SCALABLE
RELIABLE
EFFICIENT

THE SGLang PRODUCTION HANDBOOK

Architecture, Deployment, and Optimization for Large Language Model Inference

NATHAN REYNOLDS

The SGLang Production Handbook

Architecture, Deployment, and Optimization for Large Language Model Inference

Steve Publications

This book is available at https://leanpub.com/thesglangproductionhandbook

This version was published on 2026-09-08

Contents

Architecture, Deployment, and Optimization for Large Language Model Inference

This book is a complete technical guide to SGLang: a high-performance serving framework for large language models and multimodal models. It takes you from first principles through production operation, explaining the architecture, deployment patterns, optimization strategies, and operational procedures needed to run SGLang at scale. The content is based on current documentation, source code, and primary technical sources, and is intended for ML engineers, platform engineers, SREs, and infrastructure architects who need a reliable reference for designing, deploying, and maintaining SGLang-based inference platforms.

Introduction: The Inference Stack and Why SGLang Exists

Every modern generative AI system eventually arrives at the same problem: how to get tokens out of a model, fast, reliably, and at scale. The research pipeline that produces today's frontier models is well understood: collect data, train at massive scale, evaluate, iterate. But once a model is ready for deployment, the engineering challenges change. The training cluster that produced the model is often a temporary resource. The serving cluster must be permanent. It must handle unpredictable traffic. It must serve dozens of concurrent requests, each with different prompt lengths and generation budgets. It must do all of this while keeping latency low enough that users do not notice they are waiting for a matrix multiplication.

This book is about SGLang, a serving framework designed specifically for these challenges. To understand why SGLang makes the design choices it does, we first need to understand the problem space it was built to solve.

The inference serving problem

Large language model inference is not a single operation. It is a two-phase process with very different characteristics.

The first phase is prefill. The model receives the full input prompt and processes all tokens in parallel. This is compute-heavy and memory-bandwidth-bound, and it scales with the length of the prompt. A prompt with 10,000 tokens will take roughly ten times longer to prefill than a prompt with 1,000 tokens on the same hardware. During prefill, the GPU is doing what it was built to do: performing massive parallel matrix multiplications.

The second phase is decode. After prefilling, the model enters autoregressive generation. It produces one token at a time, each new token conditioned on all previous tokens. Each decode step is extremely lightweight compared to prefill: it processes only a single new query vector against all cached key-value states. This phase is memory-bound rather than compute-bound, because

the bottleneck is loading weights and cached states, not doing arithmetic. A generation of 512 tokens means 512 sequential forward passes through the same network.

This asymmetry creates a fundamental scheduling problem. If you serve one request at a time, you waste most of the GPU during decode: the same weights are loaded and reused thousands of times while the GPU sits idle most of the time. Static batching, where you wait for all requests in a batch to complete before starting the next batch, leaves GPUs underutilized because requests finish at different times.

Continuous batching is the standard solution. Instead of treating each batch as an atomic unit, the scheduler dynamically adds new requests to the batch as soon as a slot opens up, and removes finished requests immediately. This keeps the GPU fed with work and dramatically improves utilization. Frameworks like vLLM, TGI, and SGLang all implement continuous batching.

But continuous batching only solves one part of the problem. Real-world inference workloads have additional characteristics that naive serving systems handle poorly:

- Shared prefixes are common. RAG pipelines prepend the same system prompt and document context to many requests. Multi-turn conversations reuse earlier turns. Agent loops repeat tool descriptions. Recomputing the KV cache for these shared prefixes is wasteful.
- Workloads are heterogeneous. A single serving cluster may handle short chat completions, 100,000-token document analysis, structured JSON generation, and multimodal image understanding all at once. A serving system that treats all requests the same will perform poorly on some of these patterns.
- Output structure matters. Production applications need guaranteed JSON schemas, tool calls in a specific format, or responses that conform to grammatical rules. Post-generation parsing and retry loops add latency and complexity.
- Scale is real. At large organizations, inference is not a single GPU running in a lab. It is hundreds or thousands of GPUs, multiple models, multiple regions, and strict SLAs.

SGLang was designed with these realities in mind.

From research to production: why new serving stacks emerged

The first generation of LLM serving solutions were straightforward: load the model in transformers, serve it with FastAPI, hope for the best. This worked fine when models were small and traffic was light. As models grew and demand increased, teams started adding optimizations:

Early improvements included static batching, basic KV caching, and GPU memory management tweaks. These helped but did not address the fundamental scheduling problems.

Then came a second wave of purpose-built inference engines. vLLM introduced PagedAttention, treating the KV cache like operating system memory pages to eliminate fragmentation and enable efficient sharing. This was a breakthrough in memory management that immediately improved throughput and reduced wasted GPU memory.

Hugging Face's TGI focused on ecosystem integration and production readiness for models already in the Hugging Face ecosystem, providing a polished experience for teams that wanted straightforward deployment.

NVIDIA's TensorRT-LLM took a different angle: low-level kernel optimization and compilation targeting specific NVIDIA hardware architectures. This delivers peak performance but requires model-specific compilation and is tightly coupled to NVIDIA hardware.

