# Design Document

## 1. Introduction and Goals

### 1.1. Introduction

This document outlines the architecture and design of the Model Serving System, a distributed platform engineered to serve Large Language Models (LLMs) at scale. The system is built to handle high concurrency and deliver low latency streaming inference by strategically separating the data plane from the control plane. By leveraging a hybrid implementation—using C++ for performance critical components and Python for orchestration—the system achieves robust fault tolerance with automatic replica restarts.

### 1.2. Goals and Requirements

The system was designed with the following primary objectives :

- **High Concurrency:** Serve multiple distinct LLM model deployments to many users simultaneously.
- **Low Latency:** Achieve a fast time-to-first-token for generative models and maintain low median and tail latencies under load.
- **Streaming Support:** Provide real-time, streaming token outputs to clients.
- **Scalability:** Efficiently scale horizontally across distributed nodes to meet demand.
- **Fault Tolerance:** Implement automated detection and recovery of failed components to ensure system reliability.
- **Observability:** Expose key metrics for throughput, latency, and system health to enable comprehensive monitoring.

### 1.3. Background and Motivation

This project was initiated out of a desire to deeply understand the underlying infrastructure required for creating high performance, LLM streaming systems. This system was conceived as a hands on exploration to deconstruct the complexities of such an architecture, from handling thousands of concurrent connections at the network edge to orchestrating distributed computational resources and managing the intricate data flow needed to deliver a seamless, token-by-token experience to the end user. The goal was not just to build a functional model server, but to gain a comprehensive, practical understanding of the principles and trade-offs involved in engineering streaming systems.

The previous version of this system was implemented in Python end to end. This approach suffered from:

a) Python's Global Interpreter Lock (GIL) which limits true parallelism in I/O bound tasks.
b) Event loop contention when handling large numbers of concurrent streaming connections.
c) Higher per request latency especially in burst loads

The redesigned architecture moves **network bound and concurrency heavy code paths to C++**, while retaining Python in **model execution paths** where its ecosystem (e.g., vLLM, PyTorch) offers maximum productivity and library support. This hybrid approach reduces tail latency, increases concurrency, and maintains flexibility for rapid model iteration.

## 2. High-Level Architecture

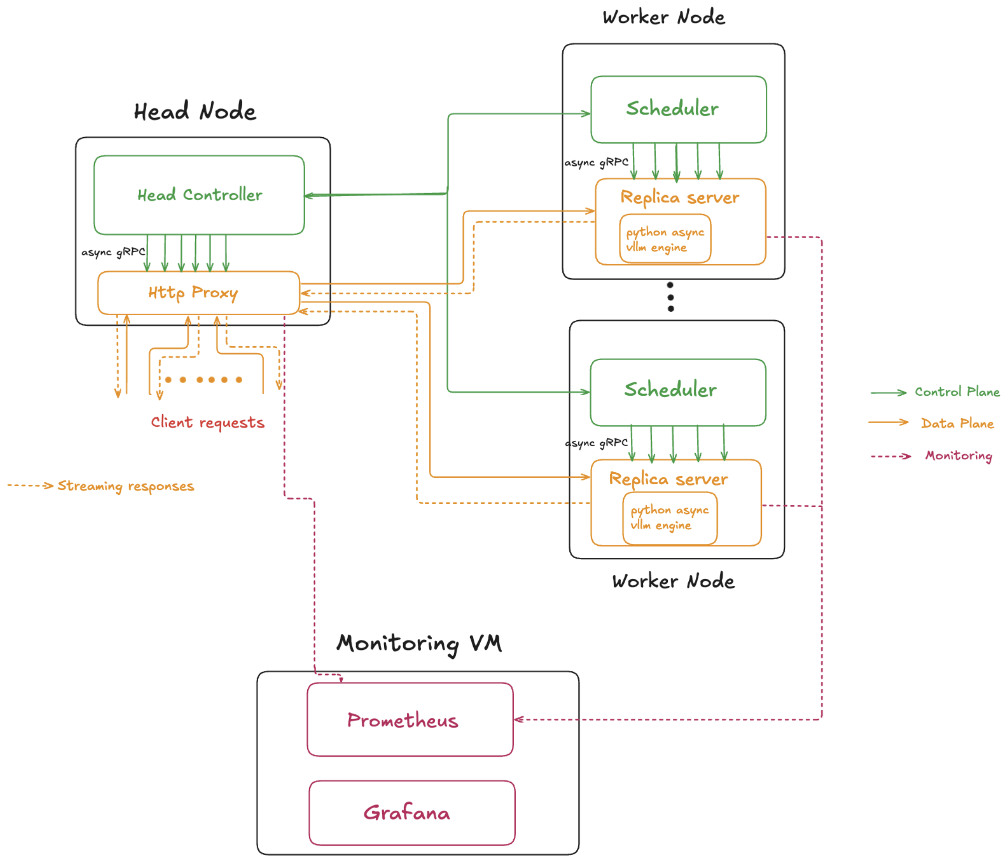
The system is split into two distinct planes:

