Individual Diploma Thesis

EVALUATING IDLENESS OF MACHINE LEARNING MICROSERVICES ON GPUs

Constantinos Hadjieftychiou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS
Faculty of Pure and Applied Sciences
DEPARTMENT OF COMPUTER SCIENCE

EVALUATING IDLENESS OF MACHINE LEARNING MICROSERVICES ON GPUs

Constantinos Hadjieftychiou

Advisor
Dr. Haris Volos

The Individual Diploma Thesis was submitted for the partial fulfillment of the requirements for obtaining the degree in Computer Science of the Department of Informatics of the University of Cyprus

May 2026

Acknowledgments

I would like to sincerely thank Dr. Haris Volos for his support and invaluable guidance during the course of this thesis. His dedicated coordination and insightful advice were crucial in bringing this work to completion.

ABSTRACT

Modern machine learning applications are increasingly deployed as microservices that serve inference requests on GPU accelerators. Model serving frameworks such as TorchServe simplify deployment and scaling by providing REST/gRPC interfaces, request batching, model versioning, and monitoring support. Prior studies show that inference execution often contains substantial idle time, while GPUs continue to consume a significant fraction of peak power even when little useful work is being performed. TorchBench measurements reported in the thesis draft indicate that GPU active time during inference can be limited relative to overall execution time, which motivates closer study of idleness in serving systems.

This thesis presents a reproducible experimental framework for characterizing GPU idleness in inference microservices. A deep neural network is deployed using TorchServe and driven by controlled Poisson-distributed workloads that span underload, balanced load, and saturation regimes. During execution, fine-grained telemetry is collected, including GPU power, utilization, request arrival and completion counts, queue backlog, and request latency. These measurements are aggregated into fixed-duration windows to construct a labeled telemetry dataset that captures GPU behavior over time.

The resulting dataset is used to define and analyze GPU idleness using both request-level and energy-based criteria. The thesis shows how idle periods correlate with workload intensity, queue dynamics, and power consumption, and it establishes a foundation for future predictive and energy-aware scheduling mechanisms in ML serving systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

- API Application Programming Interface
- CPU Central Processing Unit
- GPU Graphics Processing Unit
- ML Machine Learning
- DNN Deep Neural Network
- RPS Requests Per Second
- CDF Cumulative Distribution Function
- NVML NVIDIA Management Library
- MAR Model Archive (TorchServe format)
- HTTP Hypertext Transfer Protocol
- CSV Comma-Separated Values
- DVFS Dynamic Voltage and Frequency Scaling
- TEE Trusted Execution Environment
- λ Arrival Rate (Requests per Second)
- μ Service Rate (Requests per Second)

1 Introduction

1.1 Background and Motivation

Recent advances in deep learning have led to the widespread adoption of GPU-accelerated inference services in modern cloud and edge computing environments. Applications such as image classification, natural language processing, recommendation systems, and real-time analytics increasingly rely on machine learning inference deployed as scalable microservices. Frameworks such as TorchServe enable these models to be exposed through network-accessible APIs, allowing requests from multiple users and applications to be processed concurrently.

GPUs are well suited for deep learning inference because of their high degree of parallelism and ability to efficiently execute tensor operations. However, high computational capability does not necessarily imply continuous utilization. Prior work on inference serving systems has shown that request-driven workloads often lead to intermittent GPU activity and periods of underutilization, particularly under variable or low request rates. Inference serving therefore differs from traditional high-performance computing workloads, where GPUs are typically driven by long-running compute-intensive jobs.

In many production inference systems, workloads exhibit several common characteristics:

- relatively small request sizes,
- irregular or stochastic inter-arrival patterns,
- latency-sensitive execution requirements.

These characteristics create highly dynamic execution behavior. Request arrivals are generated asynchronously by independent users or applications and are commonly modeled using stochastic processes such as Poisson arrivals. At the same time, inference serving frameworks frequently rely on queueing and batching mechanisms to improve throughput and hardware efficiency. As a result, requests may accumulate temporarily before being executed together on the GPU.

The interaction between stochastic arrivals and batched execution produces burst-like processing behavior. Instead of operating continuously at a stable utilization level, the GPU alternates between periods of activity and periods of low utilization or idleness. This temporal variability becomes particularly important in microservice-based inference systems, where workload intensity may fluctuate significantly over time. In real-world deployments,

workload intensity is rarely constant. User interactions, diurnal traffic patterns, and external application events continuously change the rate at which requests arrive at the system. These workload variations cause the system to transition between different operating regimes, including underloaded conditions, balanced operation, and saturation. Such transitions are reflected in observable metrics such as request arrival rate, queue backlog, GPU utilization, and power consumption.

These dynamics have important implications for energy efficiency. Modern GPUs maintain a substantial baseline power draw even during periods of low utilization. Unlike CPUs, which can aggressively reduce power consumption using deep idle states, GPUs often remain partially active due to memory subsystems, execution contexts, and communication interfaces that must remain operational. Consequently, idle or underutilized periods may still incur significant energy cost.

This behavior indicates that GPU energy consumption is not fully energy proportional. In an energy-proportional system, power consumption scales closely with workload intensity. GPU inference systems, however, may continue consuming relatively high power even when performing little useful work. Improving efficiency therefore requires not only optimizing computation itself, but also understanding when and why idle periods occur. The observable relationship between workload dynamics, queue state, and GPU activity motivates further investigation into GPU idleness in inference systems. Rather than treating idleness as a purely hardware-level phenomenon, this thesis studies how it emerges from the interaction between stochastic request arrivals, queueing behavior, and GPU execution dynamics.

The focus of this thesis is to characterize GPU idleness behavior in machine learning inference microservices using externally observable system metrics. By combining workload traces, request logs, and GPU telemetry, the study investigates how queue dynamics, power behavior, and workload intensity relate to idle periods and overall system efficiency.

1.2 Problem Statement

This thesis studies GPU-based inference microservices, with a focus on understanding how workload dynamics and queueing behavior influence GPU activity and idle periods during execution. The system under study consists of a machine learning model deployed using TorchServe, where clients submit inference requests over HTTP to a GPU-backed serving pipeline. Due to stochastic request arrivals and the batching behavior of the inference

framework, GPU execution is not continuous. Instead, the system alternates between periods of active computation and periods where the GPU is underutilized or waiting for new work.

GPU activity is partially observable through telemetry interfaces such as NVML and `nvidia-smi`, which expose metrics including utilization, power consumption, and clock-state information. However, these measurements are coarse-grained and do not directly reveal the internal execution state of the inference pipeline, such as batching decisions, queue state, or per-request scheduling behavior. As a result, understanding GPU idleness requires combining multiple observable signals rather than relying on a single hardware metric.

To analyze system behavior, execution is divided into fixed-duration time windows. For each interval, workload statistics, queue state, and GPU telemetry are aggregated into a unified representation. Observable features include request arrivals, request completions, estimated queue backlog, GPU utilization, and power consumption. This time-aligned view enables the study of how queue buildup, workload bursts, and batching behavior interact with GPU execution over time.

The core problem addressed in this thesis is to characterize how GPU idle behavior emerges from the interaction between workload dynamics, queueing effects, and GPU execution in inference microservices. A key challenge is that no single metric provides a complete view of system state. GPU utilization may remain non-zero during low activity, while power consumption often stays high even when useful work is limited. Similarly, latency measurements may hide queue buildup due to batching effects. Consequently, GPU idleness must be inferred from the combined behavior of multiple imperfect and temporally aggregated signals.

1.3 Research Objectives

The objectives of this thesis are framed as measurement, modeling, and analysis goals:

1. Design a reproducible GPU inference testbed using TorchServe
2. Generate controlled workloads using Poisson arrival processes
3. Collect synchronized telemetry for:
    - request activity
    - latency
    - GPU power and energy
    - GPU utilization

4. Construct a window-based dataset aligning workload and hardware behavior
5. Define and compare multiple notions of idleness
6. Characterize how idleness varies across load regimes:

$$\rho = \frac{\lambda}{\mu}$$

where λ is arrival rate and μ is service rate

1.4 Contributions of the Thesis

This thesis makes several contributions toward understanding GPU idleness in inference microservices.

First, it introduces a reproducible experimental testbed for studying GPU-based inference under controlled conditions. The system integrates TorchServe for model serving, a Poisson-based workload generator to simulate realistic request arrivals, and a monitoring subsystem that captures GPU power and utilization. This setup enables systematic experimentation while minimizing external sources of variability.

Second, the thesis constructs a fine-grained dataset that captures system behavior over time. By aggregating request activity and GPU telemetry into fixed-duration windows, it provides a unified view of workload intensity, queue state, and hardware activity. This window-based representation allows the analysis to move beyond coarse averages and instead examine the temporal dynamics of GPU utilization and idleness.

Third, the work develops a practical interpretation of GPU idleness based on observable system behavior. Rather than relying on a single rigid definition, idleness is examined through multiple signals, including request completion patterns and hardware activity indicators such as power and utilization. This perspective reflects how idleness appears in real systems, where it cannot be directly measured but must be inferred.