SGLang, released in early 2024 from LMSYS and academic collaborators at UC Berkeley and Stanford, approached the problem from yet another direction. It starts from the observation that many production workloads are not single isolated completions. They are complex programs involving multiple generation calls, control flow, tool use, and structured outputs. If you design the serving system with this workload pattern in mind, you can optimize for it more aggressively than a system designed around individual requests.

This design philosophy led to three core innovations:

- RadixAttention for automatic prefix caching across all requests using a radix tree data structure.
- A frontend domain-specific language that lets developers express generation programs as first-class entities, enabling better optimization opportunities than raw API calls.

- A runtime that co-designs frontend and backend, aggressively optimizing batch scheduling and structured generation at the kernel level.

The result is a system that particularly excels at workloads with shared prefixes, structured generation needs, and complex agentic patterns. In benchmarks published by the SGLang team, SGLang achieved up to 6.4 times higher throughput and 3.7 times lower latency than competing systems on complex workloads involving agent control, reasoning, and JSON decoding. These numbers are workload-dependent and should always be validated against your own traffic patterns, but they illustrate the potential advantage when SGLang's design choices align with your use case.

What SGLang is: frontend language and inference runtime

SGLang is two things: a language and a runtime.

The frontend language provides primitives for expressing generation programs. Instead of treating each model call as a black box, you write programs that combine prompts, generation calls, branching logic, and external tool use. This looks like regular Python with special primitives that track token-level structure and dependencies. The language is expressive enough to define complex workflows: a RAG agent that retrieves documents, generates multiple responses, and then selects the best answer; a tool-calling agent that iterates between reasoning and external API calls; or a structured extraction pipeline that guarantees valid JSON output.

The runtime, officially called SRT (SGLang Runtime), is the inference engine. It takes these programs and executes them on GPUs using aggressive optimizations. The runtime is where the heavy engineering lives: continuous batching, KV cache management, RadixAttention prefix caching, CUDA graph capture, kernel fusion, speculative decoding, and support for tensor parallelism, pipeline parallelism, and expert parallelism.

The key insight is that co-designing frontend and backend unlocks optimizations that would be impossible if they were independent. When the frontend tracks token-level structure, the runtime knows exactly which prefixes are shared across which requests and can cache them efficiently. When the frontend declares structured output requirements, the runtime can use

compressed finite state machines to enforce constraints at the token level rather than relying on post-hoc validation.

In practice, you often do not need to write SGLang programs directly. The runtime exposes OpenAI-compatible APIs that work with standard client libraries. You can use the familiar chat completions interface and still benefit from all the runtime optimizations. The frontend language becomes important when you need more control: complex multi-step workflows, tight integration with application logic, or custom generation patterns that do not fit the standard API.

The SGLang design philosophy

SGLang's design reflects a few consistent principles:

The first principle is prefix reuse is king. In any realistic deployment, prompts share structure. System prompts, RAG context, conversation history, tool definitions, and even user input patterns all create opportunities to share computation. RadixAttention is the cornerstone optimization that exploits this. Instead of treating each request's KV cache independently, SGLang stores all cached states in a shared radix tree. When a new request arrives, the system walks the tree to find the longest matching prefix and reuses the cached computation. This can dramatically reduce prefill time and improve time-to-first-token.

The second principle is zero overhead in the control plane. Scheduling, batching, memory management, and prefix matching are all CPU operations that happen alongside GPU computation. If the CPU spends too much time managing these tasks, it becomes a bottleneck. SGLang's zero-overhead scheduler overlaps CPU scheduling with ongoing GPU computation, preparing the next batch while the current one executes. This keeps both CPU and GPU busy without either blocking the other.

The third principle is correctness before cleverness. Structured generation, tool calling, and constraint satisfaction should work reliably, not just work most of the time. SGLang bakes structured output support into the decoding loop itself. When you request JSON-constrained generation, the model literally cannot produce invalid tokens: the vocabulary mask is enforced at each step by a compiled grammar. This eliminates retry loops and parsing errors at the source.

The fourth principle is progressive complexity. You should be able to start with a single GPU and a familiar API, then scale up gradually as your needs grow. SGLang supports everything from a developer's laptop to multi-node clusters with thousands of GPUs. The same runtime that handles your local experiment can scale to production without requiring a complete re-architecture.

Where SGLang fits in the ecosystem

SGLang competes with other inference serving frameworks, but competition in this space is productive rather than zero-sum. Different frameworks excel at different things, and the right choice depends on your specific requirements:

- If your primary concern is raw throughput on a single model running on NVIDIA hardware and you want maximum optimization, TensorRT-LLM is a strong candidate. It requires model-specific compilation but can extract more performance from specific hardware than a general-purpose framework.
- If you want broad model support with a fast path to production and strong community backing, vLLM remains the safe default. It has excellent documentation, wide adoption, and continuous improvement across many model architectures.
- If you are already deeply integrated with Hugging Face and want tight ecosystem compatibility, TGI provides a polished experience specifically for that stack.
- If your workload involves significant prefix reuse, structured generation, complex agentic patterns, or multimodal inference, SGLang is particularly well-suited. RadixAttention shines when requests share prefixes. The structured generation capabilities reduce engineering overhead when you need guaranteed output formats. The frontend language provides better control over complex workflows.