1. Data Plane - Handles end-to-end inference requests, from client ingestion at an HTTP proxy through gRPC streaming to the model execution backend, and back to the client with token-by-token streaming.

2. Control Plane- Manages deployment metadata, replica lifecycle, routing updates, and health monitoring through a Head Controller and per-node Schedulers.

Fig (1) shows the overall design of the system, the system consists of the following major components:

- HTTP Proxy (C++) – Ingress service that accepts client HTTP requests, forwards to the backend replicas and streams tokens back via HTTP chunked encoding.
- Replica Server (C++ + Python) – Runs model inference with a C++ gRPC server for networking and embedded Python (vLLM) for token generation.
- Head Controller (Python) – Central control plane service running on Head Node, managing overall health of the system.
- Scheduler (Python) – Agent running on each Worker Node that launches, monitors, and restarts replica processes as instructed by the Head Controller.
- Monitoring VM - This is optionally started to scrape metrics for further analysis.



*Fig(1):Overall System*

## 3. Component Design

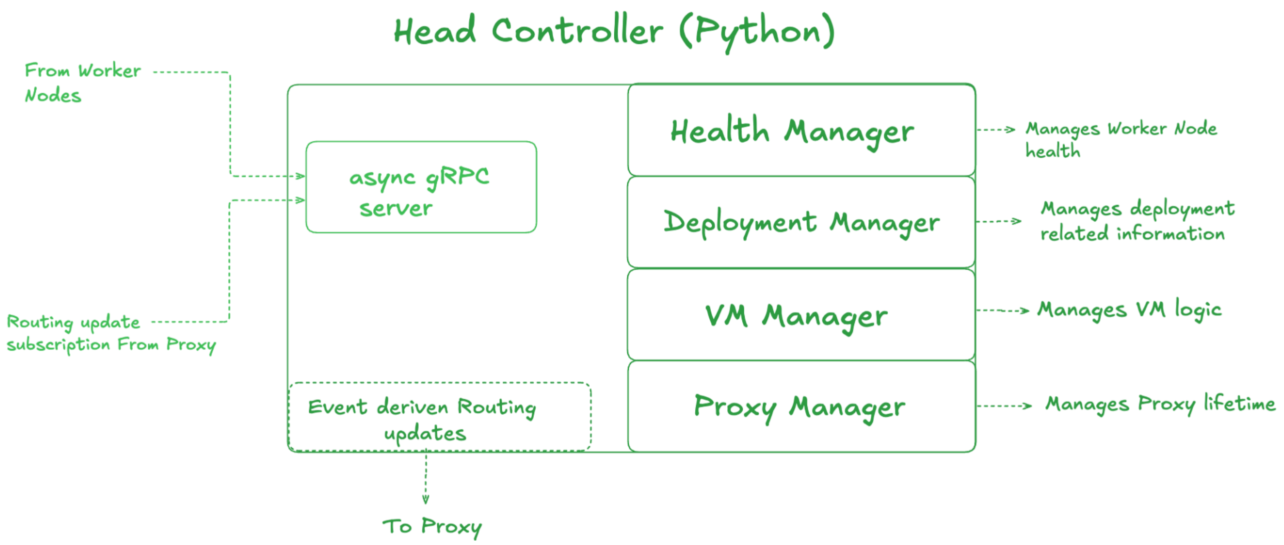
### 3.1. Head Node Components

#### Head Controller

- **Role**: The Head Controller is the cluster's central control plane service, implemented in asynchronous Python. It maintains the authoritative state of all deployments, replicas, and worker nodes, and is responsible for sending routing snapshots to proxy, orchestrating replica lifecycle events, and processing health and metrics updates from workers. Fig(2) shows the basic architecture
- **Concurrency model**: The Head Controller runs entirely within an `*asyncio* event loop,

allowing multiple long lived background tasks to run concurrently without blocking each other. Synchronization is handled via asynchronous locks to protect shared state and asynchronous events to notify subscribers when updates are ready.

- **Functional Components**: There are different components within the head controller responsible for various functions:
  - **Health Manager:** Responsible for tracking the health of all worker nodes. It ingests periodic health reports sent by workers and also performs its own active health pings, creating a dual layer monitoring mechanism. Any detected changes in worker health immediately update the in-memory deployment state, which is then propagated to HTTP proxies through the routing update stream as needed.
  - **Deployment Manager:** Responsible for managing the complete lifecycle of deployments. It supports adding or updating deployments and incorporating replica state changes, producing fresh, timestamped routing snapshots. These snapshots are updated dynamically based on health and metrics signals, enabling automatic replica count adjustments and timely routing updates to HTTP proxy.
  - **Proxy Manager:** Handles the full lifecycle of the HTTP Proxy. It is responsible for starting the proxy process, performing periodic health checks, and monitoring its operational status. If the proxy becomes unresponsive, the manager automatically triggers a restart to restore service availability.
  - **VM Manager:** Sets up and manages the provisioning of new worker VMs on the cloud provider(AWS). It configures each instance to start the scheduler or worker container with the necessary controller connection details, enabling the control plane to add or remove capacity as needed.
- **gRPC services:** It has two main gRPC interfaces exposed:
  - Proxy facing: Server streaming RPC delivering full snapshots every time a routing update is seen.
  - Worker facing: Handles worker registration, health updates, metrics ingestion.
- **Failure Handling**: The current Head Controller implementation does not provide fault tolerance and cannot be scaled horizontally because it is stateful. A potential future improvement is to make it stateless and support multiple synchronized replicas, enabling high availability and fault tolerance.

