Chapter 7: Request-Level Routing and SGLang

In the previous chapter, we covered vLLM, which uses model parallelism (TP/PP/DP/EP) to distribute large models across GPUs. vLLM excels at high-throughput workloads with large batches and is optimized for models that require multiple GPUs just to fit in memory.

But what if you need ultra-low latency for interactive chat applications? Or what if your models fit on a single GPU but you need to handle thousands of concurrent requests with session persistence? That's where SGLang comes in. This chapter introduces SGLang's distributed inference architecture, focusing on router-based request routing, prefill/decode disaggregation, and multi-node coordination. Unlike vLLM's model parallelism approach, SGLang emphasizes request-level routing and workload disaggregation for high-QPS, low-latency distributed inference.

Introduction to SGLang and Setup

SGLang (Structured Generation Language) is a high-performance inference engine optimized for low-latency, high-QPS workloads. It uses a router-based distributed architecture that emphasizes request routing, session affinity, and workload disaggregation rather than model weight sharding.

Prerequisites

Before installing SGLang, ensure you have:

- **OS**: Linux (required for GPU support)
- **Python**: 3.10+
- **NVIDIA GPU**: With CUDA support (for CUDA-based installation)
- **CUDA**: Compatible CUDA version installed

Installation

SGLang can be installed using several methods. **Docker is the quickest way to try SGLang** without installing dependencies locally.

Docker Setup

Docker provides the quickest way to get started with SGLang without installing dependencies locally. Pre-built images are available on the SGLang Docker Hub page.

SGLang supports seven main types of models. Base models like `meta-llama/Llama-3.2-1B` are pre-trained language models without instruction tuning, and they use the `/v1/completions` endpoint with a `prompt` parameter. Chat models such as `Qwen/Qwen2.5-0.5B-Instruct` are fine-tuned for conversational tasks and use `/v1/chat/completions` with a `messages` parameter. Embedding models like `Qwen/Qwen3-Embedding-0.6B` generate vector representations and use `/v1/embeddings` with an `input` parameter. Diffusion language models like `inclusionAI/LLaDA2.0-mini` enable non-autoregressive text generation with parallel decoding and use the `/generate` endpoint with a `text` parameter. Multimodal/vision language models (VLMs) like `meta-llama/Llama-3.2-11B-Vision-Instruct` accept multimodal inputs (images, videos, and text) and use `/v1/chat/completions` with multimodal content in the `messages` parameter. Rerank models like `BAAI/bge-reranker-v2-m3` rerank search results based on semantic relevance and use the `/v1/rerank` endpoint with a `query` and `documents` parameters. Reward models like `jason9693/Qwen2.5-1.5B-apeach` output scalar reward scores for reinforcement learning or content moderation and use `/v1/embeddings` with the `--is-embedding` flag. Additionally, SGLang also supports video and image generation through SGLang Diffusion, which accelerates diffusion models for tasks like text-to-image, image-to-image, and text-to-video generation.

The SGLang server exposes an OpenAI-compatible API with endpoints for completions, chat completions, embeddings, reranking, classification, and more. For a complete list of endpoints with detailed descriptions and usage examples, see the **OpenAI-Compatible API Endpoints** section in the Appendix.

Pull the Latest Image

Pull the latest Docker image. The image requires approximately 16GB of disk space.

```
docker pull lmsysorg/sglang:latest-runtime
```

`lmsysorg/sglang:latest` is actually development docker with build tools and development dependencies and around 35GB. SGLang is larger primarily because it includes Triton Inference Server and its dependencies, which provide more serving infrastructure but add significant size. vLLM uses a lighter base and focuses on a smaller dependency set, resulting in a smaller image.

Run the Docker Container

The Docker image runs an OpenAI-compatible server. To avoid hardcoding the model name, set it as an environment variable:

```
export SGLANG_MODEL="Qwen/Qwen2.5-0.5B-Instruct"
```

Then run the container:

```
docker run --runtime nvidia --gpus all \
  -v $HOME/.cache/huggingface:/root/.cache/huggingface \
  --env "HF_TOKEN=$HF_TOKEN" \
  --env "SGLANG_MODEL=$SGLANG_MODEL" \
  -p 30000:30000 --ipc=host --shm-size 32g \
  lmsysorg/sglang:latest-runtime \
  python3 -m sglang.launch_server \
    --model-path $SGLANG_MODEL \
    --host 0.0.0.0 \
    --port 30000
```

> **Note:** SGLang does not support OPT models (e.g., `facebook/opt-125m`). SGLang supports architectures such as Llama, Mistral, Qwen, Gemma, Phi3, and others. Some models may require `--trust-remote-code` flag. See the SGLang documentation for the full list of supported model architectures.

Here are some small models suitable for learning purposes:

Model Name	Type	Parameter Size
`Qwen/Qwen2.5-0.5B-Instruct`	Chat/Instruct	0.5B
`meta-llama/Llama-3.2-1B-Instruct`	Chat/Instruct	1B
`meta-llama/Llama-3.2-1B`	Base	1B
`microsoft/Phi-tiny-MoE-instruct`	MoE/Instruct	3.8B total, ~1.1B active
`Qwen/Qwen3-Embedding-0.6B`	Embedding	0.6B
`Qwen/Qwen2-VL-2B-Instruct`	VLM/Multimodal	2B
`BAAI/bge-reranker-v2-m3`	Rerank	0.6B
`jason9693/Qwen2.5-1.5B-apeach`	Reward/Classify	1.5B

To use any of these models, replace the model name in the Docker command:

```
... --model-path <MODEL_NAME> ...
```

For example:

```
... --model-path meta-llama/Llama-3.2-1B-Instruct ...
```

The `--runtime nvidia --gpus all` flag enables GPU access. To use specific GPUs, replace `--gpus all` with `--gpus '"device=0"'` for a single GPU or `--gpus '"device=0,1"'` for multiple GPUs. You can also

set `--env "CUDA_VISIBLE_DEVICES=0,1"` to limit visible GPUs.

The `-v $HOME/.cache/huggingface:/root/.cache/huggingface` volume mount shares your local Hugging Face cache with the container, avoiding repeated model downloads. The container path `/root/.cache/huggingface` assumes the container runs as root. Adjust this path if your container uses a different user, or set the `HF_HOME` environment variable to customize the cache location.

The `--ipc=host` flag allows the container to access the host's shared memory, which PyTorch uses for efficient data sharing during tensor parallel inference.

The `--shm-size 32g` flag sets the shared memory size, which is important for SGLang's KV cache management and RadixAttention features.

Verify the Setup

Once the container is running, verify it's working correctly. First, check that the server is responding:

```
curl -w "HTTP Status: %{http_code}\n" http://localhost:30000/health
```

List the available models:

```
curl http://localhost:30000/v1/models
```

Test a base model completion:

```
curl http://localhost:30000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "'"$SGLANG_MODEL"'",
    "prompt": "The result of 1+1 is",
    "max_tokens": 3
  }'
```

Test a chat model:

```
curl http://localhost:30000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "'"$SGLANG_MODEL"'",
    "messages": [
      {"role": "user", "content": "Hello, how are you?"}
    ]
  }'
```

For deterministic output (same result every time), add `"temperature": 0`:

```
curl http://localhost:30000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "'"$SGLANG_MODEL"'",
    "messages": [
      {"role": "user", "content": "Hello, how are you?"}
    ],
    "temperature": 0
  }'
```

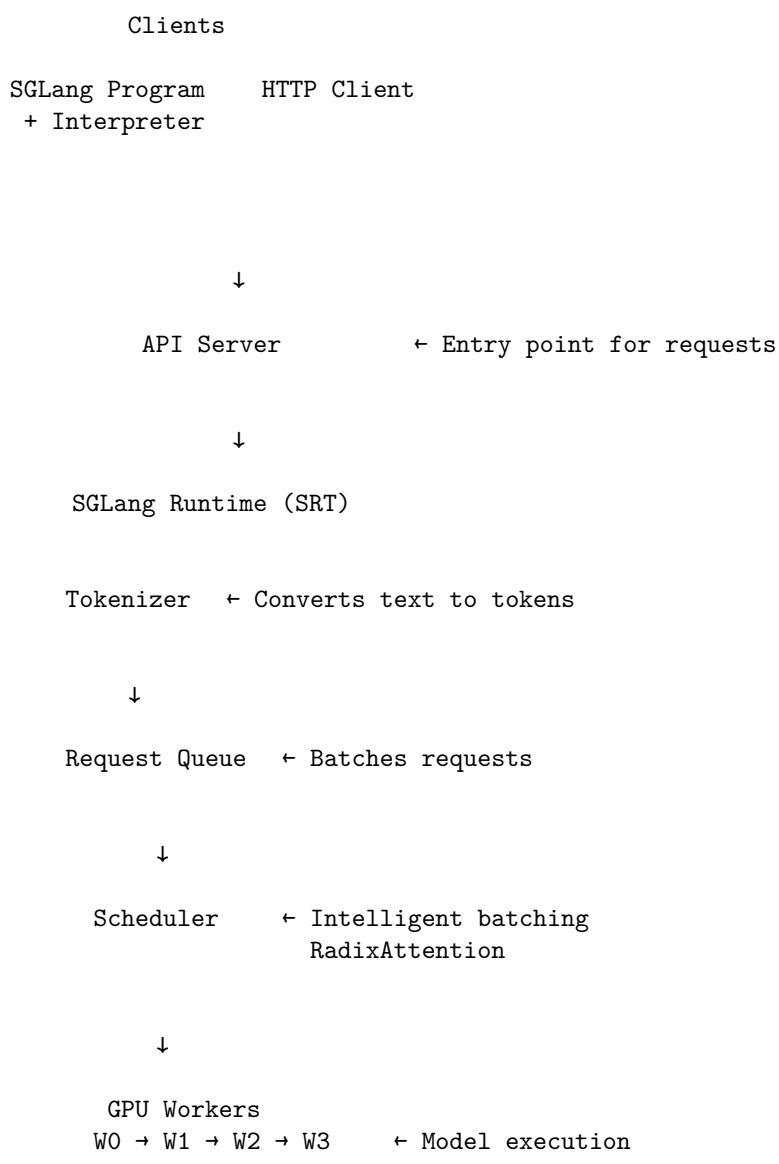
In addition to the models listed in the table, SGLang also supports video and image generation through SGLang Diffusion (though these models are not shown in the small models table above). For each model type, there are slightly different parameters for the server and client. For detailed launch commands, API

usage, and model-specific requirements, refer to the SGLang documentation under the "Supported Models" section.

For alternative installation methods (pip, uv, or from source), please refer to the SGLang installation guide.

Overview of the SGLang Architecture

SGLang's architecture follows a **frontend-backend** design pattern, as described in the Hugging Face blog post on SGLang. The system consists of an **API Server** (frontend) that handles client requests and the **SGLang Runtime (SRT)** (backend) that executes inference:



```
        ↓

    API Server    ← Returns responses
```

Key Components:

- **API Server**: The entry point that receives requests from SGLang programs (via Interpreter) or HTTP clients, and returns detokenized responses.
- **SGLang Runtime (SRT)**: The backend execution engine containing:
  - **Tokenizer**: Converts incoming text into numerical tokens for the model
  - **Request Queue**: Buffers tokenized requests for batching and managing concurrency
  - **Scheduler**: Intelligently batches requests, prioritizes tasks that benefit from KV cache reuse (RadixAttention), and minimizes tail latency. Implements zero-overhead scheduling with CPU/GPU overlap.
  - **GPU Workers**: Execute model inference on GPUs. Can be organized for tensor parallelism (W0 $\rightarrow$ W1 $\rightarrow$ W2 $\rightarrow$ W3) or as independent workers for data parallelism.
  - **Detokenizer**: Converts generated tokens back into human-readable text

For Distributed Deployments:

When scaling across multiple nodes, SGLang adds a **Router (Model Gateway)** layer above the API Server:

- **Router (Model Gateway)**: Routes requests across multiple SRT instances
  - **Control Plane**: Worker Manager, Load Monitor, Health Checker
  - **Data Plane**: HTTP/gRPC routers with load balancing policies (cache-aware, power-of-two, round-robin)
  - Maintains session affinity for KV cache locality

Key Architectural Differences from vLLM:

- **vLLM**: Uses a scheduler-executor-worker pattern focused on model parallelism (TP/PP/DP) to split model weights across GPUs. The executor coordinates distributed inference with weight synchronization via all-reduce operations.
- **SGLang**: Uses a frontend-backend pattern with request-level optimizations. The SRT scheduler focuses on intelligent batching with RadixAttention for prefix cache reuse, rather than model weight sharding. For distributed deployments, the Router distributes requests based on load, cache locality, and session affinity, making it better suited for high-QPS, low-latency workloads.

SGLang Core Theory

SGLang's architecture is built around several core innovations that enable high-performance inference: RadixAttention for KV cache reuse, structured output decoding with X-Grammar, graph-based Intermediate Representation (IR), operator fusion, and an advanced scheduler with CPU/GPU overlap. These techniques work together to significantly improve throughput and reduce latency. **While vLLM focuses on optimizing model parallelism (TP/PP) and memory efficiency (PagedAttention), SGLang emphasizes request-level optimizations like RadixAttention and router-based routing**, making it better suited for high-QPS, low-latency workloads where request routing and cache locality matter more than model weight distribution.

RadixAttention: Prefix Cache Reuse

RadixAttention is one of SGLang's most significant innovations, enabling efficient KV cache reuse across requests that share common prefixes. This is particularly valuable in scenarios where multiple requests share the same prompt prefix, such as AI agents with system prompts or few-shot learning examples. **Unlike vLLM's PagedAttention which optimizes memory efficiency within a single request's KV**

cache, RadixAttention optimizes across multiple requests by sharing common prefixes, making it especially effective for high-QPS scenarios with shared system prompts or few-shot examples.

The Problem: Redundant KV Cache Computation

In traditional inference systems, each request independently computes KV cache for all tokens, including shared prefixes. Consider a scenario where 100 users interact with an AI assistant:

- Each user's request starts with the same system prompt (e.g., "You are a helpful assistant…")
- Traditional systems compute this prefix's KV cache 100 times
- This wastes computation and memory

RadixAttention Solution

RadixAttention uses a **radix tree (prefix tree)** data structure to store and share KV cache across requests:

Key Concepts:

1. **Radix Tree Structure**: KV cache is organized as a tree where:
   - Each node represents a token sequence
   - Common prefixes are stored once and shared
   - Leaf nodes represent unique request suffixes
2. **Prefix Matching**: When a new request arrives:
   - The system finds the longest matching prefix in the radix tree
   - Reuses existing KV cache for the matched prefix
   - Only computes KV cache for new tokens
3. **Dynamic Tree Updates**: As requests complete:
   - Shared prefixes remain in the tree for future reuse
   - Unique suffixes are evicted when no longer needed
   - Tree structure adapts to request patterns

Example:

```
Request 1: "You are helpful. What is Python?"
Request 2: "You are helpful. Explain ML."
Request 3: "You are helpful. Write code."

Radix Tree:
Root
   "You are helpful. "
      "What is Python?" (Request 1)
      "Explain ML." (Request 2)
      "Write code." (Request 3)

KV Cache:

- "You are helpful. " computed once, shared by all 3 requests
- Each unique suffix computed separately
```

Implementation Details:

```python
class RadixCache:
    """Radix tree for KV cache prefix sharing."""

    def match_prefix(self, key: RadixKey) -> MatchResult:
        """Find longest cached prefix matching the key."""
        # Traverse tree to find matching prefix
        # Returns indices of cached KV cache
```

```
        pass

    def insert(self, key: RadixKey, kv_indices: torch.Tensor):
        """Insert new prefix into radix tree."""
        # Split nodes if necessary
        # Update tree structure
        pass
```

Source Code Location:

The RadixAttention implementation can be found in the SGLang source code:

- **RadixCache class**: `python/sglang/srt/mem_cache/radix_cache.py` - Main radix tree implementation for KV cache prefix sharing
- **RadixAttention layer**: `python/sglang/srt/layers/radix_attention.py` - Attention layer that uses RadixCache
- **RadixKey and MatchResult**: Defined in `python/sglang/srt/mem_cache/radix_cache.py`

Performance Benefits:

- **Computation Savings**: Up to 90% reduction in prefill computation for shared prefixes
- **Memory Efficiency**: Shared prefixes stored once instead of per-request
- **Latency Reduction**: 2-3x faster for requests with cache hits
- **Throughput Improvement**: Enables larger batch sizes by reducing per-request memory

Radix Cache-Aware Scheduling:

SGLang's scheduler uses **longest prefix matching** to optimize batch formation:

1. Requests are sorted by longest matching prefix length
2. Requests with longer shared prefixes are prioritized
3. This maximizes cache hit rates and GPU utilization

```
def get_next_batch_to_run(self):
    # Sort waiting queue by longest prefix match
    waiting_queue.sort(key=lambda r: len(r.prefix_indices), reverse=True)

    # Select requests with longest shared prefixes
    batch = []
    for req in waiting_queue:
        if can_fit_in_batch(req):
            batch.append(req)
    return batch
```

Source Code Location:

The cache-aware scheduling implementation can be found in:

- **SchedulePolicy class**: `python/sglang/srt/managers/schedule_policy.py` - Contains `_compute_prefix_matches()` and `_sort_by_longest_prefix()` methods
- **Request initialization**: `python/sglang/srt/managers/schedule_batch.py` - The `Req.init_next_round_input()` method calls `tree_cache.match_prefix()` to find cached prefixes

Cache Locality Optimization:

- **Session Affinity**: Requests from the same session are routed to the same worker
- **Prefix Persistence**: Hot prefixes stay in cache longer
- **Eviction Policies**: LRU or custom policies for cache management

Structured Output Decoding with X-Grammar

Structured Output Decoding (also called **Constraint Decoding**) enables LLMs to generate outputs that conform to specific formats, such as JSON, SQL queries, or custom grammars. SGLang's **X-Grammar** framework provides an efficient implementation that supports any Context-Free Grammar (CFG). X-Grammar uses **Finite State Machines (FSMs)** for simple rules and **Pushdown Automata (PDAs)** for complex rules requiring stack state, ensuring guaranteed syntactically correct output.