It is also worth noting that many production systems use multiple frameworks. Different models might run on different backends depending on their characteristics. The SGLang Model Gateway can route traffic across different workers, even to different inference frameworks that expose OpenAI-compatible APIs. This allows you to mix and match tools in the same deployment.

How to read this book

This book is organized to take you from understanding SGLang to running it in production. The first section establishes the foundation: what SGLang is, how it works internally, and how requests flow through the system. If you are new to SGLang, start here. If you already understand the basics, you can skim these chapters.

The second section covers installation, configuration, and deployment. It walks through getting SGLang running across different environments: single GPU, multi-GPU, multi-node clusters, Docker, Kubernetes, and different hardware platforms. These chapters contain complete, copy-paste-ready configurations rather than isolated fragments.

The third section dives into SGLang's capabilities: model support, APIs, structured generation, tool calling, speculative decoding, LoRA serving, and other advanced features. These chapters explain not only how to use each feature but also the trade-offs and limitations involved.

The fourth section focuses on scaling: parallelism strategies, prefill-decode disaggregation, expert parallelism for MoE models, and the SGLang Model Gateway for large-scale routing. These are the tools you need when you move beyond a single machine.

The fifth section is about optimization and operations: benchmarking methodology, performance tuning, capacity planning, monitoring, security, troubleshooting, and lifecycle management. These chapters treat optimization as a systematic process rather than a collection of magic flags, and they provide practical guidance for running SGLang in real production environments.

The book concludes with a comprehensive configuration reference and troubleshooting guide organized by symptom so you can quickly find solutions to problems.

SGLang is actively evolving. The content in this book reflects the state of the project as of mid-2026, with specific references to features and configurations from version 0.5 and earlier. Always verify commands and parameters against the current documentation for your specific version, especially as SGLang continues to add capabilities and refine its API surface.

Chapter 1: What is SGLang

This chapter provides a focused introduction to SGLang itself: what the project is, how it originated, what specific problems it was designed to solve, and how it differs from alternative approaches. By the end of this chapter, you will understand the core value proposition of SGLang and the workloads for which it is particularly suited.

The inference serving problem

Large language model inference serves as the backbone of modern AI applications, but it is fundamentally difficult. A single model deployment must handle an unpredictable mix of workloads: short interactive queries from thousands of users, long context analysis for document processing, structured generation for data extraction, and tool-augmented reasoning for agent systems. Each of these patterns stresses the system differently, and optimizing for one can degrade performance for another.

The fundamental asymmetry of transformer inference creates this challenge. During the prefill phase, the model processes the entire input prompt in parallel, performing heavy matrix multiplications that fully utilize GPU compute. During the decode phase, the model generates one token at a time in a tight autoregressive loop. Each decode step performs only a fraction of the computation of prefill but repeats thousands of times. This makes decode memory-bandwidth-bound rather than compute-bound, leaving GPU tensor cores underutilized.

Early serving approaches treated each request independently or used static batching, which led to poor GPU utilization. When requests complete at different times, statically batched workloads leave GPUs idle waiting for the slowest request in the batch. When short requests arrive while long requests are still decoding, those short requests queue unnecessarily.

Continuous batching solved the basic utilization problem by dynamically adding and removing requests from batches at each step. Every modern inference engine uses it. But continuous batching alone does not address the

full complexity of production workloads. Real systems need to handle shared computation across requests, enforce output constraints, manage many different models, scale across clusters, and provide reliable observability.

From research to production: why new serving stacks emerged

The inference serving ecosystem evolved rapidly as LLMs moved from research curiosities to production systems. Each generation of tools addressed limitations of the previous one:

The earliest deployments used raw transformers libraries with basic HTTP wrappers. These worked for small models and low traffic but did not scale. Manual optimization added features like basic KV caching and static batching, but these were bolted on rather than integrated into the core design.

The next wave brought purpose-built inference engines with architectural innovations. vLLM's PagedAttention treated the KV cache as paged memory, eliminating fragmentation and enabling efficient sharing across variable-length sequences. This single innovation dramatically improved throughput and memory efficiency, making it possible to serve larger batches and longer contexts.

Hugging Face's TGI focused on production polish and ecosystem integration. For teams already using Hugging Face models and tooling, TGI provided a straightforward deployment path with good operational tooling.

NVIDIA's TensorRT-LLM took a compilation approach, optimizing models at the kernel level for specific hardware targets. This delivered peak performance on NVIDIA GPUs but required model-specific compilation and locked deployments tightly to NVIDIA hardware.

SGLang emerged from this landscape with a different starting assumption. Most of these systems were designed around individual inference requests: here is a prompt, return a completion. SGLang's creators observed that production AI applications are rarely this simple. They involve multi-step workflows, conditional logic, repeated calls with shared context, tool interactions, and structured output requirements.

This observation led to a co-design approach: build a frontend language that can express these complex programs, and build a runtime optimized

to execute them efficiently. The frontend tracks token-level structure and dependencies. The runtime uses this information to share computation, optimize batching, and enforce constraints at the kernel level.

What SGLang is: frontend language and inference runtime

SGLang is both a programming language and an inference runtime. Understanding this dual nature is essential to understanding what makes SGLang distinctive.

