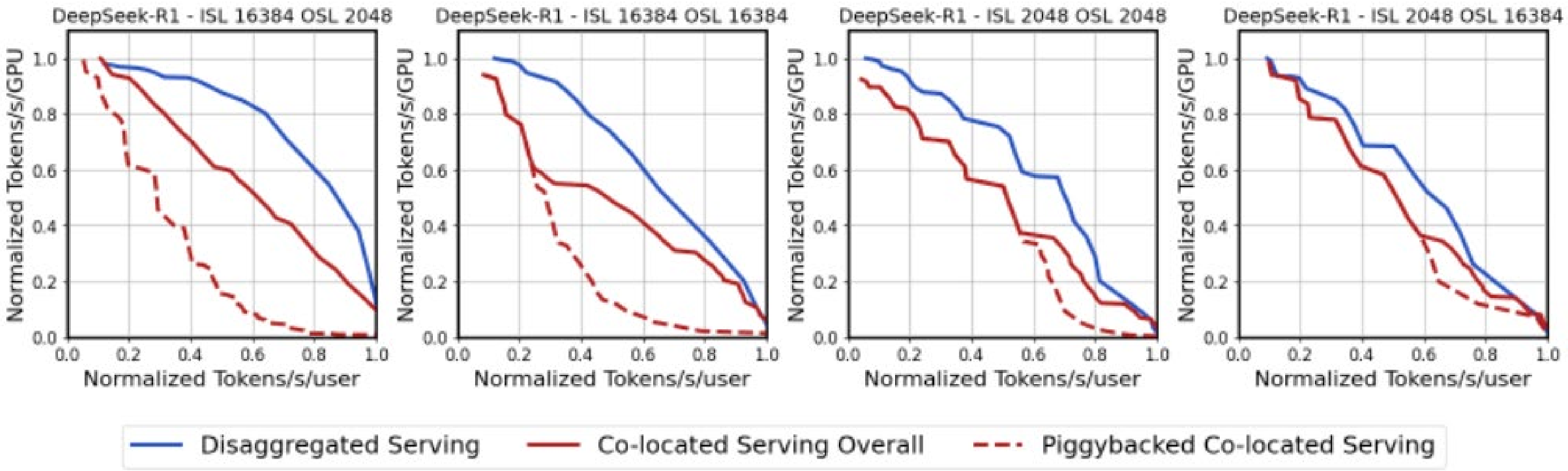



Figure 7: Larger models benefit more from disaggregated serving due to a richer search space.

# Evaluation - Traffic



# Evaluation – P/D Ratio

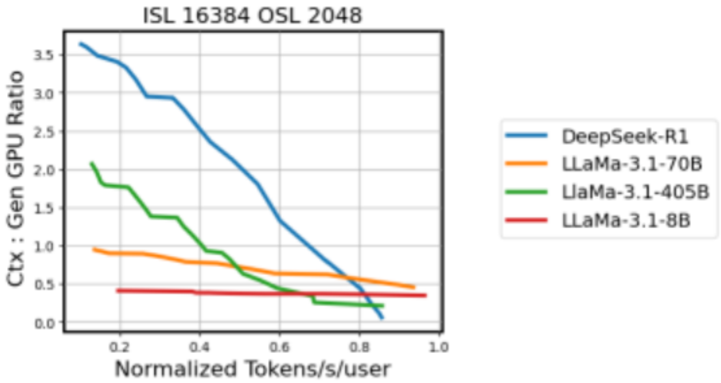



Figure 9: The optimal ratio of ctx-to-gen GPUs varies across models and target latencies

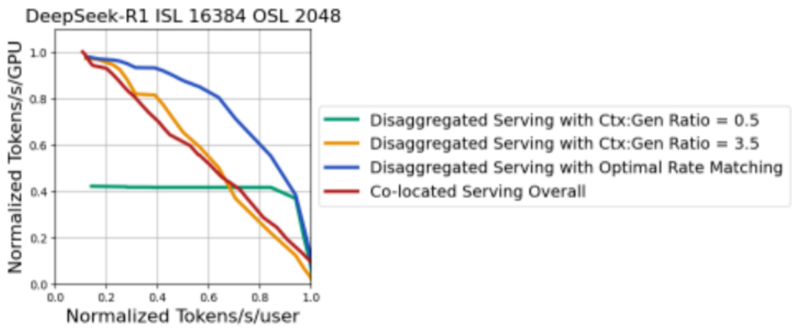



Figure 10: Optimal rate matching dynamically adapts Ctx:Gen ratio to deliver Pareto optimal performance. A ratio of 3.5 is performant at the most relaxed latency target but degrades as latency tightens. Conversely, a ratio of 0.5 favors tight latency but suffers significantly under relaxed latency.

# Evaluation – Network

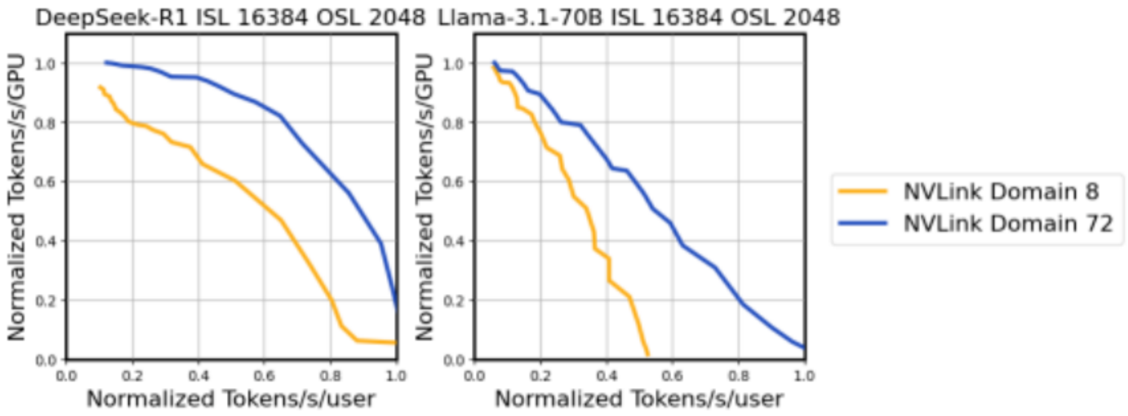



Figure 11: Larger NVLink domain helps disaggregated serving performance. DeepSeek-R1 benefits from higher EP and batching at medium-latency. Llama-3.1-70B benefits from high TP at low-latency.

# Critical Thinking Review

- Missing considering other factors
  - Hardware
  - Serving engines
  - KV Cache Reuse
  - Offloading
  - Heterogenous serving
- Metrics: cost, energy, P99 latency
- Workloads: real-world traffics
- Assumptions: transfer latency, failure

# Homework

- Read the related materials, and answer the following questions:
  - 1. What benefits and limits does DistServe propose for its design?
  - 2. By comparing the evaluation results in DistServe and TaiChi, what differences are there? What can be the potential reasons?
  - 3. How does *Throughput is Not All You Need* argue for its idea? Do you think the arguments hold in real serving scenarios? What are your reasons?
- Related materials:
  - DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving
  - Prefill-Decode Aggregation or Disaggregation? Unifying Both for Goodput-Optimized LLM Serving
  - Throughput is Not All You Need: Maximizing Goodput in LLM Serving using Prefill-Decode Disaggregation

# Q & A