The Problem: Naive Constraint Decoding

The simplest approach to constraint decoding involves:

1. Generate candidate tokens
2. Check each token against grammar rules
3. Mask invalid tokens (set probability to near zero)
4. Sample from filtered distribution

Scalability Issues:

- For Llama-3 with 128K vocabulary, checking every token at each step is computationally prohibitive
- Even with likelihood-based sorting, asymptotic complexity remains problematic
- Stack management for nested structures requires expensive duplication

X-Grammar Solution

X-Grammar introduces major improvements in both **rule expressiveness** and **system efficiency**:

1. Context-Free Grammar Support:

X-Grammar supports any CFG, not just JSON:

```
# JSON example
json_grammar = """
value -> object | array | string | number | boolean | null
object -> "{" (pair ("," pair)*)? "}"
pair -> string ":" value
array -> "[" (value ("," value)*)? "]"
"""

# SQL example
sql_grammar = """
select_stmt -> SELECT column_list FROM table_name WHERE condition
column_list -> column ("," column)*
condition -> column operator value
"""
```

2. Adaptive Token Mask Cache (FSM-based):

For context-free rules (where token validity depends only on current state), X-Grammar uses **Finite State Machines (FSMs)** to precompile valid tokens:

```
# Example: boolean value rule
bool_value -> "true" | "false"

# Precompiled cache:
# State: bool_value
# Valid tokens: ["true", "false"]
# No runtime validation needed for 75%+ of tokens
```

3. Pushdown Automaton (PDA) for Context-Dependent Rules:

For rules requiring stack state (e.g., matching parentheses), X-Grammar uses **Pushdown Automata (PDAs)**, which extend FSMs with a stack to handle context-dependent validation:

```
# Example: balanced parentheses
S -> "(" S ")" S |

# PDA stack tracks unmatched "("
# ")" validity depends on stack state
```

4. Tree-Based Stack Management:

Traditional stack snapshotting requires costly duplication. X-Grammar uses tree-based management with node reuse:

- Reduces memory copies by 90%
- Enables efficient backtracking
- Supports branching and rollback

5. Optimizations:

- **State Inlining**: Embed simple non-terminals into parent rules
- **Equivalent State Merging**: Combine identical PDA states
- **Context Inference**: Convert stack-dependent tokens to context-free where possible
- **Parallel Grammar Compilation**: Distribute compilation across multi-core CPUs
- **GPU Computation Overlap**: Overlap CPU grammar processing with GPU computation

Usage Example:

```
import sglang as sgl

# Define JSON schema
schema = {
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "age": {"type": "number"},
        "city": {"type": "string"}
    },
    "required": ["name", "age"]
}

# Generate structured output
response = sgl.generate(
    prompt="Extract information: John is 30 years old, lives in NYC",
    grammar=schema,
    max_tokens=100
)

# Output guaranteed to be valid JSON matching schema
print(response)  # {"name": "John", "age": 30, "city": "NYC"}
```

Performance Benefits:

- **Guaranteed Validity**: Output always conforms to specified grammar
- **Efficient Validation**: 75%+ tokens validated via precompiled cache
- **Low Overhead**: Minimal impact on generation speed
- **Flexible**: Supports any CFG, not just JSON

Source Code Location:

- **X-Grammar backend**: `python/sglang/srt/constrained/xgrammar_backend.py` - Main X-Grammar implementation
- **Grammar compilation**: Uses the `xgrammar` library for CFG compilation and PDA construction
- **Grammar integration**: `python/sglang/srt/managers/scheduler_output_processor_mixin.py` - Grammar token acceptance during decoding

Graph-Based Intermediate Representation (IR)

SGLang uses a compact graph-based IR that:

- Compiles operator sequences into fused kernels at compile time
- Reduces kernel launch latency significantly
- Enables aggressive operator fusion optimizations

Source Code Location:

- **IR definition**: `python/sglang/lang/ir.py` - Core IR data structures
- **IR compilation**: `python/sglang/srt/compilation/compile.py` - Compilation passes
- **Backend compilation**: `python/sglang/srt/compilation/backend.py` - Backend-specific compilation
- **Pass manager**: `python/sglang/srt/compilation/pass_manager.py` - Optimization passes

IR Structure:

```
# Example IR graph
graph = {
    "nodes": [
        {"op": "embed", "inputs": ["input_ids"]},
        {"op": "layernorm", "inputs": ["embed"]},
        {"op": "linear", "inputs": ["layernorm"]},
        {"op": "gelu", "inputs": ["linear"]},
    ],
    "fused_kernels": [
        "layernorm_linear_gelu"  # Fused into single kernel
    ]
}
```

Operator Fusion

Why Operator Fusion Matters:

- **Reduces Kernel Launches:** Combining small operations into single kernels eliminates launch overhead
- **Lowers Memory Traffic:** Intermediate buffers are combined, reducing read/write operations
- **Enables Custom Optimizations:** Fused kernels (e.g., layernorm + linear + activation) outperform generic compositions

Example:

```
# Fused layernorm + linear + activation
class FusedLayerNormLinearActivation(nn.Module):
    def forward(self, x):
        # All operations fused into single kernel
        x = self.layernorm(x)
        x = self.linear(x)
        x = self.activation(x)
        return x
```

Common Fusion Patterns:

- `layernorm + linear + activation`: Most common pattern
- `attention + output_projection`: Reduces memory traffic
- `mlp_up + gelu + mlp_down`: Complete MLP fusion

Source Code Location:

- **Fused operations**: `sgl-kernel/csrc/` - CUDA/C++ fused kernel implementations
- **MoE fusion**: `sgl-kernel/csrc/moe/` - Fused MoE operations (e.g., `moe_fused_gate.cu`, `fused_experts_kernel_impl`)
- **Custom ops**: `python/sglang/srt/custom_op.py` - Custom operator base class with fusion support
- **TopK fusion**: `sgl-kernel/python/sgl_kernel/top_k.py` - Fused top-k operations

Zero-Overhead Scheduler

SGLang's scheduler has evolved from serial CPU/GPU execution to a zero-overhead design that overlaps CPU scheduling with GPU computation.

Serial Scheduler (Early Versions)

Traditional inference systems execute steps serially:

```
Pre-schedule → Compute batch → Sample → Post-schedule
     ↓              ↓             ↓          ↓
   CPU            GPU           GPU        CPU
```

Problems:

- GPU idle time while CPU schedules next batch
- Scheduler overhead can consume 50%+ of total time
- Poor GPU utilization

Zero-Overhead Scheduler (Current)

SGLang's zero-overhead scheduler overlaps CPU scheduling with GPU computation:

Pipeline Stages:

1. **Pre-schedule (CPU-S)**:
   - Collect incoming requests
   - Schedule using Radix tree and longest prefix matching
   - Allocate memory for tokens
2. **Compute batch (GPU)**:
   - Forward pass on GPU
   - Generate logits
3. **Sample (GPU)**:
   - Sample next tokens from logits
   - Apply constraint decoding if needed
4. **Post-schedule (CPU-S)**:
   - Check finish conditions
   - Remove completed requests
   - Update cache

Overlap Strategy:

```
Batch 1: Pre-schedule → Compute → Sample → Post-schedule
Batch 2:              Pre-schedule → Compute → Sample → Post-schedule
Batch 3:                           Pre-schedule → Compute → Sample
```

Implementation:

SGLang splits CPU into two parts:

- **CPU-S (Scheduler CPU)**: Handles Pre-schedule and Post-schedule
- **CPU-L (Launch CPU)**: Launches kernels and processes results

```
# Overlap event loop
last_batch = None
while True:
    # Pre-schedule (CPU-S)
    recv_reqs = recv_requests()
    process_input_requests(recv_reqs)
    batch = get_next_batch_to_run()

    # Compute and sample (CPU-L + GPU, async)
    result = run_batch(batch)  # Returns immediately
    result_queue.put((batch, result))

    # Post-schedule (CPU-S, overlaps with GPU)
    if last_batch is not None:
        tmp_batch, tmp_result = result_queue.get()
        process_batch_result(tmp_batch, tmp_result)

    last_batch = batch
```

Performance Benefits:

- **Zero GPU Idle Time**: GPU always has work to do
- **Hidden Scheduling Overhead**: CPU scheduling overlaps with GPU computation
- **Higher Throughput**: Up to 2x improvement over serial scheduling
- **Lower Latency**: Reduced end-to-end latency

Source Code Location:

- **Scheduler main class**: `python/sglang/srt/managers/scheduler.py` - Main scheduler implementation
- **Schedule policy**: `python/sglang/srt/managers/schedule_policy.py` - Scheduling policies and batch formation
- **Schedule batch**: `python/sglang/srt/managers/schedule_batch.py` - Batch data structures and request management
- **Overlap utilities**: `python/sglang/srt/managers/overlap_utils.py` - CPU/GPU overlap utilities

Token Placeholder Mechanism

To enable async execution, SGLang uses token placeholders:

```
class TpModelWorkerClient:
    def forward_batch_generation(self, batch):
        # Send batch to GPU (async)
        input_queue.put(batch)

        # Return placeholder tokens immediately
        placeholder_tokens = self.allocate_placeholders(batch)
        return placeholder_tokens

    def forward_thread_func_(self):
        # Async forward pass
        while True:
            batch = input_queue.get()
            logits = model.forward(batch)
```

```
            tokens = sample(logits)

            # Replace placeholders with actual tokens
            self.replace_placeholders(batch, tokens)
            output_queue.put((batch, tokens))
```

Source Code Location:

- **TP Worker**: `python/sglang/srt/managers/tp_worker.py` - Tensor parallel worker with async execution
- **TP Worker overlap thread**: `python/sglang/srt/managers/tp_worker_overlap_thread.py` - Overlap thread implementation
- **Request structure**: `python/sglang/srt/managers/schedule_batch.py` - Request data structures with placeholder support

Lightweight Scheduler

SGLang uses an event loop scheduler that:

- Batches small work items efficiently
- Dispatches fused kernels with minimal overhead
- Optimizes for low latency rather than maximum throughput
- Supports dynamic batch formation with continuous batching

Source Code Location:

- **Scheduler implementation**: `python/sglang/srt/managers/scheduler.py` - Main scheduler with event loop
- **Scheduler enhancer**: `python/sglang/srt/managers/scheduler_enhancer.py` - Scheduler optimizations
- **Schedule policy**: `python/sglang/srt/managers/schedule_policy.py` - Policy implementations for batch selection

Router-Based Distributed Architecture

Unlike vLLM's model parallelism (TP/PP/DP) which focuses on splitting model weights, SGLang uses a **router-based architecture** that emphasizes request-level routing, session affinity, and workload disaggregation. This approach is optimized for high-QPS, low-latency workloads where request routing matters more than model sharding.

Key Distinction: While vLLM's model parallelism (TP/PP) splits model weights across GPUs and requires synchronization between workers, SGLang's Router uses request-level routing to distribute requests across independent workers. This makes SGLang better suited for high-QPS, low-latency distributed inference scenarios where request routing and cache locality are more important than model weight distribution.

Key Architectural Differences from vLLM

vLLM (Chapter 6) focuses on:

- **Model-level parallelism**: How to split model weights (TP/PP/DP/EP)
- **Memory efficiency**: PagedAttention for KV cache
- **Throughput optimization**: Continuous batching for large batches

SGLang (This chapter) focuses on:

- **Request-level routing**: How to route requests to optimize latency and cache locality
- **Workload disaggregation**: Separating prefill and decode workloads
- **Session management**: Maintaining KV cache locality through routing

SGLang Model Gateway Architecture

SGLang Model Gateway (formerly SGLang Router) is a high-performance model-routing gateway for large-scale LLM deployments. It centralizes worker lifecycle management, balances traffic across heterogeneous protocols (HTTP, gRPC, OpenAI-compatible), and provides enterprise-ready control over history storage, MCP tooling, and privacy-sensitive workflows.

Comparison with vLLM: Unlike vLLM's model parallelism approach where workers must synchronize model weights via all-reduce operations (TP) or pipeline stages (PP), SGLang's Model Gateway routes requests to independent workers without requiring weight synchronization. This request-level routing approach eliminates communication overhead between workers, making it ideal for high-QPS scenarios where latency matters more than maximizing single-request throughput.