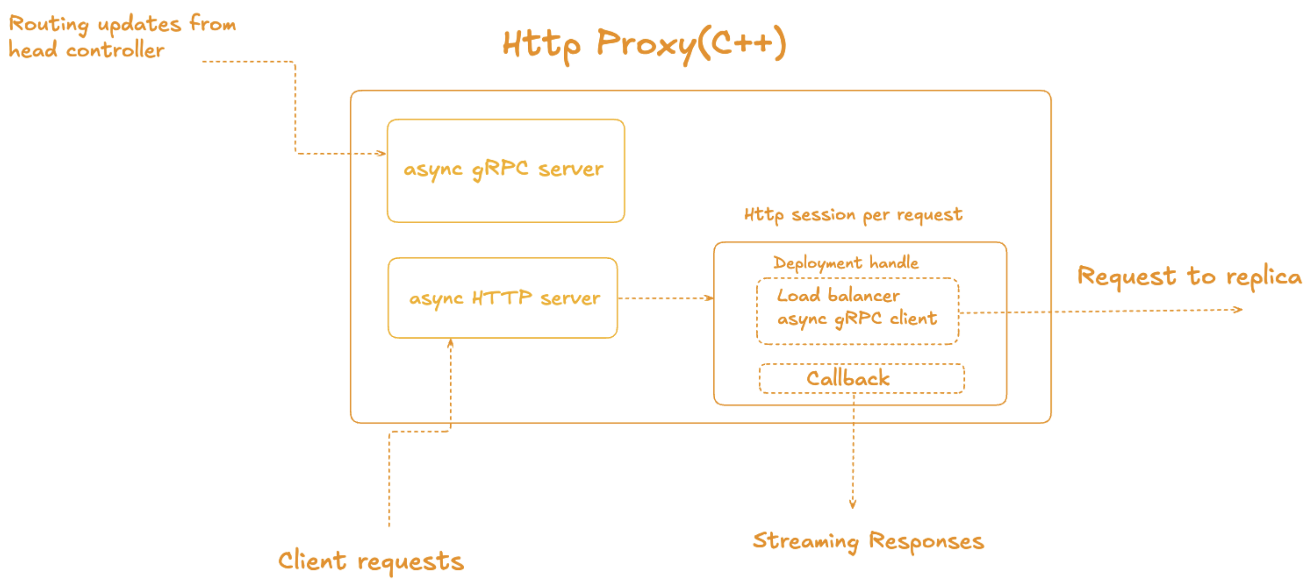


*Fig(2):Head Controller Design*

**HTTP Proxy**

- **Role**: The proxy is the system's sole entry point, implemented in high performance, asynchronous C++ to serve as a streaming gateway. It acts as an HTTP streaming gateway, forwarding client requests to replicas over gRPC and streaming tokens back to clients using HTTP chunked transfer encoding. Fig(3) shows the basic architecture
- **Routing and Selection**: The proxy maintains its routing state through a long lived gRPC subscription to the Head Controller, receiving updates that map model deployments to their available replicas. Requests are dispatched using a least-loaded selection policy.
- **Concurrency mode**l: HTTP I/O and gRPC communication are handled in separate asynchronous runtimes, isolating client request processing from backend calls. The HTTP server runs in a Boost.Asio I/O context, while the gRPC related communication happens in its own dedicated CompletionQueue context. All network operations are coroutine based, enabling non blocking sequential flow and fine grained backpressure using token streaming.
- **Failure Handling:** If the gRPC subscription to the Head Controller drops, the proxy automatically attempts to re-establish it using an exponential backoff strategy. In the event the proxy process itself goes down, the Head Controller restarts it as part of the system's process management. Because the proxy is stateless, any in-flight requests are

lost and must be re-issued by the client, while new requests are seamlessly routed once the proxy is back online.