Finally, the thesis provides a system-level analysis of inefficiency in inference workloads. It shows how idleness emerges from the interaction between workload intensity, queueing behavior, and GPU execution dynamics. This analysis establishes a foundation for future work on predictive modeling and energy-aware resource management.

2 Background and Related Work

2.1 GPU-Accelerated Machine Learning Inference

Modern machine learning inference systems rely heavily on GPUs due to their ability to exploit massive data parallelism in neural network computations. Unlike CPUs, which are optimized for sequential execution, GPUs are designed to execute thousands of threads in parallel, making them particularly well-suited for matrix operations and tensor computations that dominate deep learning workloads.

In production environments, trained models are deployed behind serving systems that translate incoming requests into GPU-executed inference operations. These systems abstract the complexity of model execution and expose a simple request-response interface. Common examples include NVIDIA Triton Inference Server and TorchServe, both of which provide standardized mechanisms for model deployment, request handling, and hardware utilization. A key distinction exists between training and inference workloads, which fundamentally affects how GPUs are utilized. Training workloads are typically batch-oriented, long-running, and compute-bound, allowing GPUs to operate at consistently high utilization. In contrast, inference workloads are request-driven, short-lived, and latency-sensitive. Requests arrive asynchronously and must often be processed individually or in small batches, leading to irregular execution patterns.

As a result, GPU utilization during inference is often significantly lower than during training. Empirical studies have shown that inference workloads contain substantial periods of reduced activity, even under moderate load. For example, benchmarks such as TorchBench report that active execution accounts for only a portion of total runtime, with the remainder consisting of inter-request gaps, kernel launch overhead, synchronization delays, and data transfers. This produces a characteristic temporal behavior in which GPU execution alternates between bursts of activity and periods of reduced utilization.

Modern GPU monitoring interfaces such as NVML expose hardware-level activity indicators, including utilization counters, power measurements, and clock event reasons such as `nvmlClocksEventReasonGpuIdle`. However, these signals provide only a partial view of system behavior. They do not directly expose higher-level inference dynamics such as request queueing, batching decisions, or per-request scheduling delays inside the serving framework. In addition, utilization and power measurements are typically sampled at coarse time granularity, which can obscure short transient idle periods. Consequently, average GPU

utilization alone is insufficient to characterize inference-system efficiency or explain how idle behavior emerges over time.

2.2 Microservices for Machine Learning Systems

Microservices architecture is widely used in modern software systems to improve modularity, scalability, and maintainability. In this paradigm, applications are decomposed into independent services that communicate through lightweight interfaces, typically over HTTP. Each service encapsulates a specific functionality and can be developed, deployed, and scaled independently.

In machine learning deployments, this architectural style is commonly adopted to structure the inference pipeline. Rather than embedding all functionality into a single monolithic application, different stages of the inference process are separated into logically distinct components. For example, a typical deployment may include an inference service responsible for executing the model, alongside auxiliary components responsible for request handling, monitoring, logging, and orchestration. A point that requires clarification is the role of data preprocessing. While preprocessing is indeed part of the training pipeline, it also exists in inference systems as a runtime operation. Incoming requests such as images, text, or structured data must be transformed into the numerical tensor format expected by the model. This may involve operations such as resizing images, normalization, tokenization, or feature extraction. In practice, this preprocessing step can either be implemented inside the model-serving framework (e.g., within a TorchServe handler) or as a separate microservice in more complex deployments. The key idea is that preprocessing is applied per request at inference time, not only during training.

The use of microservices introduces flexibility but also leads to variability in system behavior. Inference services receive requests from multiple independent clients, which results in asynchronous and often bursty traffic patterns. Unlike tightly controlled batch-processing systems, there is no guarantee of steady input rates. Instead, request arrivals depend on external factors such as user activity, application demand, or upstream services. This variability has direct implications for GPU utilization. Because requests arrive irregularly, the GPU is not continuously fed with work. Instead, it alternates between periods of high activity when multiple requests are processed and periods of low or no activity, when the request queue is empty. These transitions occur at fine time scales and are a primary source of the idleness phenomenon studied in this thesis.

From a systems perspective, the key challenge introduced by microservices is the decoupling between request arrival and execution. The inference service must react to incoming requests rather than control their timing. As a result, resource utilization is driven by workload dynamics rather than by deterministic scheduling. Understanding how this interaction produces idle periods is central to analyzing GPU efficiency in inference workloads.

2.3 Model Serving Frameworks

Model serving frameworks provide the infrastructure required to deploy trained machine learning models in production environments. These systems abstract the complexity of model execution by exposing standardized interfaces for inference, while internally managing request handling, scheduling, and execution on hardware accelerators such as GPUs.

Several serving systems are widely used in practice. TensorFlow Serving is an open-source system designed for serving TensorFlow models, with strong support for versioning and high-performance inference in production pipelines. NVIDIA Triton Inference Server extends this functionality by supporting multiple deep learning frameworks (e.g., TensorFlow, PyTorch, ONNX) within a unified serving environment and includes advanced features such as dynamic batching and concurrent model execution across GPUs. Multi-model serving systems, such as those used in cloud platforms, enable multiple models to be hosted simultaneously on shared infrastructure, improving resource utilization through consolidation and scheduling. TorchServe, which is the focus of this thesis, is a PyTorch-specific model server that provides a lightweight and modular interface for deploying models with minimal configuration overhead.

Despite differences in implementation, these frameworks share a common set of core functionalities. They provide network-accessible inference endpoints (typically REST or gRPC APIs), manage model lifecycle operations such as loading and unloading, support request batching to improve throughput, and expose logging and monitoring hooks for system observability. Internally, they maintain request queues and dispatch incoming requests to worker processes that execute the model on the underlying hardware.

2.4 GPU Utilization and Energy Efficiency

Energy efficiency is a critical concern in modern machine learning systems, particularly for inference workloads where utilization varies over time. A key factor underlying this inefficiency is that hardware power consumption does not scale proportionally with workload intensity.

On CPUs, power management mechanisms are relatively mature. Modern processors implement C-states, which are idle power states that progressively shut down parts of the processor when it is not executing instructions. In deeper C-states, components such as caches, clock domains, and execution units are partially or fully disabled, significantly reducing power consumption. As a result, CPUs can achieve low power draw during idle periods, especially when idle intervals are sufficiently long and predictable.

GPUs, however, operate under different constraints. While they also support power management techniques such as Dynamic Voltage and Frequency Scaling (DVFS) and clock gating, these mechanisms are generally less effective at fine-grained time scales. GPU inference workloads consist of short bursts of kernel execution interleaved with idle gaps that may last only a few milliseconds. These intervals are often too short for the hardware to transition into deeper low-power states. Additionally, GPUs maintain active memory systems, driver contexts, and execution pipelines even when not actively computing, which contributes to a relatively high static (baseline) power consumption.

As a result, GPU power can be approximated as:

$$P \approx P_{\text{static}} + P_{\text{dynamic}}$$

where P_{static} represents baseline power consumption and P_{dynamic} depends on active computation. In practice, P_{static} constitutes a large fraction of total power.

In the experimental setup used in this thesis (NVIDIA Tesla V100 of UCY HPC), empirical measurements show that the GPU consumes approximately 50–60 W at idle, even when no inference requests are being processed. Under load, power increases depending on the request rate. This relatively small difference between idle and active power highlights a fundamental limitation: the GPU remains energy-intensive even when performing little or no useful work.

This behavior has important implications for inference workloads. Because requests arrive asynchronously, the GPU alternates between short periods of activity and idle gaps.

However, during these idle intervals, power consumption remains close to the baseline level. Consequently, energy consumption is not proportional to useful work performed.

To quantify this, consider the energy per request:

$$e = \frac{E}{N}$$

where E is the total energy consumed and $N$is the number of completed requests. At low request rates, the system processes few requests while still consuming baseline power over time, leading to high energy per request. As the workload increases and the GPU becomes more continuously utilized, energy consumption is amortized across more requests, improving efficiency.

It is important to distinguish between utilization and efficiency. High utilization does not necessarily imply optimal energy usage, and low utilization does not imply low power consumption. In GPU-based inference systems, inefficiency often arises from temporal fragmentation, where useful computation is interleaved with idle periods that still incur significant energy cost.

For clarity of structure, this section focuses on hardware-level power behavior. The interaction between workload characteristics (such as arrival patterns and batching) and energy efficiency is analyzed in later chapters, where experimental results provide a more detailed view of how idleness emerges and affects system performance.

3 TorchServe Architecture and Deployment

3.1 Overview of TorchServe

TorchServe is an open-source model serving framework designed to deploy PyTorch models behind a network-accessible API. It provides a standardized interface for receiving inference requests, executing model computation on accelerator hardware, and returning responses to clients.

In this thesis, TorchServe serves as the inference substrate, enabling controlled experimentation on GPU behavior under varying workloads. It exposes a clear request–response abstraction, where each incoming HTTP request corresponds to an inference operation submitted to the serving pipeline. This abstraction is important because it allows the system to be studied in terms of observable inputs (request arrivals) and outputs (completed responses), without requiring direct access to internal execution details.