The frontend language, structured generation language, provides primitives for writing programs that interact with language models. Instead of treating each model call as an opaque API request, you write programs that combine prompts, generation calls, control flow, and external computation. The language is Python-based, using decorators and special primitives that make model interactions explicit and trackable.

A simple SGLang program might look like this:

```
import sglang as sgl

@sgl.function
def generate_with_system_prompt(s, prompt):
    s += "You are a helpful assistant.\n"
    s += f"User: {prompt}\n"
    s += "Assistant: "
    s += sgl.gen("answer", max_tokens=100)
```

This looks like regular Python with a twist: the `s` object accumulates tokens, and `sgl.gen` marks where generation happens. But this apparent simplicity hides what is important: the runtime tracks every token added to `s`, knows which parts are static prompts and which parts are generated, and uses this information to optimize execution across multiple invocations. If you call `generate_with_system_prompt` multiple times with different prompts, the runtime recognizes that the system prompt prefix is shared and caches its KV state once for reuse.

The runtime, SRT, is a high-performance inference engine built to execute these programs efficiently. It manages GPUs, schedules batches, handles KV

caches, supports distributed execution, and exposes APIs for applications to interact with models. Under the hood, it implements all the optimizations we expect from a modern inference engine: continuous batching, memory-efficient attention, CUDA graph capture, kernel fusion, and support for tensor parallelism across multiple GPUs.

What distinguishes SRT from other runtimes is that its optimizations are designed with the frontend language in mind. The token tracking in the frontend feeds directly into RadixAttention's prefix caching. The structured generation declarations in the frontend enable optimized grammar-based constraint enforcement in the decoder. The runtime and frontend are not independent components glued together: they are designed to work as a single system.

In practice, you do not always need to write SGLang programs. The runtime exposes standard APIs, including full OpenAI-compatible endpoints for chat completions and text completions. You can deploy SGLang as a drop-in replacement for other inference servers and benefit from all the runtime optimizations without changing your application code. The frontend language becomes valuable when you need the advanced capabilities it enables: complex multi-step workflows, fine-grained control over generation, or tight integration between application logic and model calls.

The SGLang design philosophy

SGLang's design choices reflect a consistent philosophy about how inference serving should work in production. Four principles are particularly important:

The first principle is prefix reuse is king. In realistic production workloads, prompts share structure. System prompts are repeated across thousands of requests. RAG pipelines prepend the same document context to multiple queries. Conversation histories create progressively longer shared prefixes as turns accumulate. Agent loops repeat tool definitions and instruction templates. Every time you recompute the KV cache for a shared prefix, you waste compute and increase latency.

SGLang addresses this with RadixAttention, which stores KV cache states in a radix tree data structure that enables efficient prefix matching and reuse. When a request arrives, the system walks the tree to find the longest matching prefix and reuses the cached computation instead of re-running prefill. For

workloads with significant prefix overlap, this can reduce time-to-first-token dramatically and improve throughput by avoiding redundant computation.

The second principle is zero overhead in the control plane. Modern inference engines have complex control planes: schedulers that form batches, memory managers that allocate KV cache, prefix matchers that find shared computation, and distributed coordinators that manage communication across GPUs. All of this work happens on CPU, and if the CPU becomes a bottleneck, GPU utilization suffers regardless of how well-optimized the kernels are.

SGLang's zero-overhead scheduler addresses this by overlapping CPU scheduling with GPU computation. While the GPU is processing the current batch, the CPU prepares the next batch: it matches prefixes, allocates memory, forms the batch structure, and updates token maps. By the time the GPU finishes the current step, the next batch is already ready to go. This keeps both CPU and GPU busy without either blocking the other.

The third principle is correctness before cleverness. When your application depends on structured outputs, retry loops and post-processing are expensive and fragile. If you need valid JSON for a tool call, having the model sometimes produce malformed output that you must retry or fix is unacceptable at scale.

SGLang bakes structured generation into the decoding loop. When you declare a JSON schema or grammar constraint, the runtime compiles it into a finite state machine that tracks valid tokens at each position. During decoding, the vocabulary mask is computed dynamically based on the grammar state, and the model literally cannot produce invalid tokens. This guarantees correctness without sacrificing too much performance: SGLang's compressed finite state machine implementation can make constrained decoding faster than unconstrained decoding in some cases by allowing multi-token decoding when the grammar makes the next tokens deterministic.

The fourth principle is progressive complexity. A good serving system should work well for a single developer running an experiment on a laptop and for a large organization running thousands of GPUs in production. The complexity should scale with your needs, not be mandatory from day one.

SGLang supports this by making advanced features opt-in. A basic deployment is a single command that launches a server on one GPU. When you need more capacity, you add tensor parallelism with a flag. When you need more throughput, you add data parallelism with more replicas. When you need to scale across nodes or disaggregate prefill from decode, the architecture sup-

ports it without requiring a fundamental re-architecture. The same runtime handles all these configurations.