*Fig(3): Http Proxy design*

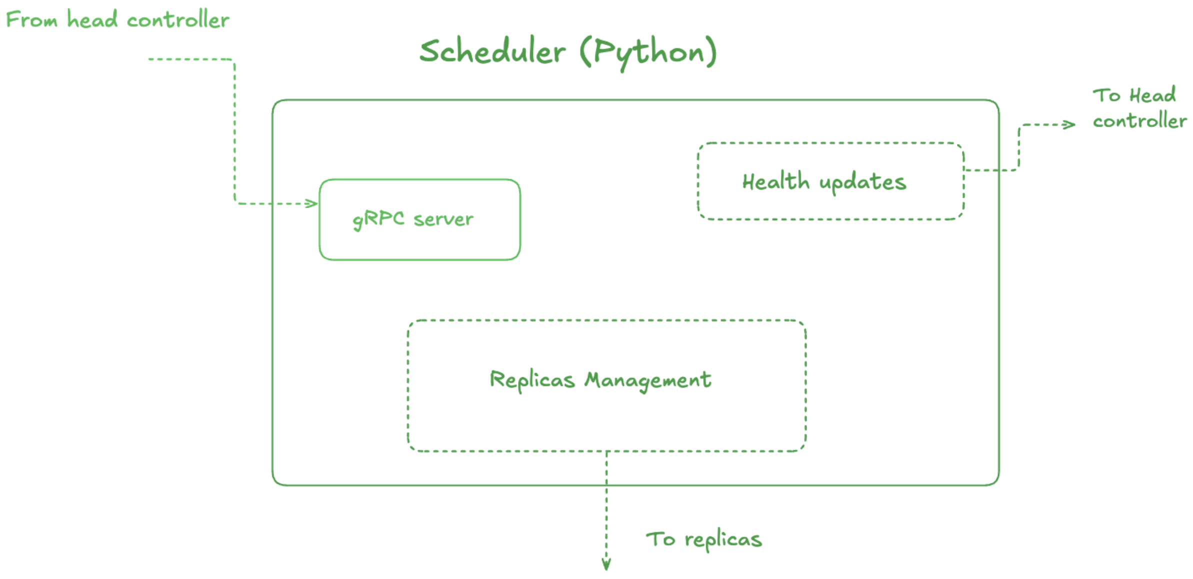
## 3.2. Worker Node Components

### Scheduler

- **Role:** The Scheduler is a per-node control plane agent that manages the lifecycle of all replica processes on its worker node implemented in asynchronous python. It receives commands from the Head Controller to create or terminate replicas and ensures each replica is launched with the correct configuration, resource allocation, and networking parameters. Fig(4) shows the basic architecture.
- **Concurrency**: The Scheduler runs as an asynchronous gRPC server in Python, handling incoming control plane RPCs while executing background tasks for replica health checks, restarts, and cleanup.
- **Lifecycle Operations**: On receiving a  replica creation command, the Scheduler spawns a C++ Replica Server process, registers it with the Head Controller, and adds it to an internal registry. Termination requests trigger process cleanup and registry removal. Any failed

replicas are restarted according to recovery policies.

- **Health Management:** The scheduler sends regular health status of the replicas to the Head Controller and actively pings each replica to verify responsiveness. Replica failures locally are retried before sending the status to the head controller.
- **Failure Handling:** If the Scheduler process goes down, it is restarted by the system's process manager, but any replicas running on the node are gracefully shut down and restarted by the head controller.