TorchServe is particularly suitable for this study for several reasons. First, it implements a production-like serving pipeline, including request queueing, worker management, batching, and GPU-backed execution. This ensures that the observed behavior reflects realistic system dynamics rather than simplified or synthetic models. Second, its architecture is sufficiently modular to allow the system to be treated as a black box, which aligns with the goal of characterizing GPU behavior from external telemetry. Third, TorchServe allows precise control over key configuration parameters, such as the number of workers, batch size, and GPU allocation, enabling isolation of workload effects without introducing additional system complexity.

From a systems perspective, TorchServe can be approximated as a queue-driven service system with GPU-backed execution. Incoming requests arrive according to a stochastic process and are placed into an internal request queue. A worker process dequeues requests from the queue and executes them on the GPU. Depending on the batching configuration and request availability, execution may involve either a single request or a batch of multiple requests processed together.

Although batching introduces state-dependent service behavior, the system can still be approximated as a single logical server for analytical purposes because all requests pass through a single worker and a single GPU execution pipeline. This abstraction allows the

system to be analyzed using concepts from queueing theory while still capturing the temporal effects introduced by batching.

A key parameter in this model is the utilization ratio:

$$\rho = \frac{\lambda}{\mu}$$

where $\lambda$is the arrival rate (requests per second) and $\mu$is the effective service rate of the inference pipeline.

Strictly speaking, the system is not a classical M/M/1 queue. Inference execution times are not exponentially distributed, and batching causes the effective service rate to vary with queue state. However, because the configuration used in this thesis employs a single worker and a single GPU execution pipeline, the system can still be modeled approximately as a single logical server with state-dependent service behavior. This abstraction is sufficient for reasoning about queue buildup, utilization regimes, and idle behavior.

The utilization ratio provides a compact way to describe the operating regime of the system:

- When $\rho \ll 1$, the arrival rate is much lower than the service rate. The system has excess capacity, requests are processed quickly, and the queue remains mostly empty. In this regime, the GPU frequently becomes underutilized because there is insufficient incoming work to sustain continuous execution.
- When $\rho \approx 1$, the arrival rate approaches the service rate. The GPU remains active most of the time, and the system operates near its maximum sustainable throughput. Queue buildup becomes more common and idle periods become less frequent.
- When $\rho > 1$, the arrival rate exceeds the service rate. The GPU cannot keep up with incoming requests, causing backlog growth and increased queueing delay. Over time, the system enters a saturated regime.

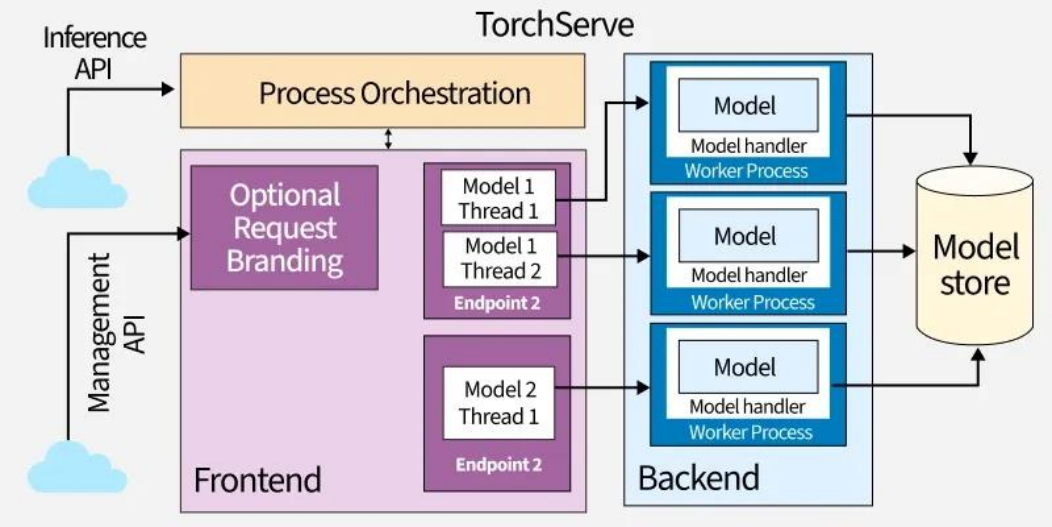
Thus, the utilization ratio does not directly determine idleness, but instead governs the balance between incoming work and processing capacity, which in turn shapes queue buildup, batching behavior, and GPU activity patterns.

Any figure included in this section should focus specifically on the simplified request-processing pipeline used in this thesis rather than the full TorchServe architecture. A suitable abstraction is:

Client → Inference API → Request Queue → Worker → GPU Execution → Response

This representation directly reflects the execution flow analyzed throughout the thesis and aligns with the queueing abstraction introduced above.

Figure 3.1: Simplified TorchServe request-processing pipeline used in this thesis



3.2 TorchServe System Architecture

TorchServe organizes its functionality around two primary interfaces: the inference API and the management API. The inference API is responsible for handling prediction requests submitted by clients and returning model outputs. The management API is used to control the lifecycle of deployed models, including loading, unloading, and scaling model instances. This separation decouples operational control from runtime inference execution.

Models in TorchServe are packaged as `.mar` (Model Archive) files and stored in a model store accessible to the TorchServe process. Each archive contains the serialized model, a handler script, and optional metadata or dependencies. The handler script defines how incoming requests are processed, including input decoding, preprocessing, model execution, and output formatting. In this sense, the handler acts as the interface between external request data and internal GPU execution.

At startup, TorchServe loads model archives from the model store, initializes the corresponding models, and spawns worker processes. Each worker represents an independent execution instance capable of serving requests. In the configuration used in this thesis, a single worker is employed. This simplifies the execution model by ensuring that all requests pass through a single queue and GPU execution pipeline, making system dynamics easier to

analyze. However, this configuration may also increase queueing delay under high load and can expose idle periods more clearly than multi-worker deployments. The goal is not to maximize throughput, but to isolate and study the relationship between workload dynamics, batching behavior, and GPU activity.

Internally, TorchServe follows a structured request-processing pipeline:

Client → Inference API → Request Queue → Worker → GPU Execution → Response

Each component in this pipeline has a distinct role. The client submits an inference request to the inference API, which acts as the system entry point. The request is then inserted into an internal queue, where it waits until the worker becomes available. The worker retrieves the request, invokes the handler for preprocessing, executes the model on the GPU, and returns the generated response to the client.

This pipeline introduces several sources of latency. Queueing delay occurs when requests accumulate while waiting for execution. Preprocessing delay arises from input transformation and decoding. GPU execution time corresponds to the model forward pass, while postprocessing delay includes output formatting and serialization. The sum of these components determines the end-to-end latency observed by the client.

A critical aspect of TorchServe's execution model is batching. In deep learning systems, batching refers to grouping multiple requests together and processing them within a single GPU execution. GPUs are optimized for parallel tensor operations, making batched execution substantially more efficient than executing many small inference calls independently. As a result, batching improves throughput and increases hardware utilization.

However, batching also introduces important temporal effects. Requests may wait briefly in the queue while additional requests accumulate before execution begins. When a batch is executed, multiple requests complete within a short time interval, producing burst-like completion patterns.

In TorchServe, batching behavior is controlled through parameters such as `batch_size` and `max_batch_delay`. The `max_batch_delay` parameter specifies the maximum amount of time a request may wait before the batch is executed, even if the batch is not full. In practice, batching delays are typically on the order of a few milliseconds to tens of milliseconds, depending on configuration and latency requirements. This mechanism introduces a tradeoff between latency and throughput: larger batching windows improve GPU efficiency but increase request waiting time.

The interaction between batching and queueing creates bursty execution behavior. Requests accumulate in the internal queue until they are processed individually or as part of a batch, after which multiple completions may occur in a short interval. Under low load, the queue is frequently empty and the GPU experiences periods of reduced activity between requests. Under high load, backlog accumulates and the GPU operates more continuously. These dynamics directly influence the temporal patterns of utilization and idleness analyzed later in the thesis.

The complete TorchServe architecture includes additional components such as request routing, endpoint management, and multi-model support. These features are not relevant to the simplified single-model, single-worker configuration used in this thesis and are therefore omitted from the analysis.

Overall, TorchServe can be viewed as a queue-driven execution system where the interaction between request arrivals, queue dynamics, batching behavior, and GPU execution determines system behavior. Understanding this execution pipeline is essential for interpreting the temporal patterns of backlog, utilization, power consumption, and idle behavior analyzed in later chapters.

3.3 Model Packaging and Deployment

To deploy a model in TorchServe, it must first be packaged into a `.mar` (Model ARchive) file. This archive encapsulates all components required for inference, including the serialized model, the request handler, and any auxiliary files such as configuration data or label mappings. Packaging the model in this way ensures that deployment is reproducible and independent of the surrounding system environment, which is essential for controlled experimentation.