The gateway architecture is organized into **Core Runtime Features** and **Enterprise Extensions**:

Core Runtime Features

The core runtime provides essential distributed inference capabilities:

Control Plane

The control plane manages worker registration, monitoring, and orchestration:

1. Worker Manager - Discovers worker capabilities via `/get_server_info` and `/get_model_info` endpoints - Tracks real-time load statistics for each worker - Registers and removes workers from the shared registry - Validates worker health and capabilities

2. Job Queue - Serializes add/remove worker requests to prevent race conditions - Exposes status via `/workers/{url}` endpoint for tracking onboarding progress - Manages background operations asynchronously

3. Load Monitor - Feeds cache-aware and power-of-two policies with live worker load statistics - Tracks pending requests, active sessions, and resource utilization - Updates policies in real-time based on current system state

4. Health Checker - Continuously probes workers to verify availability - Updates worker readiness status and circuit breaker state - Maintains router metrics for observability - Supports configurable health check intervals and timeouts

5. Service Discovery (Kubernetes) - Optional Kubernetes integration for automatic worker discovery - Keeps registry aligned with pod lifecycle - Supports independent selectors for regular and PD workloads

Data Plane

The data plane routes traffic across multiple protocols with shared reliability primitives:

1. HTTP Routers - Regular Router: Handles classic single-stage workers with per-model policy overrides - Implements `/generate`, `/v1/chat/completions`, `/v1/completions`, `/v1/responses`, `/v1/embeddings`, `/v1/rerank` - Supports streaming and non-streaming modes - Full OpenAI-compatible API surface

- **Prefill/Decode Router**: Coordinates disaggregated prefill and decode workers
  - Merges metadata from prefill and decode phases
  - Manages streaming fan-in for token generation
  - Handles KV cache transfer coordination

2. gRPC Router (Industry-First) - Fully Rust implementation of OpenAI-compatible gRPC inference gateway - Native in-process tokenizer, reasoning parser, and tool parser execution - Streams tokenized requests directly to SRT gRPC workers - Supports both single-stage and PD (prefill/decode) topologies - Built-in reasoning parsers for DeepSeek, Qwen, Llama, Mistral, GPT-OSS, Step-3, GLM4, Kimi K2, and

other structured-thought models - Tool-call parsers for JSON, Pythonic, XML, and custom schemas - Tokenizer factory supporting HuggingFace models, local tokenizer.json files, and chat template overrides - Provides same `/v1/*` APIs as HTTP router with higher throughput

3. OpenAI Router - Proxies OpenAI-compatible endpoints to external vendors (OpenAI, xAI, Gemini, etc.) - Preserves headers and SSE streams end-to-end - Keeps chat history and multi-turn orchestration local - Supports `/v1/responses` background jobs with cancellation, deletion, and listing - Enables agentic, multi-turn orchestration without persisting data at remote vendor endpoints

4. Router Manager - Coordinates multiple router implementations when Inference Gateway Mode (`--enable-igw`) is enabled - Dynamically instantiates multiple router stacks (HTTP regular/PD, gRPC) - Applies per-model policies for multi-tenant deployments - Manages router selection based on request headers and model ID

Enterprise Extensions

The following features are enterprise extensions that extend the core runtime with advanced capabilities:

1. History Connectors (Enterprise Extension) - Centralizes chat history inside the router tier - Supports multiple storage backends: in-memory, disabled (none), or **Oracle ATP (enterprise extension)** - Enables context reuse across models and MCP loops without leaking data to upstream vendors - Conversation and response history stored at router boundary for enterprise privacy

2. MCP Integration (Enterprise Extension) - **MCP (Model Context Protocol)** integration for advanced tooling and agentic workflows - Native MCP client supporting all transport protocols (STDIO, HTTP, SSE, Streamable) - Tool execution loops with reasoning parser integration - Privacy-preserving tool execution within router boundary

3. Inference Gateway Mode (Enterprise Extension) - **Inference Gateway Mode (`--enable-igw`)** for multi-model, multi-tenant deployments - Dynamically instantiates multiple router stacks (HTTP regular/PD, gRPC) - Applies per-model policies for multi-tenant deployments - Supports heterogeneous model fleets - Enables independent scaling per model type

4. Enterprise Privacy Features (Enterprise Extension) - Agentic multi-turn `/v1/responses` API - History storage operates within router boundary - Compliant data sharing across models/MCP loops - Advanced security and authentication mechanisms

Reliability & Flow Control

1. Retry Mechanism - Exponential backoff with jitter - Configurable retry counts and timeouts - Request-level retry tracking

2. Circuit Breakers - Worker-scoped circuit breakers - Automatic failover when workers become unhealthy - Configurable failure thresholds

3. Rate Limiting - Token-bucket rate limiting with queuing - Per-worker and global rate limits - Request queuing when rate limits are exceeded

4. Background Health Checks - Continuous worker health monitoring - Automatic worker removal on persistent failures - Cache-aware load monitoring

Observability

1. Prometheus Metrics - Request-level metrics (latency, throughput, errors) - Worker-level metrics (load, health, cache hit rates) - Router-level metrics (queue depth, policy decisions) - Detailed job queue statistics

2. Structured Tracing - OpenTelemetry trace support - Request ID propagation across router and workers - End-to-end request tracing

3. Logging - Structured logging for all router operations - Request/response logging with configurable verbosity - Error tracking and debugging information

Why Router-Based Architecture?

Advantages over Model Parallelism:

1. **Cache Locality**: Session affinity keeps KV cache on the same worker, avoiding expensive transfers
2. **Independent Scaling**: Scale prefill and decode workers independently based on workload
3. **Fault Tolerance**: Router can route around failed workers without model re-sharding
4. **Flexible Routing**: Implement custom routing policies (priority, SLA, A/B testing)
5. **Lower Latency**: Avoids synchronization overhead of model parallelism for small-to-medium models

When Router-Based Works Best:

- Models fit on single GPU or small TP group (2-4 GPUs)
- High QPS with many concurrent sessions
- Latency-sensitive workloads (chat, interactive applications)
- Need for session persistence and cache locality

When Model Parallelism (vLLM) is Better:

- Very large models requiring TP/PP across many GPUs
- Throughput-optimized workloads with large batches
- Memory-constrained environments needing PagedAttention

Prefill/Decode Disaggregation

Prefill/Decode (PD) disaggregation is a key distributed architecture pattern in SGLang that separates the computationally intensive prefill phase from the decode phase, enabling independent scaling and better resource utilization. **Unlike vLLM's pipeline parallelism (PP) which splits model layers across stages and requires strict synchronization, SGLang's PD disaggregation separates workload types (prefill vs decode) rather than model layers**, allowing independent scaling of compute-intensive prefill workers and latency-optimized decode workers without the synchronization overhead of pipeline stages.

Motivation for PD Disaggregation

The Problem:

- Prefill (initial prompt processing) is compute-intensive and can block decode workers
- Decode (token generation) requires low latency and high throughput
- In unified scheduling, prefill batches frequently interrupt ongoing decode batches, causing substantial delays
- In data-parallel attention, one DP worker may process prefill while another handles decode, leading to increased decode latency

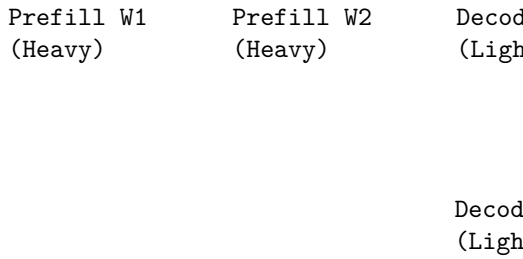
The Solution:

- Separate prefill workers handle initial prompt processing
- Dedicated decode workers handle token generation
- Router intelligently routes requests to appropriate workers
- KV cache is transferred from prefill workers to decode workers using high-performance transfer engines

PD Disaggregation Architecture

```
Router

↓              ↓              ↓
```



Benefits

1. **Independent Scaling**: Scale prefill and decode workers separately based on workload
2. **Better Resource Utilization**: Prefill workers can use more compute, decode workers optimize for latency
3. **Fault Isolation**: Failure in prefill workers doesn't affect decode workers
4. **Cost Optimization**: Use different hardware types for different workloads

Transfer Engines

In PD disaggregation, after a prefill worker processes the initial prompt and generates the KV cache, this cache must be transferred to a decode worker for token generation. **Transfer Engines** are the mechanisms responsible for efficiently moving KV cache data between prefill and decode workers over the network.

Why Transfer Engines Matter:

- **Performance**: KV cache can be large (especially for long contexts), so efficient transfer is critical for low latency
- **Network Optimization**: Different engines use different network protocols (RDMA, UCX) optimized for high-bandwidth, low-latency transfers
- **Hardware Compatibility**: Different engines support different hardware (GPUs, NPUs) and network fabrics

SGLang supports multiple transfer engines for KV cache transfer between prefill and decode workers:

- **Mooncake**: High-performance transfer engine using RDMA (Remote Direct Memory Access) for efficient, low-latency data transfers. Optimized for InfiniBand networks and provides direct memory-to-memory transfers without CPU involvement.
- **NIXL**: UCX-based transfer engine for flexible deployment across different network fabrics (InfiniBand, Ethernet, etc.). Provides a unified interface for high-performance communication.
- **ASCEND**: Specialized transfer engine for Ascend NPU deployments, optimized for Huawei Ascend hardware and network stacks.

PD Disaggregation Setup

Prerequisites:

For Mooncake backend:

```
uv pip install mooncake-transfer-engine
```

For NIXL backend:

```
pip install nixl
```

1. Start Prefill Workers (Mooncake):

```
# Prefill worker
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode prefill \
    --port 30000 \
    --disaggregation-ib-device mlx5_roce0
```

2. Start Decode Workers:

```
# Decode worker
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode decode \
    --port 30001 \
    --base-gpu-id 1 \
    --disaggregation-ib-device mlx5_roce0
```

3. Start Router with PD Disaggregation:

```
python -m sglang_router.launch_router \
    --pd-disaggregation \
    --prefill http://127.0.0.1:30000 \
    --decode http://127.0.0.1:30001 \
    --host 0.0.0.0 \
    --port 30000
```

For NIXL backend, replace `--disaggregation-ib-device` with `--disaggregation-transfer-backend nixl`.

Distributed Inference Architecture and Parallelism Strategies

SGLang provides comprehensive support for distributed inference through multiple parallelism strategies. Understanding how these strategies work together is crucial for building scalable inference systems. **While vLLM primarily relies on model parallelism (TP/PP) for distributed inference, SGLang combines model parallelism with router-based request routing**, offering both approaches: model parallelism for very large models and router-based routing for high-QPS scenarios where request-level distribution is more effective than weight sharding.

Parallelism Strategy Overview

SGLang supports four main parallelism dimensions:

1. **Tensor Parallelism (TP)**: Splits model weights across GPUs
2. **Pipeline Parallelism (PP)**: Splits model layers across GPUs/nodes
3. **Data Parallelism (DP)**: Replicates model across workers for different requests
4. **Expert Parallelism (EP)**: Distributes MoE experts across devices

Parallelism Dimensions:

```
Total GPUs = TP × PP × DP × EP

Example: 128 GPUs
- TP=8, PP=2, DP=4, EP=2
- 8 GPUs per TP group
- 2 pipeline stages
- 4 data parallel replicas
- 2 expert parallel groups
```

Tensor Parallelism: Weight Sharding

Tensor parallelism splits model weights across multiple GPUs, similar to Megatron's approach.

How TP Works:

1. **Weight Partitioning**: Each layer's weights split across TP ranks
2. **Communication Pattern**: All-reduce after each layer
3. **Memory Distribution**: Each GPU stores 1/TP of model weights

Attention with TP:

```
# QKV projection split across TP ranks
# Rank 0: QKV[:, :hidden_size//tp_size]
# Rank 1: QKV[:, hidden_size//tp_size:2*hidden_size//tp_size]
# ...

# Attention computation
q, k, v = split_qkv(input, tp_rank, tp_size)
attn_output = attention(q, k, v)

# All-reduce to gather full attention output
attn_output = all_reduce(attn_output, tp_group)
```

MLP with TP:

```
# Column parallel (up projection)
mlp_up_output = column_parallel_linear(input, tp_rank, tp_size)

# Row parallel (down projection)
mlp_down_output = row_parallel_linear(mlp_up_output, tp_rank, tp_size)
# Automatically all-reduces to gather full output
```

Communication Overhead:

- **Per-Layer All-Reduce**: Communication after every attention and MLP layer
- **Bandwidth Requirements**: High bandwidth needed for efficient TP
- **Topology Sensitivity**: Best performance when TP groups within NVLink domain

TP Configuration:

```
# Single node TP
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 8

# Multi-node TP
# Node 0
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 2 \
    --node-rank 0

# Node 1
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
```

```
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 2 \
    --node-rank 1
```

Source Code Location:

- **Tensor parallelization**: `python/sglang/srt/layers/model_parallel.py` - Main `tensor_parallel()` function that applies TP plans to model layers
- **TP group initialization**: `python/sglang/srt/distributed/parallel_state.py` - `initialize_model_parallel()` function that sets up TP communication groups
- **Column parallel linear**: `python/sglang/srt/layers/linear.py` - `ColumnParallelLinear` class for column-wise weight sharding (QKV projections, MLP up projection)
- **Row parallel linear**: `python/sglang/srt/layers/linear.py` - `RowParallelLinear` class for row-wise weight sharding (MLP down projection, output projection)
- **TP communication**: `python/sglang/srt/layers/communicator.py` - All-reduce operations and TP communication context (`AttnTpContext`)
- **TP worker**: `python/sglang/srt/managers/tp_worker.py` - Tensor parallel worker implementation with async execution
- **Model runner TP support**: `python/sglang/srt/model_executor/model_runner.py` - `apply_torch_tp()` method and TP-aware forward passes

Pipeline Parallelism: Layer Sharding

Pipeline parallelism splits model depth across multiple GPUs or nodes.

How PP Works:

1. **Layer Assignment**: Contiguous layers assigned to different stages
2. **Micro-batch Execution**: Micro-batches flow through pipeline stages
3. **Communication**: Point-to-point communication between stages

Pipeline Stages:

```
Stage 0: Layers 0-7   (GPU 0)
Stage 1: Layers 8-15  (GPU 1)
Stage 2: Layers 16-23 (GPU 2)
Stage 3: Layers 24-31 (GPU 3)
```

Pipeline Execution:

```
Time Step 1: [Prefill] → [Idle] → [Idle] → [Idle]
Time Step 2: [Decode] → [Prefill] → [Idle] → [Idle]
Time Step 3: [Decode] → [Decode] → [Prefill] → [Idle]
Time Step 4: [Decode] → [Decode] → [Decode] → [Prefill]
Time Step 5: [Decode] → [Decode] → [Decode] → [Decode] (steady state)
```

Pipeline Bubbles:

- **Startup Bubbles**: Initial stages where pipeline is filling
- **Drain Bubbles**: Final stages where pipeline is draining
- **Minimization**: Use virtual pipeline parallelism to reduce bubbles

PP Configuration:

```
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 8 \
    --pp 4 \
    --dist-init-addr 172.16.4.52:20000 \
```

```
    --nnodes 2 \
    --node-rank 0
```

Data Parallelism: Request-Level Replication

Data parallelism replicates the model across multiple workers, with each worker processing different requests.

How DP Works:

1. **Model Replication**: Each DP worker has full model copy
2. **Request Distribution**: Different requests sent to different workers
3. **Independent Execution**: No communication needed (unlike training)

DP vs TP Trade-offs:

Aspect	Data Parallelism	Tensor Parallelism
Memory	Full model per worker	1/TP model per worker
Communication	None (inference)	All-reduce per layer
Latency	Lower (no sync)	Higher (sync overhead)
Throughput	Higher for small batches	Higher for large batches
Scalability	Limited by model size	Scales to very large models

DP Configuration:

```
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --dp-size 4
```

DP with Router:

For distributed DP, use router-based architecture:

```
# Worker 1
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000

# Worker 2
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000

# Router
python -m sglang_router.launch_router \
    --worker-urls http://worker1:30000 http://worker2:30000 \
    --policy cache_aware
```

Hybrid Parallelism: Combining Strategies

Real-world deployments often combine multiple parallelism strategies:

1. TP + PP (Common for Large Models):

```
Total: 32 GPUs
TP=8, PP=4

Structure:
```

```
- 4 pipeline stages
- Each stage has 8-GPU TP group
- Total: 4 stages × 8 GPUs = 32 GPUs
```

2. TP + DP (High Throughput):

```
Total: 16 GPUs
TP=4, DP=4

Structure:

- 4 data parallel replicas
- Each replica has 4-GPU TP group
- Total: 4 replicas × 4 GPUs = 16 GPUs
```

3. TP + PP + EP (MoE Models):

```
Total: 128 GPUs
TP=4, PP=2, EP=16

Structure:

- 2 pipeline stages
- Each stage: 4-GPU TP group
- 16 expert parallel groups
- Total: 2 stages × 4 TP × 16 EP = 128 GPUs
```

4. DP Attention + TP (Models with MLA):

```
Total: 32 GPUs
DP=8, TP=4 (for MLP only)

Structure:

- 8 data parallel workers for attention
- 4-GPU TP group for MLP (all DP workers form one TP group)
- Total: 8 DP workers × 4 TP = 32 GPUs
```

Communication Patterns and Optimization

Understanding communication patterns is crucial for optimizing distributed inference.