Where SGLang fits in the ecosystem

SGLang is one option in a crowded ecosystem of inference serving frameworks. Understanding where it fits relative to alternatives helps you decide when it is the right tool:

SGLang versus vLLM. These are the two most actively developed open-source inference engines. vLLM pioneered PagedAttention and has been the default choice for many teams due to its broad model support and strong community. SGLang's differentiator is RadixAttention for automatic prefix caching and tighter integration of structured generation into the runtime. On workloads with significant prefix reuse, SGLang can outperform vLLM in both throughput and latency. On workloads with unique prompts and no structured output requirements, the performance gap narrows. If you need the safest bet with the broadest community support, vLLM is still a strong default. If your workload pattern aligns with SGLang's strengths, SGLang can provide meaningful advantages.

SGLang versus TGI. TGI focuses on Hugging Face ecosystem integration and production polish for teams already using that stack. SGLang provides broader low-level control and optimization, particularly for complex workloads. If your team is deeply invested in Hugging Face tooling and has straightforward serving needs, TGI may be simpler. If you need the performance characteristics SGLang provides, it is worth the additional engineering investment.

SGLang versus TensorRT-LLM. TensorRT-LLM is optimized specifically for NVIDIA hardware with aggressive kernel-level compilation. For single-model deployments on NVIDIA GPUs where throughput is critical and you want maximum hardware utilization, TensorRT-LLM can extract more performance than a general-purpose framework. The trade-offs are flexibility: TensorRT-LLM requires model-specific compilation, supports fewer model architectures, and is tightly coupled to NVIDIA hardware. SGLang trades some absolute peak performance for flexibility, broader model support, multi-vendor hardware support, and a richer feature set.

SGLang versus llama.cpp and Ollama. These targets are different. llama.cpp and Ollama excel at local, CPU-friendly, resource-constrained inference.

SGLang targets GPU-accelerated server deployments at scale. If you need to run models on edge devices, laptops, or CPUs, look at llama.cpp. If you need to serve models in a data center or cloud GPU environment, SGLang is in the right category.

It is also worth noting that production systems often use multiple frameworks. The SGLang Model Gateway can route traffic to different backends, even different inference frameworks, as long as they expose OpenAI-compatible APIs. This allows you to deploy the right tool for each model without locking into a single framework for your entire platform.

What to expect from the rest of this book

This book treats SGLang as a production system, not a research prototype. Every chapter is oriented toward practical engineering: how to deploy, configure, optimize, monitor, and operate SGLang in real environments. The subsequent chapters build on each other:

Chapters 2 through 3 explain SGLang's internal architecture and the frontend language, giving you the mental model needed to understand how requests are processed and how the system's optimizations work.

Chapters 4 through 6 cover installation, configuration, and model support, taking you from a working installation to serving specific models in different environments.

Chapters 7 through 14 dive into SGLang's core mechanisms and advanced features: scheduling, KV cache management, attention backends, quantization, parallelism, disaggregation, and speculative decoding.

Chapters 15 through 17 cover APIs, structured generation, and the Model Gateway for large-scale routing.

Chapters 18 through 24 focus on deployment patterns, performance engineering, capacity planning, observability, security, operations, and troubleshooting.

Chapter 25 provides a comprehensive configuration reference for quick lookup.

The goal is that after reading this book, you can design, deploy, and operate a serious SGLang-based inference platform without needing to consult a dozen

disconnected tutorials and blog posts. You will understand not only how to use SGLang's features but also why they exist, when they help, when they hurt, and how to troubleshoot when things go wrong.

Chapter 2: Architecture Overview

Understanding SGLang's architecture is essential for using it effectively. This chapter provides the mental model you need: the major components, how they interact, and what happens from the moment a request arrives at the server until tokens are generated and returned. We will cover the high-level structure first, then trace a request through the system, then explain the key design decisions that distinguish SGLang from other inference engines.

High-level component diagram

SGLang uses a three-layer architecture: a frontend language, an API server layer, and the SGLang Runtime. Each layer has a distinct responsibility, and the boundaries between layers are designed for both modularity and performance.

At the top is the frontend language. This is the programming interface for writing generation programs: Python code that uses SGLang primitives to combine prompts, generation calls, control flow, and structured output constraints. The frontend is optional for basic usage. You can interact with SGLang through standard APIs without writing any SGLang programs. But for complex workflows, the frontend provides the expressiveness and control that direct API calls cannot match.

Below the frontend is the API server layer. This layer provides HTTP and gRPC interfaces for applications to interact with models. It includes OpenAI-compatible endpoints for chat completions and text completions, native SGLang endpoints for SGLang-specific features, and specialized endpoints for structured generation, tool calling, tokenization, and administration. The API layer handles request parsing, authentication, streaming responses, and error handling. It communicates with the runtime using high-performance inter-process communication.

At the bottom is the SGLang Runtime, where the heavy engineering lives. The runtime coordinates model execution, schedules batches, manages GPU memory, handles KV caches, and supports distributed execution across multiple GPUs and nodes. It is organized into several key components:

The engine acts as the central coordinator within the runtime. It manages worker processes, handles tokenization, coordinates model execution, and provides the interface between the API layer and the compute layer.

The scheduler is one of the most important components. It implements continuous batching, decides which requests to include in each batch, handles prefix matching with RadixAttention, manages memory allocation, and ensures GPU utilization stays high while meeting latency targets.

The model runner executes the actual model forward passes on GPU. It handles CUDA graph capture, kernel fusion, and coordination with the attention backend. The model runner is where the heavy computation happens.