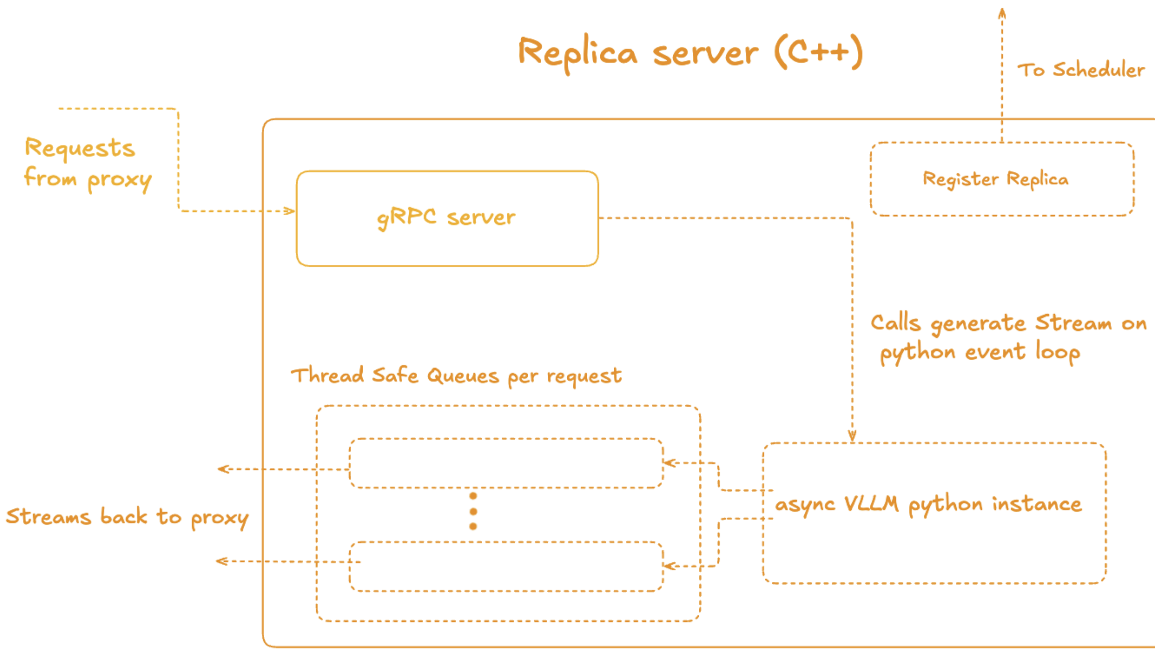
*Fig(4): Scheduler design*

## Replica Server

- **Role**: The Replica Server is responsible for executing model inference requests and streaming generated tokens back to the client. Implemented in C++ for high performance networking, it embeds a Python runtime to run the vLLM asynchronous generation engine. It exposes a gRPC server for streaming inference. Fig(5) shows the basic diagram.
- **Startup:** On startup, the server reads deployment configuration and initializes both C++ and Python runtimes. The Python side loads the model and starts the asynchronous vLLM engine. The C++ side then sets up the gRPC server, and registers the replica with its

Scheduler. After this both the runtimes operate concurrently with python running on a separate thread.

- **Request handling and Streaming:** When a request arrives over gRPC, the C++ Replica Server creates an asynchronous generation task and schedules it on the Python event loop. Tokens produced by the vLLM engine are streamed back incrementally over gRPC. The bridging between C++ and Python uses a thread safe queue, ensuring non-blocking transfer of data between runtimes.
- **Concurrency model:** C++ gRPC threads handle incoming requests and token streaming, while a dedicated Python thread runs the async generation loop. A thread safe queue bridges tokens between Python and C++. This separation ensures that generation and network I/O do not block each other.
- **Python VLLM Engine**: Generation leverages vLLM's continuous batching scheduler to maximize inference throughput. Concurrent requests are dynamically grouped into micro-batches on each decoding step, with preemption/resume and KV-cache paging to keep GPU utilization high.This batching is transparent to the proxy where each request's token stream remains ordered and independent.
- **Failure Handling**: If registration with the Scheduler fails, the process exits immediately. Transient request failures or client cancellations are handled gracefully, ensuring the pipeline drains cleanly and metrics are updated accordingly.