1. All-Reduce (Tensor Parallelism):

```python
# Pattern: All-reduce after each layer
# Communication: O(4 * hidden_size * tp_size) bytes per layer
# Frequency: After every attention and MLP layer

def tensor_parallel_attention(input, tp_rank, tp_size):
    # Local computation
    q, k, v = compute_qkv(input, tp_rank, tp_size)
    attn_output = attention(q, k, v)

    # All-reduce to gather full output
    attn_output = torch.distributed.all_reduce(
        attn_output, group=tp_group
```

```
    )
    return attn_output
```

2. Point-to-Point (Pipeline Parallelism):

```
# Pattern: P2P send/receive between stages
# Communication: O(hidden_size * batch_size) bytes per micro-batch
# Frequency: Once per micro-batch between stages

def pipeline_forward(input, stage_id, num_stages):
    if stage_id > 0:
        # Receive from previous stage
        input = recv_from_prev_stage()

    output = process_stage(input)

    if stage_id < num_stages - 1:
        # Send to next stage
        send_to_next_stage(output)

    return output
```

3. All-to-All (Expert Parallelism):

```
# Pattern: All-to-all for token routing
# Communication: O(num_tokens * hidden_size * ep_size) bytes
# Frequency: Once per MoE layer

def expert_parallel_moe(input, ep_rank, ep_size):
    # Route tokens to experts
    routed_tokens = all_to_all(input, ep_group)

    # Process with local experts
    expert_output = process_experts(routed_tokens)

    # Gather outputs
    output = all_to_all(expert_output, ep_group)
    return output
```

4. All-Gather (DP Attention):

```
# Pattern: All-gather before MLP in DP Attention
# Communication: O(hidden_size * max_tokens * dp_size) bytes
# Frequency: Before each MLP layer

def dp_attention_mlp(attn_output, dp_rank, dp_size):
    # All-gather attention outputs
    gathered_states = torch.distributed.all_gather(
        attn_output, group=dp_group
    )

    # MLP with tensor parallelism
    mlp_output = mlp(gathered_states)

    # Slice back to each DP worker
    return mlp_output[dp_rank * local_size:(dp_rank+1) * local_size]
```

Communication Optimization Techniques:

1. **Overlap Communication with Computation:**
   - Use CUDA streams to overlap all-reduce with next layer computation
   - Enable `--tp-comm-overlap` for automatic overlap
2. **Topology Awareness:**
   - Keep TP groups within NVLink domain
   - Use PP for inter-node communication
   - Optimize EP all-to-all for node topology
3. **Communication Backends:**
   - **NCCL**: Standard for multi-GPU communication
   - **DeepEP**: Optimized for MoE all-to-all
   - **Mooncake**: RDMA-based for high-performance transfers

Multi-Node SGLang Deployment

SGLang supports multi-node deployment using tensor parallelism (TP) and expert parallelism (EP) for large models, similar to vLLM. For smaller models, router-based architecture with replicated workers provides an alternative approach. **The key difference: while vLLM primarily uses model parallelism (TP/PP) for all multi-node scenarios, SGLang offers router-based request routing as the default approach for smaller models**, avoiding the communication overhead of model parallelism and achieving better latency for high-QPS workloads.

Multi-Node Architecture with Tensor Parallelism

For large models that don't fit on a single node, SGLang uses tensor parallelism across multiple nodes:

Components:

1. **Distributed Initialization**
   - Uses `--dist-init-addr` to specify master node address
   - `--nnodes` specifies total number of nodes
   - `--node-rank` identifies each node (0 for master, 1, 2, … for workers)
   - Uses NCCL for inter-node communication
2. **Tensor Parallelism**
   - `--tp` or `--tensor-parallel-size` splits model weights across GPUs
   - Works across nodes using NCCL for communication
   - Requires high-bandwidth inter-node interconnect (InfiniBand recommended)
3. **Expert Parallelism (for MoE models)**
   - `--ep` or `--expert-parallel-size` distributes experts across devices
   - Supports DeepEP and Mooncake backends for efficient all-to-all communication
   - Can span multiple nodes for large MoE models

Multi-Node Setup with Tensor Parallelism

Example: Multi-Node Deployment

```
# Node 0 (master node, replace 172.16.4.52:20000 with your master node IP:port)
python3 -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 2 \
    --node-rank 0
```

```
# Node 1 (worker node)
python3 -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 2 \
    --node-rank 1
```

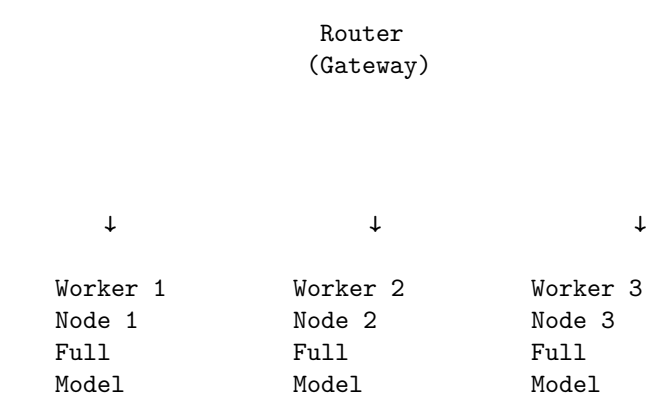
Key Parameters:

- `--dist-init-addr`: Master node IP address and port for NCCL initialization
- `--nnodes`: Total number of nodes
- `--node-rank`: Rank of this node (0 for master, 1, 2, … for workers)
- `--tp`: Tensor parallel size (total GPUs = nnodes × tp)

Router-Based Multi-Node Deployment

For smaller models, router-based architecture with replicated workers provides better latency and flexibility than model parallelism. **Unlike vLLM's model parallelism where workers must synchronize weights via all-reduce (TP) or pipeline stages (PP), SGLang's router routes requests to independent workers without weight synchronization**, eliminating inter-worker communication overhead and making it ideal for high-QPS, low-latency distributed inference scenarios.

Architecture:



1. Launch Workers on Each Node:

```
# Each node runs full model copy
# Node 1
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000

# Node 2
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000

# Node 3
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000
```

2. Configure Router (SGLang Model Gateway):

```
# Basic configuration
python -m sglang_router.launch_router \
    --worker-urls \
        http://node1:30000 \
        http://node2:30000 \
        http://node3:30000 \
        http://node4:30000 \
    --policy cache_aware \
    --port 8080

# With Inference Gateway Mode (Enterprise Extension) for multi-model deployments
python -m sglang_router.launch_router \
    --enable-igw \
    --worker-urls \
        http://node1:30000 \
        http://node2:30000 \
    --policy cache_aware \
    --port 8080

# With gRPC support
python -m sglang_router.launch_router \
    --worker-urls \
        http://node1:30000 \
        grpc://node2:50051 \
    --policy cache_aware \
    --port 8080

# With history storage (Enterprise Extension: Oracle ATP)
python -m sglang_router.launch_router \
    --worker-urls http://node1:30000 \
    --history-storage oracle_atp \
    --oracle-atp-connection-string "..." \
    --port 8080
```

3. Test Multi-Node Routing:

```
import requests

# First request creates session
response1 = requests.post(
    "http://router:8080/v1/chat/completions",
    json={
        "model": "opt-125m",
        "messages": [{"role": "user", "content": "Hello!"}],
        "session_id": "user-123"
    }
)

# Second request routes to same node (cache hit)
response2 = requests.post(
    "http://router:8080/v1/chat/completions",
    json={
```

```
        "model": "opt-125m",
        "messages": [{"role": "user", "content": "Continue"}],
        "session_id": "user-123"  # Same session ID
    }
)
```

Router Policies and Load Balancing

SGLang Router supports multiple routing policies optimized for different scenarios. **Unlike vLLM's static model parallelism where workers are fixed and requests are distributed based on batch scheduling, SGLang's router dynamically selects workers based on cache locality, load, and session affinity**, enabling intelligent request routing that adapts to workload patterns.

1. Cache-Aware Policy (Recommended):

Combines cache-aware routing with load balancing:

```
class CacheAwarePolicy:
    def select_worker(self, workers, request_text):
        # Build approximate radix tree for each worker
        # Find worker with highest prefix match
        best_match_ratio = 0
        best_worker = None

        for worker in workers:
            match_ratio = worker.approximate_tree.match_prefix(request_text)
            if match_ratio > best_match_ratio:
                best_match_ratio = match_ratio
                best_worker = worker

        # Use cache-aware if match > threshold
        if best_match_ratio > cache_threshold:
            return best_worker

        # Otherwise, use load balancing
        return self.select_least_loaded_worker(workers)
```

Features:

- **Approximate Radix Tree**: Maintains approximate prefix tree per worker
- **Dynamic Switching**: Switches between cache-aware and load balancing based on load
- **LRU Eviction**: Periodically evicts least recently used prefixes

Configuration:

```
python -m sglang_router.launch_router \
    --worker-urls http://node1:30000 http://node2:30000 \
    --policy cache_aware \
    --cache-threshold 0.5 \
    --balance-abs-threshold 10 \
    --balance-rel-threshold 1.5
```

2. Round-Robin Policy:

Simple round-robin distribution:

```
class RoundRobinPolicy:
    def __init__(self):
```

```
        self.counter = 0

    def select_worker(self, workers, request_text):
        worker_idx = self.counter % len(workers)
        self.counter += 1
        return workers[worker_idx]
```

Use Cases:

- Uniform request distribution
- No session affinity needed
- Simple load balancing

3. Shortest Queue Policy:

Routes to worker with fewest pending requests:

```
class ShortestQueuePolicy:
    def select_worker(self, workers, request_text):
        return min(workers, key=lambda w: w.pending_request_count())
```

4. Power-of-Two Choices:

Selects two random workers, chooses less loaded one:

```
class PowerOfTwoPolicy:
    def select_worker(self, workers, request_text):
        import random
        candidate1 = random.choice(workers)
        candidate2 = random.choice(workers)
        return min(candidate1, candidate2, key=lambda w: w.load())
```

Benefits:

- Better load distribution than random
- Lower overhead than checking all workers
- Good balance between performance and simplicity

5. Bucket Policy:

Groups workers into buckets based on characteristics:

```
class BucketPolicy:
    def __init__(self, num_buckets=4):
        self.buckets = [[] for _ in range(num_buckets)]

    def select_worker(self, workers, request_text):
        # Hash request to bucket
        bucket_idx = hash(request_text) % len(self.buckets)
        bucket = self.buckets[bucket_idx]

        # Select from bucket
        return self.select_from_bucket(bucket)
```

Session Affinity and Cache Locality

Session Affinity is critical for maintaining cache locality in distributed inference. **While vLLM's continuous batching optimizes throughput by batching requests together, SGLang's session affinity ensures that requests from the same conversation are routed to the same worker**, maintaining

KV cache locality and achieving 2-3x latency reduction for follow-up requests. This request-level routing strategy is particularly effective for interactive chat applications where session persistence matters.

How It Works:

1. **First Request**: Router selects worker (based on policy), creates session mapping
2. **Subsequent Requests**: Router routes to same worker using session ID
3. **Cache Hit**: Worker reuses KV cache from previous requests
4. **Latency Reduction**: 2-3x faster for follow-up requests

Implementation:

```
class SessionAwareRouter:
    def __init__(self, workers):
        self.workers = workers
        self.session_map = {}  # session_id -> worker_id

    def route_request(self, request):
        session_id = request.get("session_id")

        # Check for existing session
        if session_id and session_id in self.session_map:
            worker_id = self.session_map[session_id]
            if self.workers[worker_id].is_healthy():
                return worker_id

        # Select new worker
        worker_id = self.select_worker(request)

        # Update session mapping
        if session_id:
            self.session_map[session_id] = worker_id

        return worker_id
```

Cache Locality Benefits:

- **KV Cache Reuse**: Follow-up requests reuse cached prefixes
- **Reduced Computation**: No need to recompute shared prefixes
- **Lower Latency**: 2-3x latency reduction for conversational workloads
- **Higher Throughput**: More requests processed per second

Enterprise Extensions

The following sections cover enterprise extensions that extend SGLang's core runtime capabilities:

MCP (Model Context Protocol) Integration (Enterprise Extension)

SGLang Model Gateway includes native MCP client integration for advanced tooling and agentic workflows:

1. MCP Transport Protocols - **STDIO**: Standard input/output communication - **HTTP**: RESTful API communication - **SSE**: Server-Sent Events for streaming - **Streamable**: Custom streaming protocol

2. Tool Execution Loops - Native tool-call parsers for JSON, Pythonic, XML, and custom schemas - Streaming and non-streaming execution modes - Tool result integration with model responses - Multi-turn tool execution within conversation context

3. Reasoning Parser Integration - Built-in reasoning parsers for structured-thought models: - DeepSeek, Qwen, Llama, Mistral, GPT-OSS - Step-3, GLM4, Kimi K2, and other reasoning-capable models - Automatic

parser selection based on model type - Supports both streaming and non-streaming reasoning extraction

4. Privacy-Preserving Tool Execution - Tool execution occurs within router boundary - History and context remain local - No data leakage to external tool providers - Compliant multi-turn orchestration

Enterprise Features (Enterprise Extension)

1. Security & Authentication - Configurable authentication mechanisms - Secure communication protocols - API key management - Role-based access control (RBAC)

2. Multi-Tenant Support - Inference Gateway Mode (`--enable-igw`) for multi-model deployments - Per-model policy overrides - Isolated routing stacks per tenant - Resource quota management

3. Advanced API Endpoints - `/v1/responses`: Agentic multi-turn orchestration - `/v1/conversations`: Conversation management - `/v1/rerank`: Reranking capabilities - `/v1/embeddings`: Embedding generation - Admin endpoints for worker management

4. Dynamic Scaling - Worker lifecycle management - Automatic worker registration and removal - Kubernetes service discovery integration - Horizontal scaling based on load

Fault Tolerance and High Availability

SGLang Model Gateway provides comprehensive fault tolerance:

1. Health Checking:

Router continuously monitors worker health:

```
class HealthChecker:
    def check_worker_health(self, worker):
        try:
            response = requests.get(f"{worker.url}/health", timeout=1.0)
            return response.status_code == 200
        except:
            return False

    def background_health_check(self):
        """Continuous background health monitoring"""
        while True:
            for worker in self.workers:
                is_healthy = self.check_worker_health(worker)
                worker.update_health_status(is_healthy)
            time.sleep(self.check_interval)
```

2. Circuit Breakers:

Worker-scoped circuit breakers prevent cascading failures:

```
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.state = "closed"  # closed, open, half-open

    def call(self, func):
        if self.state == "open":
            if time.time() - self.last_failure > self.timeout:
                self.state = "half-open"
```

```
            else:
                raise CircuitBreakerOpenError()

        try:
            result = func()
            if self.state == "half-open":
                self.state = "closed"
                self.failure_count = 0
            return result
        except Exception as e:
            self.failure_count += 1
            if self.failure_count >= self.failure_threshold:
                self.state = "open"
                self.last_failure = time.time()
            raise
```

3. Automatic Failover:

When worker fails, router automatically routes around it:

```
def route_request(self, request):
    healthy_workers = [
        w for w in self.workers
        if w.is_healthy() and not w.circuit_breaker.is_open()
    ]

    if not healthy_workers:
        raise Exception("No healthy workers available")

    # Route to healthy worker with retry
    return self.select_worker_with_retry(healthy_workers, request)
```

4. Session Migration:

For failed workers, sessions can be migrated:

```
def handle_worker_failure(self, failed_worker_id):
    # Find all sessions on failed worker
    affected_sessions = [
        sid for sid, wid in self.session_map.items()
        if wid == failed_worker_id
    ]

    # Migrate sessions to healthy workers
    for session_id in affected_sessions:
        new_worker = self.select_worker(self.healthy_workers)
        self.session_map[session_id] = new_worker
        # Note: KV cache will be recomputed on new worker
        self.notify_session_migration(session_id, new_worker)
```

5. Graceful Degradation:

System continues operating with reduced capacity:

- Failed workers automatically removed from pool
- Remaining workers handle increased load
- Router automatically rebalances traffic

- Circuit breakers prevent routing to unhealthy workers
- Health checks continuously probe for recovery

Performance Considerations

Network Topology:

- **Intra-Node**: Use TP within nodes (NVLink domain)
- **Inter-Node**: Use PP or router-based for cross-node communication
- **InfiniBand**: Recommended for multi-node deployments

Communication Optimization:

- **Overlap**: Enable `--tp-comm-overlap` to hide communication latency
- **Topology Awareness**: Configure NCCL to use optimal network paths
- **Bandwidth**: Ensure sufficient bandwidth for TP all-reduce

Load Balancing Trade-offs:

- **Cache-Aware**: Best for conversational workloads with session affinity
- **Round-Robin**: Best for uniform, stateless workloads
- **Shortest Queue**: Best for dynamic, variable-length requests

Scaling Guidelines:

- **Small Models (<10B)**: Router-based with DP
- **Medium Models (10B-100B)**: TP within nodes, router across nodes
- **Large Models (100B+)**: TP+PP across multiple nodes
- **MoE Models**: TP+EP, consider DP Attention for MLA models

Expert Parallelism for MoE Models

SGLang supports Expert Parallelism (EP) for Mixture-of-Experts (MoE) models, distributing expert weights across multiple devices. This is particularly important for large-scale MoE models like Phi-tiny-MoE, Phi-tiny-MoE, and Phi-tiny-MoE.