The memory management system handles KV cache storage and allocation. It implements RadixAttention for prefix caching, manages memory pools for request tracking and KV storage, and handles cache eviction and garbage collection.

The communication layer manages data movement within a single machine and across nodes in a distributed deployment. It uses ZeroMQ for intra-process communication within a single machine and NCCL or specialized backends like Mooncake for distributed communication across nodes.

This layered architecture enables progressive complexity. A simple single-GPU deployment uses only a subset of these components. As you scale up to multi-GPU, multi-node, and disaggregated deployments, additional components become active without changing the fundamental architecture.

The request lifecycle end-to-end

Tracing a single request through SGLang provides a concrete understanding of how the pieces fit together. Consider a typical chat completion request arriving at a running SGLang server. We will follow it from arrival to response.

The request enters through the API layer. A client sends an HTTP POST to the `/v1/chat/completions` endpoint with a JSON body containing the model name, messages, sampling parameters, and any additional options. The API server validates the request: it checks that the model is loaded, the parameters are valid, authentication succeeds if required, and the request does not exceed configured limits.

The API server tokenizes the prompt. Tokenization happens in a dedicated tokenizer process rather than in the main request handling thread. This design choice keeps tokenization work from blocking other requests and from competing with the scheduler for CPU resources. The tokenizer converts the prompt text into token IDs, applies the chat template if one is configured, and returns the token IDs along with the message structure to the engine.

The engine registers the request with the scheduler. The scheduler now owns the request's lifecycle. It adds the request to a waiting queue and will decide when to include it in a batch based on available resources, priority, and cache efficiency.

The scheduler performs prefix matching with RadixAttention. Before the request enters any batch, the scheduler walks the radix tree to find the longest prefix that matches already-cached tokens. This step is critical: it determines how much computation can be reused. If 80 percent of the prompt tokens match a cached prefix, the scheduler knows that only 20 percent of prefill computation is needed. The cache hit rate is exposed as a metric and is one of the key performance indicators for SGLang deployments.

The scheduler forms the batch. SGLang uses continuous batching: at each decode step, the scheduler decides which requests to include. It considers several factors: how much KV cache memory is available, whether including this request would cause OOM, whether the request has a high cache hit rate, and whether prefill or decode work should be prioritized. The scheduler also implements chunked prefill: if a request has a very long prompt, it may be split into chunks so that decode work from other requests can continue without being blocked.

The scheduler allocates memory for the batch. Using the KV cache memory pool, the scheduler allocates space for tokens that need to be cached. Tokens that matched the radix tree reuse existing cache entries. New tokens get newly allocated cache slots. The memory pool uses paged allocation to minimize fragmentation, similar to how operating systems manage memory.

The model runner executes the batch. The batch of token sequences is sent to the GPU, and the model forward pass runs. During prefill, this is a heavy compute operation processing all prompt tokens in parallel. During decode, this is a lightweight operation processing one new token per active request. The model runner uses CUDA graphs for decode to minimize kernel launch overhead, and it may use specialized attention backends like FlashInfer

or FlashAttention for the attention computation.

During generation, the model produces log probabilities for the next token. A sampling step selects the actual next token based on the configured sampling parameters: temperature, top-p, top-k, or more advanced sampling strategies. Structured generation constraints, if active, are enforced at this step by masking invalid tokens based on the grammar state.

New tokens are cached in the KV store. After each generation step, the newly computed key-value states for the new tokens are stored in the KV cache. These will be available for reuse by future requests that share prefixes, and they will be used by this request in subsequent decode steps.

The scheduler updates state and prepares the next step. The request's position advances, memory usage is updated, and the scheduler decides what happens next. If the request has reached its maximum token count or generated a stop sequence, it is marked complete and its cache entries become candidates for eviction. Otherwise, it remains in the running batch for the next step.

Tokens are detokenized and streamed to the client. A dedicated detokenizer process converts token IDs back to text. For streaming responses, tokens are sent to the client as soon as they are generated using Server-Sent Events. This provides low latency for interactive applications. For non-streaming responses, tokens are accumulated and sent as a complete response when generation finishes.

When the request completes, resources are cleaned up. The scheduler releases the request from the running batch. The KV cache entries remain in the radix tree as candidates for reuse by future requests, subject to the eviction policy. If the cache is under memory pressure, LRU eviction removes the least recently used entries first.

This lifecycle illustrates several key design decisions. Tokenization, detokenization, and model execution happen in separate processes to avoid blocking each other. The scheduler runs continuously on CPU while model execution runs on GPU, with overlap to minimize idle time. The KV cache is shared across all requests and reused via RadixAttention rather than being per-request. The architecture is designed from the ground up for concurrent batched execution rather than sequential single-request processing.

Frontend to runtime boundary

The boundary between SGLang's frontend language and its runtime is important for understanding how SGLang achieves its performance advantages over systems that treat the frontend as a simple API wrapper.

When you write a SGLang program using the frontend language, you are not just composing API calls. You are constructing a computation graph that tracks token-level structure and dependencies. The frontend represents prompts, generation calls, and control flow as first-class objects that maintain information about which tokens are static, which are generated, and how they relate to each other.