In this thesis, the deployed model is ResNet-18 (and later ResNet-50), a convolutional neural network used for image classification. The model is first converted into a TorchScript representation, which allows it to execute independently of the Python interpreter and enables more portable deployment. The serialized TorchScript model, together with a custom handler, is then packaged using the `torch-model-archiver` tool.

Listing 3.2: Command used to package the TorchScript ResNet-18 model and its handler into a TorchServe-compatible `.mar` archive

```
cmd = [
    "torch-model-archiver",
    "--model-name", model_name,
    "--version", str(version),
    "--serialized-file", os.path.abspath(scripted_model_path),
    "--handler", os.path.abspath(handler_path),
    "--export-path", output_dir,
    "--force",
]
```

This command specifies the model name and version, the path to the serialized TorchScript model, and the handler script responsible for request processing. The `--export-path` argument determines where the resulting `.mar` file is stored, typically inside the TorchServe model store directory.

The resulting archive acts as a self-contained deployment artifact. It preserves the same model weights, preprocessing logic, and execution behavior across all experimental runs, eliminating inconsistencies caused by external dependencies or environment differences.

The handler plays a central role in the inference pipeline. It defines how raw request data is transformed into a format suitable for model execution and how model outputs are returned to the client. An excerpt of the handler logic used in this work is shown in Listing 3.2.

Listing 3.3: Core handler logic for decoding input data, applying preprocessing, executing the model on the GPU, and producing the output

```
image = Image.open(io.BytesIO(request_body)).convert("RGB")
input_tensor = self.transforms(image).unsqueeze(0).to(self.device)

with torch.no_grad():
    output = self.model(input_tensor)
```

The deployed model used in this thesis is ResNet-18, which performs image classification. As a result, each inference request contains image data that must be decoded and transformed before execution. First, the raw request body is interpreted as an image and converted into a standard RGB representation. This ensures that the input format remains consistent regardless of the original image encoding. Next, preprocessing transformations such as resizing, normalization, and tensor conversion are applied before the input is transferred to the GPU.

The tensor is augmented with a batch dimension using `unsqueeze(0)` and moved to the target device. Model execution occurs within a `torch.no_grad()` context, which disables

gradient computation during inference. Since inference does not require backpropagation, disabling gradients reduces both memory usage and computational overhead.

Each stage of this pipeline input decoding, preprocessing, device transfer, and inference execution contributes to the overall latency and behavior of the system. By explicitly defining these steps within the handler, TorchServe ensures that request processing remains transparent and reproducible.

The packaging and deployment process is critical for the validity of the experimental results. By encapsulating both the model and its execution logic within a single artifact, the system guarantees that all experiments are conducted under identical conditions. This ensures that any observed differences in performance, queue behavior, or GPU activity arise from workload dynamics rather than inconsistencies in deployment configuration.

3.4 Request Processing Pipeline

Once deployed, the system processes inference requests through a structured sequence of stages. Each request originates from the client, enters the TorchServe system through the inference API, and progresses through several internal components before a response is returned.

At a high level, the processing pipeline consists of the following steps. A request is first received at the inference endpoint and inserted into an internal request queue. When the worker becomes available, the request is selected for execution, optionally grouped into a batch with other pending requests, processed by the handler, executed on the GPU, and finally returned to the client as a response.

To analyze this pipeline quantitatively, each request is associated with a sequence of timestamps that capture its progression through the system:

- t_{send}: the time at which the request is issued by the client
- t_{queue}: the time at which the request enters the server-side queue
- t_{batch}: the time at which the request becomes part of an execution batch
- t_{exec}: the time at which GPU execution of the batch begins
- t_{comp}: the time at which the request is completed and a response is produced

These timestamps define the lifecycle of a request:

$$t_{send} \rightarrow t_{queue} \rightarrow t_{batch} \rightarrow t_{exec} \rightarrow t_{comp}$$

The interval between t_{queue} and t_{batch} represents batching delay, during which requests wait for additional requests to accumulate or for the batching timeout to expire. The interval between t_{batch} and t_{exec} corresponds to scheduling and preprocessing overhead before GPU execution begins.

Queue-related waiting time can therefore be expressed as:

$$W_q = t_{exec} - t_{queue}$$

which includes both queue waiting time and batching delay.

The interval between t_{exec} and t_{comp} corresponds to the service time, including preprocessing, data transfer to the GPU, model execution, and postprocessing. The total end-to-end latency experienced by the client is:

$$W = t_{comp} - t_{send}$$

Importantly, the request-processing pipeline is not treated as an opaque system. Instead, it is the mechanism that generates observable system-level behavior. When requests arrive faster than they can be processed, they accumulate in the queue, causing backlog growth and increased waiting time. Similarly, because GPU execution often occurs in batches, multiple requests may complete within a short interval, producing burst-like completion patterns.

These observable effects — queue buildup, latency variation, and bursty execution — are direct consequences of the pipeline structure described above. They form the basis for the analysis performed in later chapters, where system behavior is studied through aggregated window-level metrics and used to characterize GPU utilization and idle behavior.

3.5 TorchServe Deployment in This Thesis

In this work, TorchServe is deployed in a controlled single-node, single-GPU configuration. The objective of this setup is not to maximize throughput, but to eliminate unnecessary sources of variability so that system behavior can be analyzed in a controlled and interpretable manner.

The configuration is specified through the `torchserve.properties` file.

Listing 3.4: TorchServe configuration used in the experimental setup

```
number_of_gpu=1
default_workers_per_model=1

inference_address=http://127.0.0.1:8080
management_address=http://127.0.0.1:8081
metrics_address=http://127.0.0.1:8082
```

Each parameter plays a specific role in shaping system behavior.

Setting `number_of_gpu=1` ensures that all inference execution is performed on a single GPU device. This removes effects related to multi-GPU scheduling, synchronization, and load balancing. As a result, all observed behavior can be attributed to a single execution resource.

Setting `default_workers_per_model=1` means that only one worker process is responsible for handling incoming requests. This is an important design choice because multiple workers would introduce additional concurrency at the application level. By restricting the system to a single worker, all requests pass through a single execution pipeline controlled by one worker. Although multiple requests may be processed together as part of a batch, only one batch execution can be active at a time.

Binding all endpoints to `127.0.0.1` ensures that communication between the workload generator and TorchServe occurs locally within the same machine. This removes variability introduced by external networking effects such as packet delay or network congestion, ensuring that timing measurements reflect internal system behavior.

Taken together, this configuration produces several important properties. First, there is a single logical execution pipeline through which all requests pass. Second, there is no parallel request processing across multiple workers, simplifying the relationship between workload arrival rate and service behavior. Third, queue dynamics become easier to analyze because they depend only on the arrival process and the execution behavior of a single worker, rather than on scheduling interactions across multiple workers.

This simplified setup is essential for the goals of the thesis. It allows workload characteristics such as arrival rate and burstiness to be related directly to observable system behavior, including backlog formation, latency variation, batching behavior, and GPU utilization.

The deployment operates as part of a controlled open-loop experimental system. The workload generator produces inference requests according to a predefined arrival process and sends them to TorchServe independently of request completion times. In other words, request generation is not delayed or adjusted based on observed latency or GPU state.

TorchServe receives incoming requests, places them into an internal queue, and dispatches them to the worker for execution on the GPU. At the same time, a monitoring subsystem records GPU telemetry metrics such as utilization and power consumption. The outputs of these components request logs and hardware telemetry are collected and later combined into a unified dataset for analysis.

This process can be represented conceptually as:

Workload Generator → TorchServe → GPU Execution → Monitoring

The use of an open-loop workload generator is important because it allows queueing effects to emerge naturally from the interaction between workload intensity and service capacity. When the arrival rate exceeds the effective service rate, requests accumulate in the queue and backlog increases. Under lower load, the queue frequently becomes empty, leading to periods of reduced GPU activity between executions.

Because request generation is independent of system state, the observed dynamics reflect the intrinsic behavior of the serving pipeline rather than feedback effects introduced by adaptive workload control. This makes the experimental setup well-suited for analyzing queue buildup, batching behavior, utilization dynamics, and GPU idle behavior.

TorchServe therefore serves a dual role in this thesis. On one hand, it is the serving framework responsible for executing inference requests. On the other hand, it is the mechanism that introduces the queueing and batching dynamics under study. Its internal queue creates separation between request arrival and execution, leading to backlog formation under load and reduced GPU activity under low demand. Combined with batching, this produces the bursty temporal behavior analyzed throughout the remainder of the thesis.

By constraining the deployment to a single worker and a single GPU, the system retains these essential dynamics while avoiding unnecessary concurrency effects. This makes it possible to study GPU behavior as a function of workload intensity and queue dynamics in a controlled and interpretable manner.

4. Experimental System Architecture

4.1 System Overview

This chapter describes the experimental infrastructure and data collection methodology used throughout the thesis. The focus is on system setup, request generation, telemetry collection, and temporal synchronization. Detailed workload modeling is presented in Chapter 5, while dataset construction and feature extraction are described in Chapter 6.