Expert Parallelism Overview

Key Features:

- Distributes expert weights across multiple GPUs
- Optimized all-to-all communication for token routing
- Supports multiple backends: DeepEP, Mooncake, and native implementations
- Enables efficient scaling for high-performance MoE inference
- Can be combined with TP, PP, and DP for maximum scalability

How Expert Parallelism Works

MoE Architecture:

In MoE models, each layer contains multiple experts (typically 8-64 experts):

```
MoE Layer:
  Router → Selects top-K experts per token
  Experts → Process tokens in parallel
  Combine → Weighted combination of expert outputs
```

Expert Parallelism:

EP distributes experts across multiple GPUs:

```
8 Experts, EP=4:
  GPU 0: Experts 0, 1
  GPU 1: Experts 2, 3
  GPU 2: Experts 4, 5
  GPU 3: Experts 6, 7
```

Token Routing Process:

1. **Token Dispatch**: Router sends tokens to appropriate experts
2. **All-to-All Communication**: Tokens shuffled across EP ranks
3. **Expert Computation**: Each GPU processes its assigned experts
4. **All-to-All Gather**: Expert outputs gathered back
5. **Weighted Combination**: Final output computed

Communication Pattern:

```python
def expert_parallel_forward(input_tokens, ep_rank, ep_size, num_experts):
    # Step 1: Route tokens to experts
    routing_scores = router(input_tokens)  # [batch, num_experts]
    topk_experts = select_topk(routing_scores, k=2)  # Top-2 routing

    # Step 2: All-to-all to dispatch tokens
    dispatched_tokens = all_to_all(input_tokens, topk_experts, ep_group)

    # Step 3: Process with local experts
    expert_outputs = []
    for expert_id in local_experts(ep_rank, ep_size, num_experts):
        expert_tokens = filter_tokens(dispatched_tokens, expert_id)
        output = experts[expert_id](expert_tokens)
        expert_outputs.append(output)

    # Step 4: All-to-all to gather outputs
    gathered_outputs = all_to_all(expert_outputs, ep_group)

    # Step 5: Weighted combination
    final_output = weighted_combine(gathered_outputs, routing_scores)
    return final_output
```

EP Backends

All-to-All Communication Backends:

1. **`deepep`**: DeepEP library for efficient token shuffling
   - Optimized for cross-node communication
   - Supports large TopK values
   - Recommended for multi-node MoE deployments
2. **`mooncake`**: Extension of DeepEP for elastic inference
   - Leverages RDMA for high-performance transfers
   - Supports dynamic expert allocation
   - Best for large-scale deployments
3. **`none`**: Uses All-Reduce or All-Gather
   - For hybrid EP and TP setups
   - Simpler but less efficient for pure EP

MoE Computation Backends:

1. **`auto`**: Automatically selects optimal backend

- Considers hardware capabilities
- Chooses based on model characteristics

2. `triton`: Triton-based implementation
   - Good for grouped GEMMs
   - Flexible and customizable
3. `deep_gemm`: DeepGEMM backend
   - Optimized for MoE matrix multiplications
   - High performance for large experts
4. `cutlass`: CUTLASS-based backend
   - Efficient GEMMs
   - Well-optimized for NVIDIA GPUs

Expert Parallelism Setup

Example: Phi-tiny-MoE with EP

```
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --moe-a2a-backend deepep \
    --moe-runner-backend deep_gemm \
    --tp 8 \
    --ep 8
```

Example: Phi-tiny-MoE with Hybrid Parallelism

```
python -m sglang.launch_server \
    --model-path Qwen/Qwen3-235B-A22B \
    --tp 4 \
    --ep 16 \
    --pp 2 \
    --moe-a2a-backend deepep \
    --moe-runner-backend deep_gemm \
    --enable-dp-attention
```

Key Parameters:

- `--ep` or `--expert-parallel-size`: Number of GPUs for expert parallelism
- `--moe-a2a-backend`: Backend for all-to-all communication
- `--moe-runner-backend`: Backend for MoE computation
- `--enable-dp-attention`: Enable data-parallel attention (often used with EP)
- `--moe-router-topk`: Number of experts to route to (typically 2)

EP Communication Optimization

All-to-All Communication:

EP requires all-to-all communication for token routing:

```
Before A2A:
  GPU 0: [tokens for experts 0,1,2,3]
  GPU 1: [tokens for experts 4,5,6,7]

After A2A:
  GPU 0: [tokens for experts 0,1] (local)
  GPU 1: [tokens for experts 2,3] (received)
  GPU 2: [tokens for experts 4,5] (received)
  GPU 3: [tokens for experts 6,7] (received)
```

Communication Volume:

- **Per Token**: O(hidden_size) bytes
- **Per Layer**: O(batch_size * seq_len * hidden_size * ep_size) bytes
- **Frequency**: Once per MoE layer

Optimization Techniques:

1. **Token Batching**: Batch tokens by expert to reduce communication overhead
2. **Overlap**: Overlap A2A with computation using TBO or SBO
3. **Topology Awareness**: Keep EP groups within NVLink domain when possible

Computation and Communication Overlap

SGLang employs advanced overlap techniques to hide communication latency:

1. Two-Batch Overlap (TBO):

Splits requests into micro-batches, interleaving attention computation with dispatch/combine operations:

```
Batch 1: Attention → A2A → Experts → A2A → Combine
Batch 2:              Attention → A2A → Experts → A2A → Combine
```

Benefits:

- Up to 2x throughput improvement
- Hides A2A communication latency
- Better GPU utilization

Enable:

```
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --ep 8 \
    --enable-two-batch-overlap
```

2. Single-Batch Overlap (SBO):

Overlaps operations within a single batch:

```
Attention (stream 1) → A2A (stream 2) → Experts (stream 1) → A2A (stream 2)
```

Enable:

```
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --ep 8 \
    --enable-single-batch-overlap
```

Combining EP with Other Parallelism Strategies

1. EP + TP:

```
Total: 32 GPUs
EP=8, TP=4

Structure:

- 8 expert parallel groups
- Each EP group has 4-GPU TP group
- Total: 8 EP × 4 TP = 32 GPUs
```

Communication:

- EP: All-to-all for token routing
- TP: All-reduce after each layer
- Overlap: TBO can overlap both communication types

2. EP + PP:

```
Total: 64 GPUs
EP=8, PP=2

Structure:

- 2 pipeline stages
- Each stage: 8-GPU EP group
- Total: 2 PP × 8 EP = 16 GPUs per stage
```

3. EP + TP + PP:

```
Total: 128 GPUs
EP=8, TP=4, PP=2

Structure:

- 2 pipeline stages
- Each stage: 8 EP groups × 4 TP = 32 GPUs
- Total: 2 PP × 8 EP × 4 TP = 64 GPUs per stage
```

4. EP + DP Attention:

For models with MLA (Multi-Head Latent Attention):

```
Total: 32 GPUs
EP=8, DP=4 (for attention)

Structure:

- 8 expert parallel groups
- 4 data parallel workers for attention
- MLP uses EP
- Total: 8 EP × 4 DP = 32 GPUs
```

EP Performance Tuning

1. Expert Load Balancing:

MoE models can suffer from load imbalance:

```python
# Monitor expert utilization
for expert_id in range(num_experts):
    tokens_per_expert = count_tokens(expert_id)
    utilization = tokens_per_expert / expected_tokens
    print(f"Expert {expert_id}: {utilization:.2%} utilization")
```

Solutions:

- Use auxiliary loss for load balancing
- Adjust routing temperature
- Use capacity factor for token dropping

2. Communication Optimization:

```
# Use DeepEP for cross-node communication
--moe-a2a-backend deepep

# Enable overlap
--enable-two-batch-overlap

# Optimize NCCL settings
export NCCL_IB_DISABLE=0
export NCCL_IB_GID_INDEX=3
export NCCL_SOCKET_IFNAME=ib0
```

3. Batch Size Tuning:

- **Small batches**: EP overhead dominates, consider reducing EP size
- **Large batches**: EP scales well, can increase EP size
- **Optimal**: Balance between batch size and EP size

Advanced KV Cache Management

SGLang implements sophisticated KV cache management through a two-level memory pool system and radix tree-based prefix caching. This design enables efficient memory utilization and prefix reuse across requests.

Two-Level Memory Pool Architecture

SGLang uses a two-level mapping system to manage KV cache:

1. Request-to-Token Pool (`req_to_token_pool`):

Maps requests to their tokens' KV cache indices:

```
# Shape: [max_running_requests, max_context_len]
# Access: req_to_token_pool[req_pool_index][token_position] -> kv_cache_index

class ReqToTokenPool:
    def __init__(self, max_requests, max_context_len):
        self.req_to_token = torch.zeros(
            (max_requests, max_context_len),
            dtype=torch.int64
        )

    def allocate(self, req_id, num_tokens):
        """Allocate KV cache indices for request tokens"""
        for i in range(num_tokens):
            kv_index = self.get_free_slot()
            self.req_to_token[req_id][i] = kv_index
        return self.req_to_token[req_id][:num_tokens]
```

2. Token-to-KV Pool (`token_to_kv_pool`):

Maps KV cache indices to actual KV cache data:

```
# Layout: [num_layers, max_tokens, num_heads, head_dim]
# Access: token_to_kv_pool[layer_id][kv_index][head][dim]

class TokenToKVPool:
    def __init__(self, num_layers, max_tokens, num_heads, head_dim):
```

```
        self.cache_k = torch.zeros(
            (num_layers, max_tokens, num_heads, head_dim),
            dtype=torch.bfloat16
        )
        self.cache_v = torch.zeros(
            (num_layers, max_tokens, num_heads, head_dim),
            dtype=torch.bfloat16
        )

    def get_kv_cache(self, layer_id, kv_indices):
        """Retrieve KV cache for given indices"""
        return (
            self.cache_k[layer_id][kv_indices],
            self.cache_v[layer_id][kv_indices]
        )
```

3. Radix Tree Cache (`tree_cache`):

Enhances prefix reuse across requests:

```
class RadixCache:
    """Tree structure for prefix KV cache sharing."""

    def match_prefix(self, key: RadixKey) -> MatchResult:
        """
        Find longest cached prefix matching the key.

        Returns:
            - device_indices: KV cache indices for matched prefix
            - last_device_node: Terminal node of matched prefix
            - host_hit_length: Length of matched prefix
        """
        # Traverse tree to find longest matching prefix
        # Split nodes if match ends inside a stored segment
        pass

    def insert(self, key: RadixKey, kv_indices: torch.Tensor):
        """Insert new prefix into radix tree."""
        # Update tree structure
        # Link tokens via KV cache indices
        pass
```

KV Cache Lifecycle

Request Lifecycle with KV Cache:

1. **Request Arrival:**
   - Request tokenized and sent to Scheduler
   - Added to `waiting_queue`
2. **Prefix Matching:**
   - `get_new_batch_prefill` calls `match_prefix` for each request
   - Finds longest matching prefix in radix tree
   - Returns KV cache indices for matched prefix
3. **Memory Allocation:**
   - Allocates new KV cache slots for unmatched tokens

- Updates `req_to_token_pool` with new indices
- Updates `token_to_kv_pool` allocation

4. **Prefill/Extend:**
   - Computes KV cache for new tokens
   - Stores in `token_to_kv_pool` at allocated indices
   - Updates radix tree with new prefix
5. **Decode:**
   - Reuses existing KV cache from `token_to_kv_pool`
   - Appends new KV cache for generated tokens
   - Updates `req_to_token_pool` mapping
6. **Request Completion:**
   - Caches finished request prefix in radix tree
   - Frees unique suffix KV cache
   - Keeps shared prefixes for future reuse

Chunked Prefill for Long Contexts

For very long contexts that exceed available memory, SGLang uses **chunked prefill**:

How It Works:

1. **Chunking**: Long requests are split into smaller chunks
2. **Sequential Processing**: Each chunk processed sequentially
3. **Memory Management**: Older chunks can be offloaded to CPU if needed
4. **Reassembly**: KV cache reassembled for decode phase

Example:

```
def chunked_prefill(model, tokenizer, prompt, chunk_size=512):
    """Chunked prefill for long contexts"""
    chunks = [prompt[i:i+chunk_size]
              for i in range(0, len(prompt), chunk_size)]

    kv_cache = []
    for chunk in chunks:
        tokens = tokenizer(chunk).to(device)
        kv_chunk = model.forward_prefill(tokens)
        kv_cache.append(kv_chunk)

        # Optionally offload older KV pages
        if len(kv_cache) > max_gpu_chunks:
            offload_to_cpu(kv_cache[0])
            kv_cache = kv_cache[1:]

    return kv_cache
```

Benefits:

- **Memory Efficiency**: Handles contexts longer than GPU memory
- **Flexible Chunking**: Adaptive chunk sizes based on available memory
- **CPU Offloading**: Optional offloading for extreme contexts

Distributed Communication and Synchronization

Understanding communication patterns and synchronization mechanisms is crucial for optimizing distributed inference performance.

NCCL Communication Backend

SGLang uses NCCL (NVIDIA Collective Communications Library) for inter-GPU and inter-node communication.

NCCL Initialization:

```
def init_distributed_environment(
    backend="nccl",
    world_size=total_gpus,
    rank=global_rank,
    local_rank=local_gpu_id,
    distributed_init_method="tcp://master:port"
):
    """Initialize NCCL for distributed communication."""
    torch.distributed.init_process_group(
        backend=backend,
        world_size=world_size,
        rank=rank,
        init_method=distributed_init_method
    )
```

Process Groups:

SGLang creates multiple process groups for different parallelism strategies:

```
# Tensor parallel group
tp_group = torch.distributed.new_group(tp_ranks)

# Pipeline parallel group
pp_group = torch.distributed.new_group(pp_ranks)

# Expert parallel group
ep_group = torch.distributed.new_group(ep_ranks)

# Data parallel group
dp_group = torch.distributed.new_group(dp_ranks)
```

Network Topology Optimization

Intra-Node Communication (NVLink):

- **NVLink 4.0**: Up to 900 GB/s per GPU
- **Optimal for TP**: Keep TP groups within node
- **Low Latency**: Sub-microsecond latency

Inter-Node Communication (InfiniBand/Ethernet):

- **InfiniBand**: 200-400 Gb/s per port
- **Optimal for PP**: Pipeline stages across nodes
- **Higher Latency**: Microseconds to milliseconds

Topology-Aware Placement:

```
def optimize_parallelism_topology(num_gpus, num_nodes):
    """Optimize parallelism based on network topology."""
    gpus_per_node = num_gpus // num_nodes

    if num_nodes == 1:
```

```
        # Single node: Use TP
        return {"tp": num_gpus, "pp": 1, "ep": 1}
    else:
        # Multi-node: Use PP for inter-node, TP for intra-node
        return {
            "tp": gpus_per_node,  # Within node
            "pp": num_nodes,      # Across nodes
            "ep": 1
        }
```

Communication Overlap Strategies

1. Stream-Based Overlap:

Use multiple CUDA streams to overlap communication and computation:

```
# Stream 1: Computation
with torch.cuda.stream(compute_stream):
    output = layer(input)

# Stream 2: Communication (overlaps with next layer)
with torch.cuda.stream(comm_stream):
    output = torch.distributed.all_reduce(output, async_op=True)

# Next layer computation overlaps with communication
next_output = next_layer(output)
comm_stream.synchronize()  # Wait for communication to complete
```

2. Pipeline Overlap:

In pipeline parallelism, overlap stages:

```
Stage 0: [Compute Batch 1] → [Send] → [Compute Batch 2]
Stage 1: [Idle] → [Recv] → [Compute Batch 1] → [Send]
Stage 2: [Idle] → [Idle] → [Recv] → [Compute Batch 1]
```

3. Chunked Communication:

Split large communications into chunks:

```
def chunked_all_reduce(tensor, chunk_size, group):
    """All-reduce in chunks to reduce memory pressure."""
    chunks = tensor.chunk(tensor.numel() // chunk_size)
    results = []
    for chunk in chunks:
        result = torch.distributed.all_reduce(chunk, group=group)
        results.append(result)
    return torch.cat(results)
```

Distributed Checkpointing and Weight Updates

SGLang supports distributed checkpointing for efficient model loading and weight updates.