This tracking enables optimizations that would be impossible with opaque API calls. When the frontend program is executed, the runtime receives not just a prompt string but structured information about the prompt's components. It knows that certain tokens are system prompts repeated across many invocations, certain tokens are user-specific data, and certain positions are where generation will happen. This information feeds directly into RadixAttention's prefix caching: the runtime can build a radix tree that reflects the actual program structure rather than treating each full prompt as an atomic unit.

The frontend also enables intra-program optimizations. A SGLang program with multiple generation calls can be optimized so that the KV cache from earlier generation steps is reused in later steps without recomputation. Control flow branches can be handled efficiently because the runtime tracks which tokens belong to which branches. This is particularly valuable for complex agent workflows that involve multiple model calls with shared context.

For applications that use the standard OpenAI-compatible APIs rather than the frontend language, the runtime still benefits from prefix caching and batch scheduling, but it cannot exploit the deeper structural information that SGLang programs provide. The APIs treat each request as a standalone prompt, and prefix caching works only on the raw prompt content. For many workloads this is sufficient. For workloads with complex structure and repeated patterns, the frontend language can provide additional optimization opportunities.

GPU execution model overview

SGLang's GPU execution model is designed to maximize utilization while minimizing overhead. Several mechanisms work together:

CUDA graph capture is used for the decode phase to eliminate kernel launch overhead. Instead of launching kernels individually for each decode step, the decode loop is captured into a CUDA graph that is replayed each step. This reduces CPU-to-GPU synchronization and minimizes the overhead of kernel launches, which can be significant when the actual computation per step is small. CUDA graphs are captured at startup for different batch sizes, and the runtime selects the appropriate graph at runtime based on the actual batch size.

Attention backends provide optimized implementations of the attention mechanism, which is the computationally dominant operation in transformer inference. SGLang supports multiple backends: FlashInfer, FlashAttention 3 and 4, Triton, TRT-LLM, and others. Each backend has different characteristics: some are optimized for prefill, some for decode, some for specific hardware architectures, and some for specific attention patterns like grouped query attention or multi-head latent attention. The runtime selects or allows configuration of the backend based on hardware, model architecture, and workload characteristics.

Kernel fusion combines multiple operations into single kernel launches to reduce memory transfers between HBM and compute units. For example, instead of computing attention, then layer normalization, then feed-forward in separate kernels, fused kernels can execute these operations in sequence within a single kernel launch. This reduces the memory bandwidth pressure that often limits decode performance.

The model runner coordinates these mechanisms, dispatching work to the GPU in a way that maximizes throughput while respecting memory constraints. It handles batch formation on the GPU side, selects the appropriate attention backend and CUDA graph, and ensures that kernel launches overlap with data movement where possible.

Single-node vs. multi-node topology

SGLang scales from a single GPU to multi-node clusters. The topology changes with scale but the core architecture remains consistent.

On a single GPU, the architecture is straightforward: one API server process, one tokenizer process, one scheduler process, and one model runner process all on the same machine, communicating through shared memory and ZeroMQ.

On a multi-GPU single node, tensor parallelism splits model weights across GPUs. Each GPU holds a shard of the weight matrices and processes the same input tokens in parallel, with communication between GPUs after each layer to combine results. The scheduler runs on one GPU and coordinates across all GPUs in the tensor parallel group. Memory management and KV caching are also distributed: each GPU manages its portion of the KV cache.

On multi-node deployments, tensor parallelism can span nodes, with NCCL or specialized backends handling cross-node communication. Data parallelism runs multiple independent replicas of the full model on different GPU sets, increasing throughput by handling more concurrent requests. Expert parallelism distributes individual experts in MoE models across GPUs to reduce memory pressure and improve throughput for large sparse models.

The communication layer becomes critical at multi-node scale. Network latency and bandwidth directly impact performance, especially for tensor parallelism where all-reduce operations after each layer require fast interconnects. NVLink within nodes and InfiniBand or RoCE between nodes are standard for production deployments. SGLang's PD disaggregation uses specialized transfer backends like Mooncake or NIXL for efficient KV cache transfer between prefill and decode workers across nodes.

Understanding these topologies is important for deployment planning. A single-node multi-GPU deployment is relatively straightforward to set up and debug. Multi-node deployments introduce complexity around networking, fault tolerance, and configuration synchronization. The choice depends on your model size, throughput requirements, and infrastructure capabilities.

Chapter 3: The SGLang Frontend Language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

SGLang as a generation programming language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Select, generate, and control flow constructs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Structured generation with JSON and schema constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Tool calling and function calling syntax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Compiling SGLang programs to the runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 4: Installation and Environment Preparation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

System requirements (OS, Python, CUDA/ROCm, GPU memory)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

pip installation and quick start

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Source installation and build options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Docker and containerized deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Accelerator environments (NVIDIA CUDA, AMD ROCm)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model retrieval and Hugging Face integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Dependency management and version compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Common installation failures and fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 5: Launching and Configuring the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

The launch command and its arguments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model and tokenizer configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

GPU selection and memory limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Networking, ports, and host binding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Logging, debugging, and profiling flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Production-ready launch examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 6: Model Support and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Supported model families and architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Transformer models and decoder-only serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multimodal and vision-language models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Mixture-of-experts and MoE-specific considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Long-context and specialized architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