The methodology is built around three interacting components:

- an inference service that executes requests on a GPU,
- a workload generator that produces controlled request traffic,
- and a monitoring subsystem that records GPU telemetry.

These components operate concurrently and are coordinated through a shared notion of time. The inference service receives requests over HTTP and processes them using a GPU-backed model. The workload generator acts as a client, issuing requests according to a configurable arrival process. In parallel, the monitoring subsystem samples GPU metrics such as power consumption and utilization.

Conceptually, the overall pipeline can be summarized as:

Workload Generator → Inference Service → Telemetry Collection → Windowed Dataset

The architecture is intentionally simplified to isolate the core phenomena of interest. By restricting the system to a single inference service and a single GPU, the methodology avoids confounding effects such as multi-tenant interference or distributed scheduling, while still preserving realistic inference behavior such as asynchronous request arrivals, queueing, and GPU execution.

Several stages of the request lifecycle can be inferred through externally observable timestamps and telemetry measurements, enabling analysis of workload dynamics without direct instrumentation of TorchServe internals.

4.2 Window-Based Metric Aggregation

A key challenge in the methodology is temporal alignment. Different sources of information are generated independently and at different rates. Request arrivals and completions occur asynchronously as discrete events, while GPU telemetry such as power and utilization is sampled periodically by NVML. As a result, these measurements are not naturally synchronized and cannot be compared directly on a sample-by-sample basis.

To address this problem, the continuous execution timeline is divided into fixed-duration intervals of length ΔT(e.g., 50 ms). Each interval, referred to as a window, groups together all events and telemetry samples that occur within the same time boundaries. Instead of analyzing individual events independently, the system is analyzed at the level of windows, where workload activity and GPU behavior are approximately aligned in time.

Formally, every timestamp $t$is assigned to a window index:

$$\text{window\_index} = \left\lfloor \frac{t - t_0}{\Delta T} \right\rfloor$$

where $t_0$is the start of the experiment.

For each window i, summary statistics are computed, including:

- number of requests sent,
- number of completed requests,
- average GPU power,
- and GPU utilization statistics.

This process transforms asynchronous event streams into a synchronized sequence of feature vectors:

$$X_i = \text{Aggregate}(\text{events in } [t_i, t_i + \Delta T))$$

Window-based aggregation serves two purposes. First, it aligns heterogeneous measurements originating from different sources and sampling frequencies. Second, it produces a structured time-series representation suitable for statistical analysis and machine learning.

Each window therefore represents a short snapshot of system behavior, capturing both workload activity and GPU state over the same time interval.

4.3 Hardware and Software Environment

The experimental platform is a single-node GPU inference testbed deployed on the University of Cyprus (UCY) High-Performance Computing (HPC) cluster. The use of an HPC environment ensures controlled resource allocation, stable execution conditions, and reproducibility across experimental runs.

The primary computation is performed on an NVIDIA Tesla V100 GPU (16 GB), which executes the forward pass of the neural network during inference. The GPU is responsible for the tensor operations that dominate deep learning execution.

The CPU (Intel Xeon, 16 cores) performs orchestration tasks rather than the core inference computation. Specifically, the CPU is responsible for:

- running the TorchServe process,
- managing the internal request queue,
- performing preprocessing and postprocessing operations,
- and coordinating data transfers between host memory and GPU memory.

This separation of responsibilities is important because CPU scheduling and contention can indirectly influence GPU utilization and request latency.

To minimize interference between system components, CPU affinity is explicitly controlled. TorchServe, the workload generator, and the monitoring subsystem are pinned to separate CPU cores using tools such as `taskset`. This reduces contention for CPU resources and improves timing stability during experiments.

An alternative design would be to execute the workload generator on a separate machine. However, this introduces an additional challenge: accurate clock synchronization across machines. Since the methodology relies heavily on precise temporal alignment between request events and GPU telemetry, even small clock offsets or network timing variations could distort latency measurements and window alignment. Running all components on the same node avoids these synchronization issues while still allowing CPU isolation through affinity control.

The system is equipped with 32 GB of RAM, which is sufficient to hold the model, intermediate data, and logging buffers without introducing paging effects.

The software stack includes Python 3.11, PyTorch 2.5.1, torchvision 0.20.1, CUDA 12.1, and TorchServe.

Threading behavior is explicitly controlled using:

OMP_NUM_THREADS=1

This prevents implicit CPU parallelism from introducing timing variability during preprocessing and tensor operations.

To ensure reproducibility, all dependencies are installed in isolated environments with fixed versions, and the TorchServe configuration remains constant across experimental runs.

4.4 TorchServe Server Configuration

The inference server is configured as a single-worker, single-GPU service in order to provide a simplified and analytically tractable execution environment.

Specifically, the configuration sets:

```
number_of_gpu=1
default_workers_per_model=1
```

This ensures that all requests are handled by a single worker process executing on a single GPU device.

Under this configuration, requests first enter an internal queue and are then processed by the worker. The worker may execute requests individually or in small batches, depending on TorchServe's batching mechanism. Since only one worker exists, there is no application-level parallelism between requests. The system therefore behaves approximately as a single-server queueing system.

All service endpoints are bound to local interfaces (`127.0.0.1`) to eliminate network variability such as packet delays or routing overhead.

Under these conditions, the system can be approximated using classical queueing concepts, where incoming requests arrive according to a stochastic process and the worker acts as the server responsible for GPU execution.

4.5 Workload Generator Design

The workload generator is responsible for producing inference requests according to a controlled statistical process. Its purpose is to emulate realistic client behavior while maintaining precise control over request timing.

The generator performs three main tasks:

- sampling inter-arrival times,
- sending HTTP requests to the inference server,
- and recording request timestamps.

Request arrivals are modeled as a Poisson process. In practice, this is implemented by sampling inter-arrival times from an exponential distribution:

$$\Delta t_i \sim \text{Exponential}(\lambda)$$

where $\lambda$is the target request rate in requests per second.

Listing 4.1 shows the core request-generation loop.

```
sample = random.choice(samples)

send_ts = time.perf_counter()
sent_log.write(f"{send_ts}\n")
sent_log.flush()

response = send_request(url, model_name, sample)

completion_log.write(f"{response.timestamp}\n")
completion_log.flush()
```

A random input sample is selected to avoid caching effects and maintain variability in the workload. The request timestamp is recorded using a high-resolution timer and immediately flushed to disk to preserve temporal accuracy.

The request is then issued to TorchServe through an HTTP call, triggering the full inference pipeline, including queueing, preprocessing, GPU execution, and response generation. Upon completion, the response timestamp is logged, enabling end-to-end latency computation.

Importantly, the workload generator operates as an open-loop system. Request generation follows the predefined arrival process and does not adapt to observed latency or system state. This design allows queueing effects and backlog formation to emerge naturally from the interaction between workload intensity and service capacity.

4.6 Request Logging Infrastracure

The request logging infrastructure records the lifecycle of each inference request. Logging is performed entirely on the client side by the workload generator, ensuring that the inference server itself remains unmodified.

Each request is recorded at two points in time:

- when the request is issued,
- and when the response is received.

For each request i, we define:

$$t_i^{send}$$

as the request submission timestamp, and

$$t_i^{comp}$$

as the completion timestamp.

The end-to-end latency of request $i$is then computed as:

$$\text{latency}_i = t_i^{comp} - t_i^{send}$$

This latency includes queueing delay, preprocessing, GPU execution, and response transmission.

The logging infrastructure observes the system externally through request timing rather than through internal instrumentation. This aligns with the broader objective of inferring system behavior using externally observable signals.

The resulting logs can later be aligned with GPU telemetry and aggregated into fixed-duration windows for dataset construction.

4.7 GPU Monitoring and Energy Measurement

GPU telemetry is collected concurrently with workload execution using a dedicated monitoring process that operates independently of both the workload generator and the inference server.

Monitoring is based on the NVIDIA Management Library (NVML) and the Zeus energy monitoring framework. NVML provides access to low-level GPU metrics such as utilization, memory usage, temperature, and instantaneous power measurements. Zeus is an open-source energy monitoring framework built on top of NVML and designed for deep learning workloads.

NVML provides instantaneous power readings and, on some GPU architectures, cumulative energy counters. However, support for direct energy reporting depends on the GPU model and driver version. To provide a consistent measurement methodology, this work estimates energy consumption by periodically sampling NVML power readings and integrating them over time.

If power samples are denoted by:

$$P(t_1), P(t_2), \ldots, P(t_n)$$

then energy over an interval is approximated as:

$$E \approx \sum_{k=1}^{n} P(t_k)\Delta t_k$$

where Δt_krepresents the interval between samples.

GPU activity is characterized using the NVML API call:

```
nvmlDeviceGetUtilizationRates()
```

which reports the percentage of time the GPU was actively executing kernels during a recent driver-maintained sampling interval. According to NVIDIA documentation, this interval is relatively coarse-grained (typically hundreds of milliseconds), meaning that very short execution bursts may not be fully visible.