Checkpoint Engine:

```
# Install checkpoint engine
pip install 'checkpoint-engine[p2p]'

# Launch server with checkpoint engine
```

```
python -m sglang.launch_server \
    --model-path Qwen/Qwen3-8B \
    --tp 8 \
    --load-format dummy \
    --wait-for-initial-weights

# Update weights using checkpoint engine
python -m sglang.srt.checkpoint_engine.update \
    --update-method broadcast \
    --checkpoint-path /path/to/checkpoint \
    --inference-parallel-size 8
```

Update Methods:

1. **Broadcast**: Weights broadcast from loading processes to inference processes
2. **P2P**: Direct peer-to-peer weight transfer
3. **All**: Combination of broadcast and P2P

Online Weight Updates:

SGLang supports updating model weights during inference:

```
# Update weights from tensor (e.g., from training)
await engine.update_weights_from_tensor(
    named_tensors=[("layer.0.weight", tensor)],
    device_mesh=device_mesh
)
```

Distributed KV Cache Management

In router-based distributed inference, KV cache is distributed across worker nodes. Each worker maintains its own KV cache, and the router ensures session affinity to maximize cache hit rates.

Per-Worker KV Cache:

- Each worker node maintains independent KV cache
- No cross-node KV cache synchronization needed
- Session affinity ensures requests hit cached data

Cache Locality Optimization:

- Router routes requests to workers with matching session KV cache
- Minimizes cache misses and expensive recomputation
- Improves latency for conversational workloads

Session-to-Worker Mapping:

```
class DistributedKVCacheManager:
    def __init__(self, workers):
        self.workers = workers
        self.session_cache_map = {}  # session_id -> worker_id

    def get_cache_location(self, session_id):
        """Get worker that has KV cache for this session"""
        if session_id in self.session_cache_map:
            return self.session_cache_map[session_id]
        return None

    def route_request(self, request):
```

```
        session_id = request.get("session_id")
        cache_location = self.get_cache_location(session_id)

        if cache_location:
            # Route to worker with cache
            return self.workers[cache_location]
        else:
            # Route to least loaded worker
            return self.select_least_loaded_worker()
```

Speculative Decoding

Speculative Decoding is an optimization technique that accelerates inference by using a smaller "draft" model to predict multiple tokens in parallel, achieving up to K-fold speedup where K is the number of draft tokens.

Traditional Decoding

Traditional decoding generates one token at a time:

`Prefill → Decode (token 1) → Decode (token 2) → Decode (token 3) → ...`

Each decode step requires a full forward pass, making it memory-bound.

Speculative Decoding Process

Speculative decoding introduces a draft model to generate multiple tokens in parallel:

1. Draft Prefill/Extend:

- Draft model processes prefill tokens
- Typically 2-4x smaller than target model

2. Draft Decoding:

- Draft model generates N draft tokens sequentially
- Fast due to smaller model size

3. Target Verification:

- Target model verifies all N draft tokens in a single forward pass
- Three possible outcomes:

a. Full Match (Best Case):

- All N draft tokens accepted
- Returns N tokens in one round
- Achieves N-fold speedup

b. No Match (Worst Case):

- None of draft tokens accepted
- Returns only one token from target model
- Performance equivalent to traditional decoding

c. Prefix Match (Common Case):

- Some initial draft tokens accepted
- Rejected tokens' KV cache evicted
- Partial speedup based on accepted tokens

Example:

```
Draft model generates: "and it's so hot"
Target model generates: "and it's very warm"

Result: First 2 tokens ("and it's") accepted
        Remaining tokens rejected and evicted
        Speedup: 2x for this step
```

SGLang Speculative Decoding

SGLang supports speculative decoding with:

- **Configurable draft model**: Specify draft model path
- **Dynamic draft length**: Adjust based on acceptance rate
- **KV cache management**: Efficient eviction of rejected tokens
- **Grammar compatibility**: Works with constraint decoding

Configuration:

```
# Example: Use smaller Qwen model as draft for larger target model
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-7B-Instruct \
    --speculative-draft-model-path Qwen/Qwen2.5-1.5B-Instruct \
    --speculative-num-draft-tokens 4
```

Model Pairing Guidelines:

- **Target model**: Larger, more accurate model (e.g., Qwen2.5-7B-Instruct, Qwen2.5-14B-Instruct)
- **Draft model**: **Must be a different, smaller, faster model** from the same family (e.g., Qwen2.5-1.5B-Instruct, Qwen2.5-0.5B-Instruct)
- **Compatibility**: Draft and target models should share the same tokenizer and vocabulary for best results
- **Important**: The draft model must be different from the target model. Using the same model defeats the purpose of speculative decoding.
- **Self-speculative (advanced)**: As a special case, a quantized version of the target model can serve as its own draft model (e.g., INT4 quantized version as draft for FP16 target), but this still requires different model instances/weights.

Performance:

- **2-4x speedup** when acceptance rate is high
- **Minimal overhead** when acceptance rate is low
- **Memory efficient**: Draft model KV cache managed separately

Data Parallel Attention

Data Parallel Attention (DP Attention) is SGLang's optimization for models with few KV heads (e.g., models with MLA architecture), where traditional tensor parallelism leads to KV cache duplication.

The Problem: KV Cache Duplication

For models with `num_kv_heads = 1` (like models with MLA architecture):

- Traditional TP splits QKV projections across GPUs
- When `tp_size >= num_kv_heads`, KV heads are replicated
- This causes **tp_size-fold KV cache duplication**
- Wastes memory and limits batch size

Example:

```
Model with MLA architecture (num_kv_heads = 1)
TP size = 8

Traditional approach:

- Each of 8 GPUs stores full KV cache
- 8x memory waste
- Batch size limited by memory
```

DP Attention Solution

DP Attention uses data parallelism for attention components:

Architecture:

1. **Attention Layer (DP)**:
   - Each DP worker has full attention weights
   - Each worker processes different requests
   - No KV cache duplication
2. **All-Gather Before MLP**:
   - Gather attention outputs from all DP workers
   - Form complete hidden states
3. **MLP Layer (TP)**:
   - All DP workers form one TP group
   - Process MLP with tensor parallelism
4. **Slice After MLP**:
   - Slice MLP outputs back to each DP worker
   - Continue with next layer

Data Flow:

```
DP Worker 0: [Request 0, 1] → Attention (DP) → All-Gather → MLP (TP) → Slice
DP Worker 1: [Request 2, 3] → Attention (DP) → All-Gather → MLP (TP) → Slice
DP Worker 2: [Request 4, 5] → Attention (DP) → All-Gather → MLP (TP) → Slice
DP Worker 3: [Request 6, 7] → Attention (DP) → All-Gather → MLP (TP) → Slice
```

Implementation:

```python
class MLAAttentionWithDP:
    def forward(self, hidden_states, forward_batch):
        # Attention computation (data parallel)
        # Each DP worker processes its own requests
        attn_output = self.attention(hidden_states)

        # All-gather before MLP
        gathered_states, start_idx, end_idx = all_gather(
            attn_output, forward_batch, self.tp_rank, self.tp_size, self.tp_group
        )

        # MLP with tensor parallelism
        mlp_output = self.mlp(gathered_states)

        # Slice back to each DP worker
        return mlp_output[start_idx:end_idx]
```

Benefits:

- **Reduced KV Cache**: No duplication, enabling larger batch sizes
- **Higher Throughput**: Up to 1.9x decoding throughput improvement
- **Memory Efficiency**: Better GPU memory utilization

When to Use:

- Models with few KV heads (num_kv_heads <= 4)
- Large batch size scenarios
- Memory-constrained deployments
- **Not recommended** for low-latency, small-batch scenarios

Configuration:

```
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --enable-dp-attention \
    --dp-size 8 \
    --tp-size 8
```

Scheduler Evolution: From Serial to Zero-Overhead

SGLang's scheduler has undergone significant evolution to eliminate GPU idle time and maximize throughput.

Phase 1: Serial Scheduler

Initial Implementation:

```
Pre-schedule (CPU) → Compute (GPU) → Sample (GPU) → Post-schedule (CPU)
     ↓                  ↓               ↓                ↓
   Blocking          Blocking        Blocking         Blocking
```

Problems:

- GPU idle during CPU scheduling
- Scheduler overhead can be 50%+ of total time
- Poor resource utilization

Phase 2: CPU/GPU Overlap

Improved Implementation:

- CPU scheduling overlaps with GPU computation
- Pre-schedule for batch N+1 while batch N computes
- Post-schedule for batch N-1 while batch N computes

Pipeline:

```
Batch 1: Pre-schedule → Compute → Sample → Post-schedule
Batch 2:              Pre-schedule → Compute → Sample → Post-schedule
Batch 3:                           Pre-schedule → Compute → Sample
```

Implementation Details:

SGLang splits CPU responsibilities:

- **CPU-S (Scheduler CPU)**: Pre-schedule and post-schedule
- **CPU-L (Launch CPU)**: Kernel launch and result processing

Token Placeholder System:

To enable async execution, SGLang uses placeholders:

```python
class TpModelWorkerClient:
    def __init__(self):
        # Ring buffer for token placeholders
        self.future_token_ids_map = torch.zeros(
            (5 * max_running_requests,), dtype=torch.int64
        )
        self.future_token_ids_limit = 3 * max_running_requests
        self.future_token_ids_ct = 0

    def forward_batch_generation(self, batch):
        # Send batch to GPU (async)
        self.input_queue.put(batch)

        # Allocate placeholder tokens
        batch_size = len(batch.reqs)
        placeholders = []
        for i in range(batch_size):
            placeholder_idx = -(self.future_token_ids_ct + i + 1)
            placeholders.append(placeholder_idx)

        self.future_token_ids_ct = (
            (self.future_token_ids_ct + batch_size) % self.future_token_ids_limit
        )

        return placeholders  # Return immediately

    def forward_thread_func_(self):
        # Async forward pass
        while True:
            batch = self.input_queue.get()
            logits = self.model_runner.forward(batch)
            tokens = self.model_runner.sample(logits, batch)

            # Replace placeholders with actual tokens
            self.replace_placeholders(batch, tokens)
            self.output_queue.put((batch, tokens))
```

Phase 3: Multiple CUDA Streams and FutureMap

Latest Implementation:

- Multiple CUDA streams for better overlap
- FutureMap for async result handling
- Improved memory management

Benefits:

- **Zero GPU Idle Time**: GPU always has work
- **Hidden Overhead**: Scheduling overhead completely overlapped
- **Higher Throughput**: Up to 2x improvement
- **Lower Latency**: Reduced end-to-end latency

Continuous Batching

SGLang implements **continuous batching** (also called **dynamic batching**) to efficiently handle variable-length sequences and dynamic request arrival. **While vLLM's continuous batching optimizes throughput by dynamically batching requests together and processing them in parallel within model parallelism, SGLang's continuous batching is enhanced by router-based request routing and session affinity**, allowing requests to be batched while maintaining cache locality through intelligent worker selection. **The key difference: vLLM's batching focuses on maximizing GPU utilization within model parallelism, while SGLang's batching works with router-based routing to optimize both throughput and latency through cache-aware request distribution**.

Traditional Static Batching

Traditional systems use static batching:

- Batch size fixed at start
- All requests must complete before batch finishes
- Short requests wait for long ones
- Poor GPU utilization

Continuous Batching

SGLang's continuous batching:

- **Dynamic Addition**: New requests added to batch as they arrive
- **Dynamic Removal**: Completed requests removed immediately
- **Variable Batch Size**: Batch size changes over time
- **Prefill-Prioritized**: New prefill requests can interrupt decode

Implementation:

```
def get_next_batch_to_run(self):
    # Merge last batch into running batch
    if self.last_batch:
        self.running_batch.merge(self.last_batch)

    # Prioritize prefill requests
    new_batch = self.get_new_batch_prefill()

    if new_batch:
        # Process new prefill requests
        return new_batch
    else:
        # Process decode requests
        # Remove finished requests
        self.running_batch.remove_finished()

        # Retract if memory insufficient
        if memory_insufficient():
            self.running_batch.retract_decode()

        return self.running_batch
```

Benefits:

- **Better GPU Utilization**: GPU always processing useful work
- **Lower Latency**: Short requests return immediately
- **Higher Throughput**: More requests processed per second

- **Flexible**: Adapts to varying request patterns

Production Deployment Patterns

This section covers common deployment patterns for different model sizes and use cases.

Pattern 1: Small Models (<10B) - Router-Based DP

Use Case: High QPS, low latency, many concurrent sessions

Configuration:

- Model fits on single GPU
- Router-based architecture with data parallelism
- Session affinity for cache locality

Deployment:

```
# Worker nodes (each runs full model)
# Node 1
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 30000

# Node 2-4 (similar)
...

# Router
python -m sglang_router.launch_router \
    --worker-urls \
        http://node1:30000 \
        http://node2:30000 \
        http://node3:30000 \
        http://node4:30000 \
    --policy cache_aware \
    --port 8080
```

Performance:

- **Latency**: 50-100ms TTFT
- **Throughput**: 1000+ QPS
- **Cache Hit Rate**: 60-80% for conversational workloads

Pattern 2: Medium Models (10B-100B) - TP Within Nodes

Use Case: Models requiring 2-8 GPUs per node

Configuration:

- Tensor parallelism within nodes
- Router across nodes for load balancing
- Session affinity maintained

Deployment:

```
# Each node runs TP group
# Node 1 (TP=8)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
```

```
    --tp 8 \
    --port 30000

# Node 2-4 (similar)
...

# Router
python -m sglang_router.launch_router \
    --worker-urls \
        http://node1:30000 \
        http://node2:30000 \
        http://node3:30000 \
        http://node4:30000 \
    --policy cache_aware
```

Performance:

- **Latency**: 100-200ms TTFT
- **Throughput**: 200-500 QPS
- **Scalability**: Linear scaling with number of nodes

Pattern 3: Large Models (100B+) - TP+PP Across Nodes

Use Case: Very large models requiring multiple nodes

Configuration:

- Tensor parallelism within nodes
- Pipeline parallelism across nodes
- Total: TP × PP GPUs

Deployment:

```
# Node 0 (Master, PP Stage 0)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --pp 4 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 4 \
    --node-rank 0

# Node 1 (PP Stage 1)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --pp 4 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 4 \
    --node-rank 1

# Node 2-3 (PP Stages 2-3, similar)
...
```

Performance:

- **Latency**: 200-500ms TTFT (pipeline startup)

- **Throughput**: 50-200 QPS
- **Memory**: Distributed across 64+ GPUs

Pattern 4: MoE Models - EP+TP Hybrid

Use Case: Large MoE models like Phi-tiny-MoE, Phi-tiny-MoE

Configuration:

- Expert parallelism for MoE layers
- Tensor parallelism for dense layers
- Optional DP Attention for MLA models

Deployment:

```
# Phi-tiny-MoE with EP and DP Attention
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --tp 8 \
    --ep 16 \
    --moe-a2a-backend deepep \
    --moe-runner-backend deep_gemm \
    --enable-dp-attention \
    --enable-two-batch-overlap
```

Performance:

- **Latency**: 150-300ms TTFT
- **Throughput**: 100-400 QPS
- **Memory Efficiency**: Reduced KV cache duplication

Pattern 5: PD Disaggregation for Mixed Workloads

Use Case: Separating prefill-heavy and decode-heavy workloads

Configuration:

- Dedicated prefill workers for initial processing
- Dedicated decode workers for token generation
- High-performance transfer engine (Mooncake/NIXL)

Deployment:

```
# Prefill workers (compute-intensive)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode prefill \
    --port 30000 \
    --disaggregation-ib-device mlx5_roce0

# Decode workers (latency-optimized)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode decode \
    --port 30001 \
    --base-gpu-id 1 \
    --disaggregation-ib-device mlx5_roce0

# Router with PD disaggregation
```

```
python -m sglang_router.launch_router \
    --pd-disaggregation \
    --prefill http://prefill-worker:30000 \
    --decode http://decode-worker:30001 \
    --port 8080
```

Benefits:

- **Independent Scaling**: Scale prefill and decode separately
- **Better Utilization**: Optimize each worker type for its workload
- **Fault Isolation**: Failure in one type doesn't affect the other

Performance Optimization Guide

Parallelism Strategy Selection

Decision Tree:

```
Model Size?
  <10B → Router-based DP
     Single GPU or small TP (2-4)
  10B-100B → TP within nodes, Router across nodes
     TP=4-8 per node
  100B-500B → TP+PP
     TP within nodes, PP across nodes
  >500B or MoE → TP+PP+EP
      Full hybrid parallelism
```

Model Type Considerations:

- **Dense Models**: Use TP+PP
- **MoE Models**: Use EP+TP, consider DP Attention for MLA
- **Long Context**: Enable chunked prefill, consider CP
- **High QPS**: Use router-based DP