LoRA adapters and dynamic adapter loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Tokenizer and chat-template gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Bringing new models into SGLang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 7: The Scheduler and Request Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Request ingestion and queuing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Prefill and decode phases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Continuous batching mechanics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chunked prefill and memory pressure management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Scheduling policies and fairness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Timeout, eviction, and cancellation

Chapter 8: KV-Cache and RadixAttention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Why KV-caching matters for inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Paged attention and memory pools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

RadixAttention and prefix tree structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

How prefix caching reduces redundant computation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Cache eviction and replacement policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Cache hit rate optimization strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Hierarchical and distributed caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 9: Attention Backends and Kernels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

The role of attention in inference performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Available attention backends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Backend selection and hardware compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

CUDA kernels and kernel fusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Diagnosing and optimizing backend performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

CUDA graphs and execution optimization

Chapter 10: Quantization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Why quantize: accuracy vs. memory vs. throughput

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Supported formats (FP8, INT4, AWQ, GPTQ, etc.)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model preparation for quantization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

KV-cache quantization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Runtime configuration for quantized models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Validating quantized model quality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Hardware-specific quantization considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 11: Parallelism Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Data parallelism for throughput

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Tensor parallelism for large models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Pipeline parallelism and multi-stage execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Expert parallelism for MoE models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multi-LoRA batching and parallel adapter serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

GPU topology and interconnect requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

NCCL and distributed communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Process layout and node configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 12: Multi-Node and Cluster Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Cluster topology and networking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Head node and worker node roles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Distributed tensor parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Cross-node KV-cache considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Fault tolerance and node failure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Configuration and launch procedures

Chapter 13: Prefill-Decode Disaggregation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

The prefill-decode split: motivation and theory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Disaggregated architecture components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Networking between prefill and decode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Load balancing and capacity allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

When disaggregation helps and when it does not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Deployment patterns and practical configuration

Chapter 14: Speculative Decoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

How speculative decoding works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Draft model selection and configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Performance gains and overheads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

When speculative decoding is worthwhile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Configuration and tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 15: APIs and Application Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

OpenAI-compatible API endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Streaming responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Sampling parameters and controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Tool/function calling endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multimodal request format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Authentication and request metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Error handling and client integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 16: Structured Generation and Constrained Decoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Grammar-based constrained generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

JSON and schema generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Regex constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Deterministic vs. probabilistic structured outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Performance overhead of constrained decoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Production patterns for structured outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 17: SGLang Model Gateway and Routing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

What the Model Gateway is and why it exists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Worker registration and discovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Routing strategies and load balancing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Health checks and failover

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multi-model deployment patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

HTTP and gRPC paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

TLS, mTLS, and API-key protection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Kubernetes integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Rate limiting and reliability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 18: Deployment Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Developer laptop and single GPU

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Dedicated inference server (multi-GPU)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multi-node on-prem cluster

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Kubernetes deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Cloud provider deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Production fleet patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Upgrade and rollback procedures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 19: Performance Engineering and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Key metrics: throughput, TTFT, ITL, GPU utilization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Benchmarking methodology and tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Workload generation and realism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Identifying bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Optimization as a decision process

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Memory and KV-cache optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Batching and concurrency tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Compute and kernel optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Speculative decoding optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Network and distributed optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Optimization profiles: latency vs. throughput vs. long-context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 20: Capacity Planning and Production Sizing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

GPU memory estimation (model, KV-cache, overhead)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Throughput and latency requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Concurrency and context-length impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Replication and scaling calculations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Network bandwidth planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Storage and model artifact planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Headroom and failure capacity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Autoscaling considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 21: Observability and Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Logs and structured logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Prometheus-compatible metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Metrics reference for operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Tracing and request diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

GPU monitoring and integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Dashboards and alerting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Health and readiness checks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Profiling and incident diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 22: Security and Production Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Network exposure and service isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

TLS and mTLS configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

API authentication and authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Secrets management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Container and Kubernetes security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model and artifact integrity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Logging sensitive data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Multi-tenant isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Rate limiting and DoS protection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Prompt injection and safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 23: Administration and Lifecycle Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Process supervision and service management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Configuration management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Rolling deployments and upgrades

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model replacement and cache management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Log rotation and storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Capacity expansion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Node replacement and migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Disaster recovery considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Operational runbooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 24: Troubleshooting and Failure Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Installation and dependency failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

CUDA/ROCm and driver problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model loading and tokenizer errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

GPU out-of-memory conditions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

KV-cache exhaustion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Initialization and startup failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Distributed communication errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Hangs, crashes, and degraded performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Structured output and tool calling failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Gateway and routing issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Kubernetes-specific problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Chapter 25: Configuration Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Server command-line arguments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Memory and cache parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Scheduling and batching parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Attention and performance parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Quantization parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Parallelism parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Distributed execution parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

LoRA parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Speculative decoding parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

MoE and expert parallelism parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

PD disaggregation parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Observability and debugging parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Security parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Advanced and internal parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Environment variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Model Gateway parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Conclusion: Building Resilient Inference Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Key principles for production SGLang

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Where SGLang is heading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Building resilient inference platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

Final thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thesglangproductionhandbook.