Raw telemetry samples are aggregated into fixed-duration windows to align them with workload events. For each window, the monitoring subsystem computes:

- average GPU power,
- estimated energy consumption,
- mean GPU utilization,
- maximum utilization,
- utilization variance,
- and a binary GPU activity indicator.

These aggregated metrics form the telemetry component of the final dataset used in later analysis.

5. Workload Modeling and Traffic Generation

5.1 Microservice Traffic Characteristics

Inference workloads in microservice-based machine learning systems differ from traditional batch-oriented workloads. Instead of processing large, coordinated jobs, an inference service receives many small requests from independent clients. Each request is handled separately, and requests arrive at unpredictable times.

This means that the workload is inherently irregular. Even if the average request rate remains stable, the exact timing of arrivals changes from moment to moment. In practice, this leads to three common patterns:

- periods where many requests arrive close together (bursts),
- short intervals where no requests arrive (idle gaps),
- occasional sudden increases in demand (spikes).

These patterns are important because they directly affect how the GPU is used. For example, when several requests arrive close together, the system may process them as a batch, improving efficiency. In contrast, when there are gaps between requests, the GPU may remain idle while still consuming power.

5.2 Poisson Arrival Process

To emulate realistic traffic, request arrivals are modeled as a Poisson process with rate λ, measured in requests per second (RPS). This means that requests arrive independently over time, without coordination. A key property of a Poisson process is that the time between consecutive arrivals (called the inter-arrival time) follows an exponential distribution. Instead of directly deciding how many requests occur in a time interval, we generate the workload by sampling these inter-arrival times.

In practice, this works as follows. For each request, we draw a random delay Δt from an exponential distribution with parameter λ, and wait for that duration before sending the next request. This process is repeated continuously. Exponential sampling refers to drawing random values from this distribution. In the implementation, this is done using Python's standard library:

- `random.expovariate(λ)`

This function returns a random delay whose expected value is:

$$E[\Delta t] = \frac{1}{\lambda}$$

As a result:

- higher $\lambda \rightarrow$ shorter delays $\rightarrow$ higher request rate
- lower $\lambda \rightarrow$ longer delays $\rightarrow$ lower request rate

An important property of the exponential distribution is memorylessness. This means that the time until the next request does not depend on when the last request occurred. In other words, arrivals are completely independent over time. This avoids artificial patterns such as periodic bursts and ensures that the generated workload reflects realistic, unsynchronized client behavior. By repeatedly sampling inter-arrival times and issuing requests accordingly, the system produces a stochastic but controlled request stream that closely follows the desired Poisson process.

5.3 Workload Phase Design

To systematically study system behavior, the workload is organized into a sequence of phases. Each phase is defined by a fixed request rate (RPS) and a fixed duration. This structure allows controlled exploration of different load conditions while keeping the statistical properties of the workload stable within each phase.

Each phase is specified in a configuration file with parameters such as the phase name, duration, and target request rate. For example:

Listing 5.3: Example workload phase configuration (phases_poisson.json)

```
{ "name": "poisson-2min-20rps", "pattern": "poisson", "duration": 120,
"rps": 20 }
```

Each phase represents a steady operating point of the system. By keeping the arrival rate constant during a phase, any observed changes in behavior (e.g., backlog growth or GPU utilization) can be attributed to system dynamics rather than changes in input traffic.

The workload generator operates in an open-loop manner. This means that requests are generated according to the predefined arrival process (Poisson), independently of how fast the system processes them. In other words, new requests continue to be issued even if previous ones are still being processed.

This design is important because it allows queueing effects to emerge naturally. Under high load, if the system cannot keep up with incoming requests, a backlog will form. In contrast, a closed-loop system (where each new request waits for a previous response) would artificially limit the load and hide saturation effects.

5.4 Request Rate Sweep Experiments

T o explore the full range of system behavior, the experimental protocol performs a systematic sweep over request rates, ranging from very low load (e.g., 1 RPS) to high load (e.g., 80 RPS). Each phase runs for a fixed duration (typically 120 seconds), which is sufficient for the system to reach steady-state behavior.

Phases are executed sequentially without artificial pauses. Instead of explicitly resetting the system between phases, we rely on the duration of each phase to absorb transient effects from the previous one.

During the experiment, two types of data are collected in parallel:

- the workload generator records the timestamp at which each request is sent,
- the monitoring subsystem records GPU telemetry (e.g., power and utilization).

5.5 Underload, Optimal Load, and Saturation Regions

As the request rate increases during the rate sweep, the system transitions through three distinct operating regimes. Rather than restating general queueing theory, we describe these regimes in terms of how they manifest in our GPU-based inference system.

At low request rates, the system has more processing capacity than incoming demand. In this regime, requests are typically executed immediately after arrival, and queueing is negligible. From a GPU perspective, this results in intermittent execution. The GPU processes short bursts of work when requests arrive, followed by idle periods where no computation is performed. These idle gaps are clearly visible in the telemetry as drops in utilization. A key observation is that, despite low utilization, power consumption remains relatively high due to the GPU's baseline (static) power. As a result, energy efficiency is poor: a significant amount of energy is consumed during idle periods where no useful work is performed.

As the request rate increases, the system reaches a regime where the GPU is kept busy most of the time, but without sustained queue buildup. In this region, arrivals are frequent enough to reduce idle gaps, allowing the GPU to operate more continuously. At the same time, the system is still able to process requests at approximately the rate they arrive, so latency remains stable and backlog remains small. From an experimental standpoint, this is the most efficient operating region, where the GPU performs useful work most of the time without being overloaded.

At high request rates, the system can no longer keep up with incoming requests. As a result, requests begin to accumulate in the queue. In this regime, the GPU is continuously active, with utilization remaining close to its maximum. However, this does not translate to better performance. Instead this highlights an important system-level effect: even though the GPU is fully utilized, system performance degrades because requests spend more time waiting in the queue.

6. Dataset Construction

6.1 Request Logs and Latency Computation

The workload generator produces two primary log files, *sent.log* and *completions.log*, each containing high-resolution timestamps recorded using `time.perf_counter()`. These logs capture the lifecycle of each inference request, from submission to completion, with minimal measurement overhead.

The generator operates in an **open-loop** manner. Requests are issued according to the Poisson arrival process, independently of whether previous requests have completed. This means that new requests continue to be generated even when the system is under load, allowing queueing effects and backlog growth to emerge naturally.

Each request is associated with two timestamps:

- t_i^{send}: the time at which the request is issued by the client,
- t_i^{comp}: the time at which the response is received.

After each experimental run, the two logs are merged into a unified dataset (*request_log.csv*), where each row corresponds to a single request:

$$\left(\text{request\_id},\ t_i^{\text{send}},\ t_i^{\text{comp}}\right)$$

The merge is performed sequentially, pairing the i-th send event with the i-th completion event. This is valid because the workload generator issues requests in order and each request produces exactly one response.

Latency is then computed as:

$$\text{latency}_i = t_i^{\text{comp}} - t_i^{\text{send}}$$

Request timestamps and GPU telemetry are measured on the same time base. In this system, both the workload generator and the monitoring process run on the same machine and use the same high-resolution clock (`time.perf_counter()`).

6.2 Window-Level Feature Extraction

Request logs and GPU telemetry are recorded at different times and at different rates. To analyze them together, we group all events into fixed-duration time windows (50 ms by default).

Each event timestamp is assigned to a window based on when it occurs. All events that fall within the same time interval are grouped together.

For each window i, we compute:

- r_i^{sent}: number of requests sent in the window,
- r_i^{comp}: number of requests completed in the window.

This process converts asynchronous event streams into a discrete-time representation, where each window summarizes system activity over a short interval. The purpose of windowing is to align request activity with GPU telemetry. Once grouped into windows, both workload metrics and hardware measurements can be analyzed on the same time axis.

6.3 GPU Energy Metrics

GPU energy consumption is measured using the Zeus monitoring library, which relies on NVIDIA's NVML API to obtain power readings at millisecond-level granularity. During execution, the monitoring process periodically samples the GPU's instantaneous power $P(t)$. Energy consumption within each time window is then computed by aggregating these samples. In practice, this is done using a discrete approximation:

$$E_i \approx \sum_j P_j \cdot \Delta t_j$$

where P_j are the sampled power values and Δt_j is the time between samples.

From these measurements, two metrics are derived for each window:

- total energy consumed within the window,
- average power over the window duration.

These metrics provide a view of how much energy the GPU consumes over time and how that relates to workload activity. By computing energy at the same window granularity as request activity, GPU behavior can be directly compared with workload patterns. In particular, this allows identification of periods where the GPU consumes energy without performing useful work, which is central to the analysis of idleness.

6.4 GPU Utilization Metrics

GPU utilization is sampled using NVIDIA's NVML API at a high frequency (approximately every 2–5 ms). Each sample represents the instantaneous percentage of time the GPU was active during the preceding interval.