Communication Optimization

1. Topology-Aware Placement:

```
# Keep TP within NVLink domain
# Use PP for inter-node communication
# Optimize EP all-to-all for node topology

# Set NCCL environment variables
export NCCL_IB_DISABLE=0
export NCCL_IB_GID_INDEX=3
export NCCL_SOCKET_IFNAME=ib0
export NCCL_DEBUG=INFO  # For debugging
```

2. Enable Communication Overlap:

```
# Tensor parallelism overlap
--tp-comm-overlap

# Expert parallelism overlap
--enable-two-batch-overlap
--enable-single-batch-overlap
```

```
# Pipeline parallelism overlap
--num-layers-per-virtual-pipeline-stage 4
```

3. Optimize Batch Sizes:

- **Small TP**: Use larger batches to amortize communication
- **Large TP**: Use smaller batches to reduce communication overhead
- **EP**: Balance batch size with expert load

Memory Optimization

1. KV Cache Management:

```
# Enable RadixAttention for prefix reuse
--enable-radix-attention

# Configure cache size
--max-num-seqs 256
--max-model-len 8192

# Chunked prefill for long contexts
--enable-chunked-prefill
--chunk-size 512
```

2. Quantization:

```
# FP8 quantization
--quantization fp8
--fp8-format hybrid

# INT4/AWQ quantization
--quantization awq
--load-format awq
```

3. Activation Recomputation:

```
# Selective recomputation
--recompute-granularity selective
--recompute-modules attention,mlp
```

Latency Optimization

1. Prefill Optimization:

- Use chunked prefill for long contexts
- Enable RadixAttention for prefix reuse
- Optimize attention backend (FlashInfer recommended)

2. Decode Optimization:

- Enable continuous batching
- Use speculative decoding for speedup
- Optimize sampling (top-p, temperature)

3. Router Optimization:

- Use cache-aware policy for session affinity
- Minimize router latency (co-locate with workers)
- Enable health checking for fast failover

Throughput Optimization

1. Batch Size Tuning:

```
# Find optimal batch size
for batch_size in [1, 2, 4, 8, 16, 32]:
    throughput = measure_throughput(batch_size)
    print(f"Batch {batch_size}: {throughput} tokens/s")
```

2. Parallelism Scaling:

- Increase TP for larger models
- Increase DP for higher QPS
- Balance TP and DP based on workload

3. Overlap Techniques:

- Enable all overlap options
- Use multiple CUDA streams
- Pipeline CPU and GPU work

Hands-on Examples

Example 1: Basic Multi-Node SGLang Deployment

Deploy SGLang across 4 nodes with router:

```
# Node 1-4: Start workers
for i in {1..4}; do
    ssh node$i "python -m sglang.launch_server \
        --model Qwen/Qwen2.5-0.5B-Instruct \
        --port 30000"
done

# Router node: Start router
python -m sglang_router.launch_router \
    --worker-urls \
        http://node1:30000 \
        http://node2:30000 \
        http://node3:30000 \
        http://node4:30000 \
    --policy cache_aware \
    --port 8080
```

Example 2: PD Disaggregation Setup with Mooncake

Separate prefill and decode workers:

```
# Install Mooncake transfer engine
uv pip install mooncake-transfer-engine

# Prefill worker
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode prefill \
    --port 30000 \
    --disaggregation-ib-device mlx5_roce0
```

```
# Decode worker (in another terminal)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --disaggregation-mode decode \
    --port 30001 \
    --base-gpu-id 1 \
    --disaggregation-ib-device mlx5_roce0

# Router with PD disaggregation
python -m sglang_router.launch_router \
    --pd-disaggregation \
    --prefill http://127.0.0.1:30000 \
    --decode http://127.0.0.1:30001 \
    --host 0.0.0.0 \
    --port 30000
```

Example 3: Session Affinity Testing

```
import requests
import time

router_url = "http://router:8080/v1/chat/completions"
session_id = "test-session-123"

# First request (creates session)
response1 = requests.post(
    router_url,
    json={
        "model": "opt-125m",
        "messages": [{"role": "user", "content": "Hello!"}],
        "session_id": session_id
    }
)
print(f"First request latency: {response1.elapsed.total_seconds()}s")

# Second request (should hit cache)
response2 = requests.post(
    router_url,
    json={
        "model": "opt-125m",
        "messages": [{"role": "user", "content": "What did I say?"}],
        "session_id": session_id
    }
)
print(f"Second request latency: {response2.elapsed.total_seconds()}s")
print(f"Cache hit improvement: {response1.elapsed / response2.elapsed:.2f}x")
```

Example 4: Large Model with TP+PP Deployment

Deploy model across 4 nodes:

```
# Node 0 (PP Stage 0, TP ranks 0-15)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
```

```
    --tp 16 \
    --pp 4 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 4 \
    --node-rank 0 \
    --port 30000

# Node 1 (PP Stage 1, TP ranks 16-31)
python -m sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --tp 16 \
    --pp 4 \
    --dist-init-addr 172.16.4.52:20000 \
    --nnodes 4 \
    --node-rank 1 \
    --port 30000

# Node 2-3 (PP Stages 2-3, similar)
...
```

Key Points:

- Total: 4 nodes × 16 GPUs = 64 GPUs
- TP=16 within each PP stage
- PP=4 across nodes
- Requires high-bandwidth inter-node interconnect

Example 5: MoE Model with EP+TP

Deploy Phi-tiny-MoE with expert parallelism:

```
# Single command for EP+TP
python -m sglang.launch_server \
    --model-path microsoft/Phi-tiny-MoE-instruct \
    --tp 8 \
    --ep 16 \
    --moe-a2a-backend deepep \
    --moe-runner-backend deep_gemm \
    --enable-dp-attention \
    --enable-two-batch-overlap \
    --moe-router-topk 2
```

Architecture:

- 8-GPU TP groups for dense layers
- 16-GPU EP groups for MoE layers
- DP Attention for MLA attention
- Total: 8 TP × 16 EP = 128 GPUs

Example 6: Custom Router Implementation

```
import requests
from typing import List, Dict, Optional

class CustomInferenceRouter:
    def __init__(self, workers: List[str]):
```

```
        self.workers = workers
        self.session_map: Dict[str, int] = {}  # session_id -> worker_id
        self.worker_load: Dict[int, int] = {i: 0 for i in range(len(workers))}
        self.worker_cache_trees: Dict[int, RadixTree] = {
            i: RadixTree() for i in range(len(workers))
        }

    def route_request(self, request: dict) -> int:
        """Route request to optimal worker."""
        session_id = request.get("session_id")
        request_text = request.get("messages", [{}])[-1].get("content", "")

        # Check for existing session
        if session_id and session_id in self.session_map:
            worker_id = self.session_map[session_id]
            if self.is_worker_healthy(worker_id):
                return worker_id

        # Cache-aware routing
        if request_text:
            worker_id = self.select_by_cache_match(request_text)
            if worker_id is not None:
                if session_id:
                    self.session_map[session_id] = worker_id
                return worker_id

        # Fallback to load balancing
        worker_id = self.select_by_load()
        if session_id:
            self.session_map[session_id] = worker_id

        return worker_id

    def select_by_cache_match(self, request_text: str) -> Optional[int]:
        """Select worker with best cache match."""
        best_match_ratio = 0.0
        best_worker = None

        for worker_id, cache_tree in self.worker_cache_trees.items():
            match_ratio = cache_tree.match_prefix_ratio(request_text)
            if match_ratio > best_match_ratio and match_ratio > 0.5:
                best_match_ratio = match_ratio
                best_worker = worker_id

        return best_worker

    def select_by_load(self) -> int:
        """Select least loaded worker."""
        available = [
            wid for wid in range(len(self.workers))
            if self.is_worker_healthy(wid)
        ]
        if not available:
```

```
            raise Exception("No available workers")

        return min(available, key=lambda wid: self.worker_load[wid])

    def is_worker_healthy(self, worker_id: int) -> bool:
        """Check if worker is healthy."""
        try:
            response = requests.get(
                f"{self.workers[worker_id]}/health",
                timeout=1.0
            )
            return response.status_code == 200
        except:
            return False

    def update_cache_tree(self, worker_id: int, prefix: str):
        """Update cache tree after request completion."""
        self.worker_cache_trees[worker_id].insert(prefix)

    def update_load(self, worker_id: int, delta: int):
        """Update worker load after request completion."""
        self.worker_load[worker_id] += delta
```

Example 7: Monitoring and Debugging Distributed Deployment

Monitor distributed SGLang deployment:

```
import requests
import time
from collections import defaultdict

class SGLangMonitor:
    def __init__(self, router_url: str, worker_urls: List[str]):
        self.router_url = router_url
        self.worker_urls = worker_urls
        self.metrics = defaultdict(list)

    def collect_metrics(self):
        """Collect metrics from router and workers."""
        # Router metrics
        router_metrics = requests.get(f"{self.router_url}/metrics").json()

        # Worker metrics
        worker_metrics = []
        for worker_url in self.worker_urls:
            try:
                metrics = requests.get(f"{worker_url}/metrics").json()
                worker_metrics.append(metrics)
            except:
                worker_metrics.append(None)

        return {
            "router": router_metrics,
            "workers": worker_metrics,
```

```
            "timestamp": time.time()
        }

    def analyze_performance(self, metrics: dict):
        """Analyze performance metrics."""
        router = metrics["router"]
        workers = metrics["workers"]

        # Calculate aggregate throughput
        total_throughput = sum(
            w["throughput"] for w in workers if w
        )

        # Calculate average latency
        avg_latency = sum(
            w["avg_latency"] for w in workers if w
        ) / len([w for w in workers if w])

        # Cache hit rate
        cache_hit_rate = router.get("cache_hit_rate", 0)

        # Load distribution
        worker_loads = [w["pending_requests"] for w in workers if w]
        load_imbalance = max(worker_loads) - min(worker_loads) if worker_loads else 0

        return {
            "total_throughput": total_throughput,
            "avg_latency": avg_latency,
            "cache_hit_rate": cache_hit_rate,
            "load_imbalance": load_imbalance
        }

    def monitor_loop(self, interval: int = 10):
        """Continuous monitoring loop."""
        while True:
            metrics = self.collect_metrics()
            analysis = self.analyze_performance(metrics)

            print(f"Throughput: {analysis['total_throughput']:.2f} tokens/s")
            print(f"Latency: {analysis['avg_latency']:.2f} ms")
            print(f"Cache Hit Rate: {analysis['cache_hit_rate']:.2%}")
            print(f"Load Imbalance: {analysis['load_imbalance']} requests")
            print("-" * 50)

            time.sleep(interval)

# Usage
monitor = SGLangMonitor(
    router_url="http://router:8080",
    worker_urls=[
        "http://worker1:30000",
        "http://worker2:30000",
        "http://worker3:30000",
```

```
        "http://worker4:30000"
    ]
)
monitor.monitor_loop(interval=10)
```

Example 8: Fault Tolerance Testing

Test fault tolerance and failover:

```
import requests
import time
import random

def test_fault_tolerance(router_url: str, num_requests: int = 100):
    """Test router's fault tolerance."""
    session_id = f"test-session-{random.randint(1000, 9999)}"
    success_count = 0
    failover_count = 0

    for i in range(num_requests):
        try:
            response = requests.post(
                f"{router_url}/v1/chat/completions",
                json={
                    "model": "opt-125m",
                    "messages": [{"role": "user", "content": f"Request {i}"}],
                    "session_id": session_id
                },
                timeout=30.0
            )

            if response.status_code == 200:
                success_count += 1
            else:
                failover_count += 1

        except requests.exceptions.RequestException as e:
            print(f"Request {i} failed: {e}")
            failover_count += 1

        # Simulate worker failure (kill one worker)
        if i == num_requests // 2:
            print("Simulating worker failure...")
            # In real scenario, this would kill a worker process
            time.sleep(2)

    print(f"Success: {success_count}/{num_requests}")
    print(f"Failover: {failover_count}/{num_requests}")
    print(f"Success Rate: {success_count/num_requests:.2%}")
```

Example 9: Performance Benchmarking

Benchmark distributed SGLang deployment:

```
import asyncio
import aiohttp
import time
from statistics import mean, stdev

async def benchmark_router(
    router_url: str,
    num_requests: int = 1000,
    concurrency: int = 10
):
    """Benchmark router performance."""
    latencies = []
    errors = 0

    async def send_request(session, request_id):
        start_time = time.time()
        try:
            async with session.post(
                f"{router_url}/v1/chat/completions",
                json={
                    "model": "opt-125m",
                    "messages": [{"role": "user", "content": f"Test {request_id}"}],
                    "max_tokens": 100
                }
            ) as response:
                if response.status == 200:
                    latency = time.time() - start_time
                    latencies.append(latency)
                else:
                    errors += 1
        except Exception as e:
            print(f"Request {request_id} error: {e}")
            errors += 1

    async with aiohttp.ClientSession() as session:
        tasks = [
            send_request(session, i)
            for i in range(num_requests)
        ]

        # Send requests with concurrency limit
        semaphore = asyncio.Semaphore(concurrency)

        async def bounded_request(task):
            async with semaphore:
                return await task

        await asyncio.gather(*[bounded_request(task) for task in tasks])

    if latencies:
        print(f"Total Requests: {num_requests}")
        print(f"Successful: {len(latencies)}")
        print(f"Errors: {errors}")
```

```
        print(f"Mean Latency: {mean(latencies):.3f}s")
        print(f"P50 Latency: {sorted(latencies)[len(latencies)//2]:.3f}s")
        print(f"P95 Latency: {sorted(latencies)[int(len(latencies)*0.95)]:.3f}s")
        print(f"P99 Latency: {sorted(latencies)[int(len(latencies)*0.99)]:.3f}s")
        print(f"Std Dev: {stdev(latencies):.3f}s")
        print(f"Throughput: {len(latencies)/sum(latencies):.2f} req/s")

# Run benchmark
asyncio.run(benchmark_router("http://router:8080", num_requests=1000, concurrency=10))
```

Distributed Coordination and Synchronization

Effective distributed inference requires careful coordination between multiple components. This section covers synchronization mechanisms, consistency guarantees, and coordination patterns.