*Fig(5): Replica Server design*

## 3.3. Data Flow Summary

A typical request flows as follows :

1. A client sends an HTTP POST request to the HTTP Proxy.
2. The proxy selects the best available Replica and forwards the request via gRPC.
3. The Replica's C++ server invokes the Python vLLM engine to generate tokens.
4. As tokens are produced, they are streamed back to the proxy over gRPC, which in turn streams them to the client via an HTTP chunked response.

Meanwhile, control messages flow between the Head Controller and Schedulers, ensuring the data plane's routing information is always current.

## 4. Key Design Decisions

This section summarizes the guiding principles and architectural trade-offs behind the system design, serving as a quick reference

- **Implementation** – C++ is used for high-performance, low latency networking (HTTP proxy, replica server), while Python powers the ML runtime (vLLM)  and control plane logic.
- **Streaming Inference protocol** – Both HTTP and gRPC paths use streaming to minimize time-to-first-token (TTFT) and enable incremental output.
- **Asynchronous Concurrency** – C++20 coroutines and Boost.Asio handle non-blocking I/O and  Python uses `*asyncio* for parallel model inference.
- **Health Aware Routing** – HTTP proxy routes to the least loaded healthy replica, with temporary local health marking to avoid request drops before control plane updates arrive.
- **Continuous Batching (vLLM)** – The Python replica manager leverages vLLM's continuous batching to maximize throughput while preserving per request ordering.
- **Observability** – Prometheus metrics are exposed from both C++ and Python components for latency, throughput, and health tracking.

## 5. Conclusion

The current design delivers a modular, high performance model serving system capable of low latency, high throughput inference while maintaining observability and resilience.