To make these measurements comparable with request activity, utilization samples are not analyzed individually. Instead, they are grouped into fixed-duration time windows of length ΔT(50 ms in our experiments). A window is therefore a short time interval (e.g., 50 ms) that contains multiple utilization samples. Given a sampling period of 2–5 ms, each window typically contains approximately 10–25 samples. Aggregating multiple samples within a window serves two purposes. First, it reduces noise inherent in instantaneous measurements. Second, it enables alignment with other signals (such as request arrivals and completions), which are also aggregated at the same temporal granularity.

For each window i, we compute summary statistics over all utilization samples $U(t)$that fall within that interval:

- U_i^{mean}: average utilization across the window, representing sustained GPU activity
- U_i^{max}: maximum utilization observed, capturing short bursts of execution
- U_i^{std}: standard deviation, reflecting variability within the window

In addition, we define a binary activity indicator:

$$\text{gpu\_active\_flag}_i = \begin{cases} 1, & \text{if } U(t) > \theta \text{ for any sample in the window} \\ 0, & \text{otherwise} \end{cases}$$

where θ is a small threshold used to filter out measurement noise.

This indicator provides a coarse signal of whether any GPU activity occurred during the window. It is particularly useful for distinguishing fully idle periods from windows that contain brief execution bursts.

6.5 Idle Window Labeling

To analyze GPU behavior, each time window is classified as either idle or active based on observed system activity. In this work, idleness is defined using completed requests within each window. Specifically, a window is considered idle if no inference requests are completed during that interval:

$$\text{idle}_i = \begin{cases} 1, & \text{if } r_i^{\text{comp}} = 0 \\ 0, & \text{otherwise} \end{cases}$$

where r_i^{comp} denotes the number of requests completed in window i.

This definition directly reflects application-level behavior. If no requests complete within a time window, then the GPU is not performing useful inference work during that period, regardless of its instantaneous utilization or power consumption. Using completed requests rather than arrivals is important. A request may arrive in a given window but be executed later due to queueing delays. Therefore, completion events provide a more accurate indication of when the GPU is actively performing inference. This binary labeling provides a simple and interpretable way to identify idle periods. It is used throughout the analysis to quantify how often the GPU remains inactive and how idleness varies across different workload conditions.

6.6 Queue Backlog Estimation

To capture system pressure, we estimate the queue backlog as the cumulative difference between arrivals and completions:

$$\text{backlog}_i = \sum_{k=0}^{i} (r_k^{\text{sent}} - r_k^{\text{comp}})$$

In practice, this is computed efficiently using cumulative sums:

Listing 6.3: analyze_idle.py def main

```
df["estimated_backlog"] = df["queue_delta"].cumsum()
df["queue_delta"] = df["requests_sent"] - df["completed_requests"]
```

Backlog serves as a key state variable, encoding the history of the system. It reflects whether the system has pending work and therefore plays a central role in both analysis and prediction. Unlike instantaneous features, backlog captures temporal dependencies and provides a stable signal of system state.

6.7 Final Dataset Structure

The final dataset, stored as `windows_<model>.csv`, contains one row per observation window. Each row aggregates workload, queue state, and GPU telemetry into a unified representation suitable for analysis and modeling.

Table 6.1: Final Dataset Schema

Column	Description
window_index	Window identifier
phase	Workload regime (underload, optimal, saturation)
rps	Target request rate
requests_sent	Number of requests issued
completed_requests	Number of completed requests
backlog	Estimated queue size
energy_j	Energy consumed (Joules)
avg_power_w	Average power (Watts)
gpu_util_mean	Mean GPU utilization (%)
gpu_util_std	Utilization standard deviation
gpu_util_max	Peak utilization (%)
gpu_active_flag	Binary GPU activity indicator
request_idle	Primary idleness label
energy_idle	Secondary idleness label

The dataset is constructed by aligning asynchronous request logs with GPU telemetry over fixed-duration windows. Derived features such as backlog encode temporal dependencies by capturing the cumulative imbalance between arrivals and completions. This representation enables analysis of both system-level dynamics and hardware behavior under partial observability.

7 Exploratory Data Analysis

7.1 System Model and Observability Constraints

We model the inference system as a single logical service driven by stochastic request arrivals and executed on a GPU. Under the configuration used in this work a single worker bound to a single GPU the system can be approximated as a single-server queue. However, unlike classical queueing systems, service is not strictly sequential. Instead, requests may be processed in groups, leading to state-dependent and non-uniform service behavior.

Requests arrive according to a Poisson process with rate λ, implying independent and memoryless inter-arrival times. The service process is characterized by an effective rate μ, representing the average processing capacity of the GPU. In contrast to classical models, $\mu$is not constant in practice. As the number of queued requests increases, more opportunities for grouping requests arise, which can improve throughput. As a result, the effective service rate depends on system state.

System behavior is observed through fixed-duration time windows, producing aggregated measurements of the form:

$$X_i = [s_i, c_i, b_i, P_i, E_i, U_i]$$

where s_i and c_i denote the number of arrivals and completions, b_i is the backlog, and P_i, E_i, and U_i represent power, energy, and utilization measurements, respectively.

A key constraint of this study is partial observability. Internal GPU execution details, such as kernel scheduling and grouping decisions, are not directly accessible. Instead, all observations are derived from external request logs and aggregated telemetry (via NVML and Zeus). Consequently, system behavior must be inferred indirectly from these coarse-grained signals.

We therefore consider a latent system state z_i, indicating whether the GPU is idle or active within a given window. This state is not directly observed; instead, we observe:

$$X_i \sim P(X_i \mid z_i)$$

meaning that the measured features are generated from a distribution conditioned on the underlying system state.

Among the observable signals, backlog plays a central role. It evolves according to:

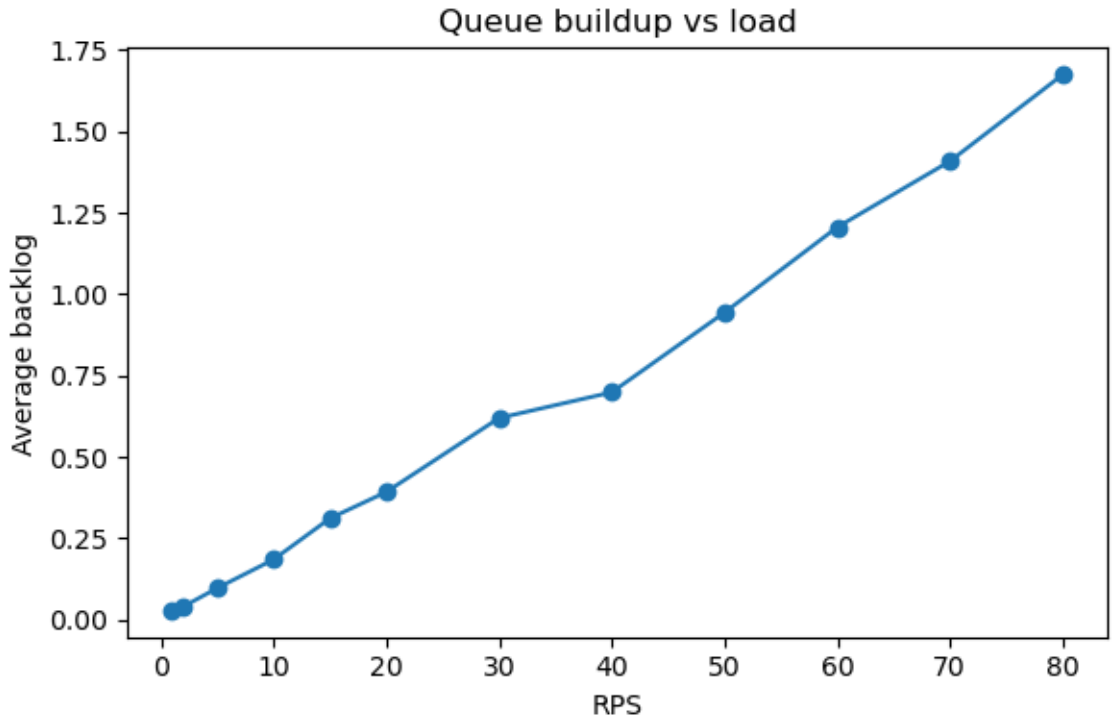
$$b_i = b_{i-1} + s_i - c_i$$

and captures the cumulative imbalance between incoming and completed work. Unlike instantaneous measurements such as arrivals or completions, backlog is stateful and encodes information about recent system history. This makes it a robust indicator of system pressure and operating conditions.

7.2 Queue Dynamics and Load Regimes

Figure 7.1 shows the relationship between request rate and average backlog. As the request rate increases, backlog grows monotonically from near-zero values at low load to progressively higher values at higher request rates. At very low rates, backlog remains close to zero, indicating that requests are processed almost immediately. As load increases, backlog becomes consistently non-zero and continues to rise.

Figure 7.1: Queue buildup vs load



This trend indicates that backlog is strongly correlated with workload intensity and provides a clear signal of system pressure. Importantly, the increase is smooth rather than abrupt. Even at the highest request rates evaluated in this study, backlog grows gradually and remains bounded within the observation interval.