Request Lifecycle in Distributed System

1. Request Reception (Router):

```
class DistributedRequestHandler:
    def handle_request(self, request):
        # Parse request
        session_id = request.get("session_id")
        model_id = request.get("model")

        # Select router for model
        router = self.select_router_for_model(model_id)

        # Route request
        worker = router.select_worker(request)

        # Forward to worker
        return self.forward_to_worker(worker, request)
```

2. Worker Processing:

```
class DistributedWorker:
    def process_request(self, request):
        # Check if request needs prefill or decode
        if self.has_cache(request.session_id):
            mode = "decode"
        else:
            mode = "prefill"

        # Process based on mode
        if mode == "prefill":
            return self.process_prefill(request)
        else:
            return self.process_decode(request)
```

3. Result Aggregation:

```
class ResultAggregator:
    def aggregate_results(self, results):
        # For TP: All-reduce to gather full output
        # For PP: Collect from last pipeline stage
```

```
        # For EP: Gather expert outputs
        # For DP: Results already complete

        return self.combine_results(results)
```

Synchronization Points

1. Tensor Parallelism Synchronization:

```
# All-reduce after each layer
def tp_synchronize(output, tp_group):
    """Synchronize across tensor parallel ranks."""
    torch.distributed.all_reduce(output, op=torch.distributed.ReduceOp.SUM, group=tp_group)
    return output / torch.distributed.get_world_size(tp_group)
```

2. Pipeline Parallelism Synchronization:

```
# Point-to-point between stages
def pp_synchronize(stage_output, next_stage_rank):
    """Send output to next pipeline stage."""
    torch.distributed.send(stage_output, dst=next_stage_rank)

def pp_receive(prev_stage_rank):
    """Receive input from previous pipeline stage."""
    input_tensor = torch.empty_like(...)
    torch.distributed.recv(input_tensor, src=prev_stage_rank)
    return input_tensor
```

3. Expert Parallelism Synchronization:

```
# All-to-all for token routing
def ep_synchronize(tokens, expert_assignments, ep_group):
    """Route tokens to experts via all-to-all."""
    # Send tokens to assigned experts
    dispatched = torch.distributed.all_to_all(
        tokens, expert_assignments, group=ep_group
    )
    return dispatched
```

Consistency Guarantees

1. Session Consistency:

- **Session Affinity**: Same session always routes to same worker
- **Cache Consistency**: KV cache consistent within session
- **State Consistency**: Request state maintained across requests

2. Model Consistency:

- **Weight Synchronization**: Model weights synchronized across replicas
- **Checkpoint Consistency**: Distributed checkpoints are consistent
- **Update Consistency**: Weight updates applied atomically

3. Request Consistency:

- **Ordering**: Requests within session processed in order
- **Idempotency**: Retries don't cause duplicate processing
- **Atomicity**: Request either fully processed or not at all

Failure Handling and Recovery

1. Worker Failure Detection:

```python
class HealthChecker:
    def __init__(self, check_interval=5):
        self.check_interval = check_interval
        self.worker_health = {}

    def check_worker(self, worker_url):
        """Check if worker is healthy."""
        try:
            response = requests.get(
                f"{worker_url}/health",
                timeout=1.0
            )
            return response.status_code == 200
        except:
            return False

    def monitor_workers(self, workers):
        """Continuously monitor worker health."""
        while True:
            for worker_url in workers:
                is_healthy = self.check_worker(worker_url)
                self.worker_health[worker_url] = is_healthy
            time.sleep(self.check_interval)
```

2. Automatic Failover:

```python
class FailoverRouter:
    def route_request(self, request):
        """Route with automatic failover."""
        max_retries = 3

        for attempt in range(max_retries):
            try:
                worker = self.select_healthy_worker(request)
                response = self.forward_to_worker(worker, request)
                return response
            except WorkerFailureException:
                # Mark worker as failed
                self.mark_worker_failed(worker)
                # Retry with different worker
                continue

        raise Exception("All workers failed")
```

3. Session Migration:

```python
class SessionMigrator:
    def migrate_session(self, session_id, from_worker, to_worker):
        """Migrate session to new worker."""
        # Note: KV cache will be recomputed
        # Session state can be preserved if stored externally
```

```
        # Update session mapping
        self.session_map[session_id] = to_worker

        # Notify clients of migration (optional)
        self.notify_migration(session_id, to_worker)
```

Distributed Monitoring and Observability

1. Metrics Collection:

```
class DistributedMetrics:
    def collect_metrics(self):
        """Collect metrics from all components."""
        return {
            "router": self.get_router_metrics(),
            "workers": self.get_worker_metrics(),
            "communication": self.get_comm_metrics(),
            "cache": self.get_cache_metrics()
        }

    def get_router_metrics(self):
        return {
            "requests_per_second": self.router_rps,
            "cache_hit_rate": self.cache_hit_rate,
            "average_latency": self.avg_latency,
            "error_rate": self.error_rate
        }

    def get_worker_metrics(self):
        return {
            "throughput": self.worker_throughput,
            "gpu_utilization": self.gpu_util,
            "memory_usage": self.memory_usage,
            "pending_requests": self.pending_reqs
        }
```

2. Distributed Tracing:

```
class DistributedTracer:
    def trace_request(self, request_id):
        """Trace request across distributed system."""
        trace = {
            "request_id": request_id,
            "router_time": self.get_router_time(request_id),
            "worker_times": self.get_worker_times(request_id),
            "communication_times": self.get_comm_times(request_id),
            "total_latency": self.get_total_latency(request_id)
        }
        return trace
```

Summary

Key Takeaways

1. **RadixAttention** enables efficient KV cache reuse across requests, achieving up to 90% computation savings for shared prefixes

2. **Structured Output Decoding (X-Grammar)** provides efficient constraint decoding supporting any CFG, with 75%+ tokens validated via precompiled cache
3. **Zero-Overhead Scheduler** eliminates GPU idle time by overlapping CPU scheduling with GPU computation, achieving up to 2x throughput improvement
4. **Router-based architecture** enables distributed inference without model sharding, optimized for high-QPS, low-latency workloads. **Unlike vLLM's model parallelism (TP/PP) which requires weight synchronization between workers, SGLang's Router uses request-level routing to distribute requests across independent workers**, eliminating communication overhead and making it ideal for high-QPS, low-latency distributed inference scenarios.
5. **Session affinity** provides 2-3x latency improvement for conversational workloads through cache locality. **While vLLM's continuous batching optimizes throughput, SGLang's session affinity ensures requests from the same conversation are routed to the same worker**, maintaining KV cache locality for better latency.
6. **PD disaggregation** allows independent scaling of prefill and decode workloads, optimizing resource utilization. **Unlike vLLM's pipeline parallelism (PP) which splits model layers and requires strict synchronization, SGLang's PD disaggregation separates workload types rather than model layers**, enabling independent scaling without pipeline stage synchronization.
7. **Multi-node deployment** scales horizontally through tensor parallelism, pipeline parallelism, and router-based replication. **For smaller models, SGLang's router-based approach avoids the communication overhead of vLLM's model parallelism**, achieving better latency for high-QPS workloads.
8. **Expert Parallelism** enables efficient scaling of MoE models with optimized all-to-all communication
9. **Hybrid Parallelism** combines TP, PP, DP, and EP for maximum scalability and performance
10. **Cache locality** is critical for performance in distributed settings, requiring careful routing and session management. **SGLang's router-based request routing maintains cache locality through session affinity, while vLLM's model parallelism focuses on weight distribution**, representing fundamentally different approaches to distributed inference.

Production Deployment Best Practices

1. Network Configuration:

```
# Optimize NCCL for multi-node
export NCCL_IB_DISABLE=0
export NCCL_IB_GID_INDEX=3
export NCCL_SOCKET_IFNAME=ib0
export NCCL_DEBUG=INFO
export NCCL_P2P_DISABLE=0
export NCCL_SHM_DISABLE=0

# For InfiniBand
export NCCL_IB_HCA=mlx5
export NCCL_IB_TIMEOUT=22
export NCCL_IB_RETRY_CNT=7
```

2. Memory Management:

```
# Set appropriate memory limits
export PYTORCH_CUDA_ALLOC_CONF=max_split_size_mb:512

# Enable memory pool
```

```
--enable-memory-pool

# Configure cache size
--max-num-seqs 256
--max-model-len 8192
```

3. Performance Tuning:

```
# Enable all optimizations
--tp-comm-overlap
--enable-two-batch-overlap
--enable-radix-attention
--enable-chunked-prefill

# Optimize batch scheduling
--schedule-policy longest-prefix-match
--new-token-ratio 0.5
--new-token-ratio-decay 0.99
```

4. Monitoring and Alerting:

- Monitor throughput, latency, and error rates
- Track cache hit rates and session affinity effectiveness
- Alert on worker failures or performance degradation
- Use distributed tracing for debugging

5. Scaling Strategies:

- **Horizontal Scaling**: Add more worker nodes
- **Vertical Scaling**: Increase TP/PP/EP sizes
- **Hybrid Scaling**: Combine both approaches
- **Auto-scaling**: Scale based on load metrics

Decision Matrix: SGLang vs vLLM

The following decision matrix provides a clear guide for choosing between SGLang and vLLM based on your specific requirements:

Decision Factor	SGLang	vLLM	Notes
Primary Architecture	Request-level routing (Router-based)	Model parallelism (TP/PP/DP/EP)	SGLang routes requests; vLLM splits weights
Worker Synchronization	None (independent workers)	Required (all-reduce for TP, pipeline for PP)	SGLang eliminates communication overhead
Model Size	Single GPU to small TP (2-8 GPUs)	Large models requiring 16+ GPUs	vLLM better for very large models
QPS Requirements	High QPS (1000+)	Moderate to high QPS	SGLang optimized for high request rates
Latency Priority	Ultra-low latency critical	Throughput prioritized	SGLang: <50ms TTFT; vLLM: maximize throughput
Session Management	Session affinity & cache locality	Continuous batching	SGLang maintains session state
Cache Optimization	RadixAttention (cross-request)	PagedAttention (per-request)	Different optimization strategies

Decision Factor	SGLang	vLLM	Notes
Distributed Strategy	Router-based request routing	Model weight sharding	Fundamental architectural difference
Deployment Complexity	Router + workers	Direct model parallelism	SGLang requires router setup
Structured Output	Native X-Grammar support	Limited	SGLang excels at constrained decoding
Memory Efficiency	RadixAttention for prefix reuse	PagedAttention for variable sequences	Both optimize differently
Fault Tolerance	Router routes around failures	Requires model re-sharding	SGLang more flexible
Use Case	Interactive chat, high-QPS APIs	Batch processing, large model serving	Different workload patterns
Performance (Multi-turn)	10-20% faster than vLLM	Baseline	SGLang benefits from RadixAttention cache reuse
Performance (Single-shot)	Baseline	1.1x faster than SGLang	vLLM optimized for simple completions
Throughput (Short inputs)	5000+ tokens/sec	5000+ tokens/sec	Both achieve high throughput in offline tests
Latency Under Load	Maintains low latency	May degrade	SGLang better for real-time applications
Learning Curve	Higher (structured generation syntax)	Lower (simple APIs)	vLLM easier to get started
Community Size	Growing, responsive maintainers	Large, established	vLLM has more tutorials and examples
Documentation	Improving, focused examples	Comprehensive guides	vLLM has more extensive documentation

Quick Decision Guide

Choose SGLang if:

- Latency (especially TTFT) is critical (<50ms)
- High QPS with many concurrent sessions (1000+)
- Models fit on single GPU or small TP group (2-8 GPUs)
- Need session persistence and cache locality
- Interactive chat applications
- Need structured output decoding
- Require RadixAttention for prefix reuse

Choose vLLM if:

- Very large models requiring TP/PP across many GPUs (16+)
- Throughput-optimized workloads with large batches
- Memory-constrained environments needing PagedAttention
- Model parallelism is the primary concern
- Simple deployment without router complexity
- Batch processing workloads

Decision Flow

```
Start: Need distributed inference
```

```
Model size?
   Fits on 1-8 GPUs → Consider SGLang
   Requires 16+ GPUs → Consider vLLM

Latency requirement?
   <50ms TTFT critical → SGLang (router-based routing)
   Throughput prioritized → vLLM (model parallelism)

QPS requirement?
   1000+ QPS → SGLang (request routing optimized)
   Moderate QPS → Either (depends on other factors)

Session management needed?
   Yes (conversational apps) → SGLang (session affinity)
   No (stateless requests) → Either

Structured output needed?
    Yes → SGLang (X-Grammar)
    No → Either
```

Key Architectural Difference

vLLM's Model Parallelism (TP/PP):

- Splits model weights across GPUs
- Requires synchronization (all-reduce for TP, pipeline stages for PP)
- Communication overhead between workers
- Optimized for large models and high throughput

SGLang's Request-Level Routing:

- Routes requests to independent workers
- No weight synchronization needed
- Eliminates inter-worker communication overhead
- Optimized for high QPS and low latency

Bottom Line: If your primary concern is **high QPS and low latency** with request routing, choose **SGLang**. If your primary concern is **fitting very large models** with weight sharding, choose **vLLM**.

Complementary Approaches

Both systems can coexist in production:

- **vLLM**: For batch processing, large model serving, high-throughput workloads where model parallelism (TP/PP) is essential for fitting very large models
- **SGLang**: For interactive chat, low-latency APIs, high-QPS endpoints with session management where router-based request routing provides better latency than model parallelism

Architectural Difference: The fundamental difference is that vLLM uses model parallelism (TP/PP) to split model weights across GPUs, requiring synchronization between workers, while SGLang's Router uses request-level routing to distribute requests across independent workers without weight synchronization, making it more suitable for high-QPS, low-latency distributed inference scenarios.

Hybrid Deployment:

Many enterprises successfully use both frameworks for different parts of their infrastructure:

```
# vLLM for batch jobs and high-throughput completion
vllm serve Qwen/Qwen2.5-0.5B-Instruct \
    --tensor-parallel-size 8 \
    --port 30000

# SGLang for interactive API and structured generation
sglang.launch_server \
    --model-path Qwen/Qwen2.5-0.5B-Instruct \
    --port 8001

# Router routes based on request type
# Batch requests → vLLM
# Interactive requests → SGLang
# Structured output requests → SGLang
```

Performance Insights:

Based on benchmarks with DeepSeek-R1 on dual H100 GPUs:

- **Multi-turn conversations**: SGLang demonstrates 10-20% speed boost over vLLM due to RadixAttention's automatic caching of partial overlaps, reducing compute costs for context-heavy applications like customer support, tutoring, or coding assistants.
- **Single-shot prompts**: vLLM is 1.1x faster than SGLang for simple text completion tasks, making it better suited for templated prompts and high-throughput batch processing.
- **Throughput**: Both engines achieve over 5000 tokens per second in offline tests with short inputs. However, SGLang maintains better latency under load, making it more suitable for real-time applications.
- **Structured generation**: SGLang avoids retry loops needed by other frameworks, resulting in better end-to-end latency for structured tasks despite potentially slower raw generation speed.

Alternatives Beyond SGLang vs vLLM

While SGLang and vLLM are popular choices for LLM inference, several other frameworks offer different trade-offs:

1. **TensorRT-LLM**: NVIDIA's optimized inference engine with the best performance on NVIDIA GPUs, but limited to CUDA environments. TensorRT-LLM provides highly optimized kernels and quantization support, making it ideal for production deployments on NVIDIA hardware where maximum performance is critical.
2. **Text Generation Inference (TGI)**: Hugging Face's serving solution with good model support but generally lower throughput than vLLM. TGI is well-integrated with the Hugging Face ecosystem and provides easy deployment for Hugging Face models, making it a good choice for teams already using Hugging Face transformers.
3. **Ray Serve**: A more general-purpose serving framework that can work with multiple inference engines. Ray Serve provides flexible deployment options and can orchestrate SGLang, vLLM, or other backends, making it suitable for complex multi-model deployments and ML pipelines.

When to Consider Alternatives:

- **TensorRT-LLM**: Choose when you need maximum performance on NVIDIA GPUs and can accept CUDA-only deployment constraints.
- **TGI**: Choose when you prioritize Hugging Face ecosystem integration and ease of deployment over raw throughput.

- **Ray Serve**: Choose when you need a unified serving framework for multiple models or inference engines, or when building complex ML pipelines.

We've now covered both training (DDP, FSDP, DeepSpeed) and inference (vLLM, SGLang) systems. But understanding the theory is only part of the equation—you also need to know how to actually run these systems in practice. The next chapter provides a hands-on guide to running distributed AI training workloads using Slurm, the job scheduler used by most HPC clusters and cloud providers. We'll cover setting up Slurm clusters, submitting distributed training jobs, integrating with PyTorch DDP and FSDP, and best practices for production workloads.

References

Official Documentation

- SGLang Documentation: Official SGLang documentation and user guide
- SGLang GitHub: SGLang source code repository
- SGLang Model Gateway (Router): Router architecture and load balancing
- PD Disaggregation: Prefill-decode disaggregation feature
- Expert Parallelism: Expert parallelism for MoE models
- Multi-Node Deployment: Multi-node deployment guide

Research Papers

- SGLang: Efficient Execution of Structured Language Model Programs: Original SGLang paper introducing RadixAttention and structured output decoding
- Efficient Large Language Model Inference with Structured Outputs: Research on structured output generation in LLM inference
- Advanced Inference Optimization Techniques: Techniques for optimizing LLM inference performance
- Distributed Inference Systems for Large Language Models: Distributed inference architectures and strategies
- Memory-Efficient Inference for Large Language Models: Memory optimization techniques in LLM inference
- High-Throughput Inference Serving Systems: Systems design for high-throughput LLM serving
- SGLang Backend: Advanced Runtime Optimizations: Deep dive into SGLang's runtime optimizations and architecture (important)
- Constrained Decoding for Language Models: Techniques and algorithms for constrained text generation

Tutorials and Code Walkthroughs

- SGLang Diffusion Code Walk Through: Detailed code walkthrough of SGLang's diffusion model support by Chenyang Zhao
- SGLang Code Walk Through: Comprehensive guide to understanding SGLang's codebase and architecture
- SGLang Scheduler Evolution: Technical evolution of SGLang's scheduler from serial to CPU/GPU overlap
- Constraint Decoding in SGLang: Concepts, methods, and optimization techniques for constraint decoding
- Understanding Constraint Decoding: Comprehensive guide to constraint decoding concepts and methods

Blog Posts

- Why SGLang is a Game-Changer for LLM Workflows: Hugging Face blog post covering SGLang's architecture, RadixAttention, structured output decoding, and production use cases
- Use Cases Favoring vLLM vs SGLang in 2025: Practical deployment guide with performance benchmarks, use case analysis, and decision framework for choosing between SGLang and vLLM