This behavior differs from the sharp queue growth often associated with idealized M/M/1 systems operating near saturation. However, this comparison should be interpreted cautiously. The experiments do not directly establish the true maximum processing capacity of the system, and the evaluated workload range may not fully reach the saturation point predicted by classical queueing models. In addition, the assumptions underlying M/M/1 models do not fully match GPU inference systems, where service behavior is influenced by batching and state-dependent execution effects.

Several factors likely contribute to the observed gradual backlog growth.

First, the effective service rate is not strictly constant. As backlog increases, more requests become available to be processed together, improving throughput through grouped execution. This creates state-dependent service behavior, where processing efficiency can increase with queue size.

Second, service times exhibit lower variability than exponential service assumptions typically used in classical queueing models. GPU inference execution consists of structured tensor operations with relatively stable execution time, especially for homogeneous inputs.

Third, the use of fixed-duration aggregation windows smooths short-term fluctuations in arrivals and completions, reducing apparent variability in backlog measurements.

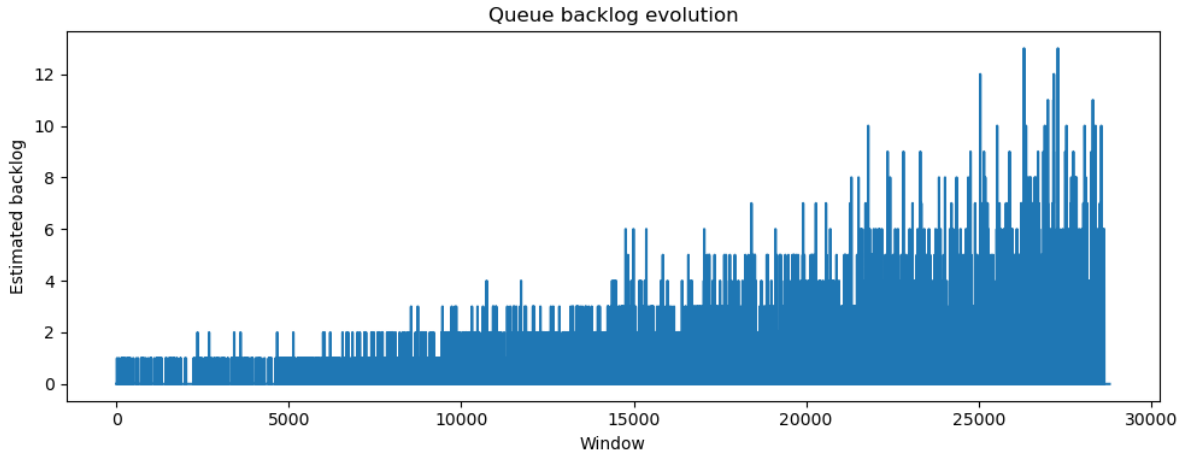
Backlog can therefore be interpreted as a cumulative measure of imbalance between arrivals and completions. When incoming requests exceed processing capacity, backlog increases; when processing catches up, backlog decreases. Because it accumulates information over time, backlog provides a stable and robust indicator of system state.

These results show that backlog is more informative than instantaneous metrics such as utilization alone. Near-zero backlog indicates underload, bounded backlog indicates balanced operation, and sustained backlog growth indicates increasing system pressure.

7.3 Temporal Dynamics and Burst Behavior

Figure 7.2 shows the evolution of backlog over time for a representative experiment. Instead of smooth changes, backlog exhibits a characteristic step-like pattern: periods of gradual increase are followed by abrupt drops. As load increases, both the magnitude and frequency of these fluctuations grow, indicating increasingly dynamic system behavior.

Figure 7.2: Backlog evolution over time



These patterns reveal that the system does not process requests continuously. During certain intervals, backlog increases steadily, indicating that requests are arriving faster than they are being completed. These periods are followed by sharp decreases in backlog, where multiple requests complete within a short interval. This alternating pattern suggests bursty execution behavior rather than constant-rate service.

The observed burstiness cannot be explained solely by the arrival process. Since arrivals follow a Poisson process, they are stochastic but do not inherently produce the regular drop patterns observed in the backlog traces. The completion patterns are instead consistent with grouped or batched execution on the GPU. When multiple requests are processed together, they generate bursts of completions that appear as sudden decreases in backlog.

Backlog evolution can therefore be interpreted as the result of continuous arrivals interacting with discrete execution phases. Requests arrive asynchronously and increase backlog, while grouped execution removes multiple requests over short periods.

An important observation is that backlog may continue growing over longer experiment durations if the average arrival rate persistently exceeds the effective service rate. Therefore, the bounded backlog observed in these experiments should be interpreted relative to the finite observation interval rather than as proof of long-term stability.

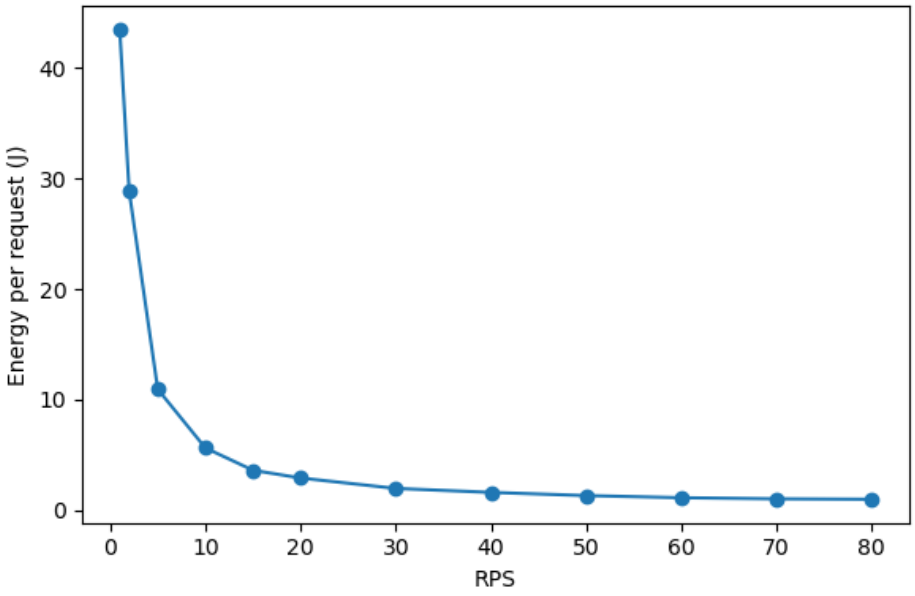
These temporal dynamics highlight an important limitation of aggregate metrics. Average values obscure the underlying execution pattern and fail to capture the bursty nature of service. Time-series analysis instead reveals the interaction between arrivals and execution behavior, providing insight into how work is scheduled and processed.

This behavior also has direct implications for GPU idleness. Periods of backlog accumulation correspond to intervals where requests are waiting in the queue, while sudden backlog reductions correspond to active execution phases. Between these execution bursts, the GPU may experience short idle intervals, particularly under lower load conditions where backlog frequently returns to zero.

7.4 Energy Efficiency and Energy Proportionality

Figure 7.3 shows the energy consumed per request as a function of the request rate. The results exhibit a strong non-linear trend. At low request rates, energy per request is extremely high. As the request rate increases, energy per request decreases rapidly and eventually stabilizes at higher load levels.

Figure 7.3: Energy per request as a function of request rate



This behavior indicates that the system is not energy proportional. In an energy-proportional system, power consumption scales with workload, so that energy per unit of work remains roughly constant across load levels. In contrast, the measured results show that energy per request varies significantly with load, especially in the low-load regime.

The root cause is the presence of a substantial baseline power consumption. Even when the GPU is idle or lightly utilized, it continues to draw a significant amount of power. This is consistent with the measured power behavior (Section 7.5), where power remains relatively high even at low request rates. As a result, when only a small number of requests are processed, the energy consumed during idle periods dominates the total cost per request. As the request rate increases, this inefficiency is reduced. The GPU spends less time idle and more time performing useful computation. Consequently, the fixed baseline power is amortized over a larger number of completed requests, leading to a rapid decrease in energy per request. This transition is clearly visible in Figure 7.3.

At higher request rates, energy per request stabilizes. In this regime, the GPU operates close to continuous utilization, and most of the consumed energy corresponds to actual computation rather than idle overhead. The system therefore reaches a steady efficiency level determined by the intrinsic execution cost of the workload. Batching plays a key role in this transition. As backlog increases, more requests are available to be processed together, leading to higher effective throughput. This increases hardware utilization and reduces the fraction of time spent idle. While batching does not reduce baseline power, it improves energy efficiency by increasing the amount of useful work performed per unit time. This behavior differs significantly from CPU-based systems. Modern CPUs can enter deep low-power states during idle periods, significantly reducing baseline power consumption. As a result, CPUs exhibit closer-to-ideal energy proportionality. GPUs, in contrast, maintain active memory subsystems and execution contexts even when idle, resulting in a persistent power floor and reduced energy proportionality.

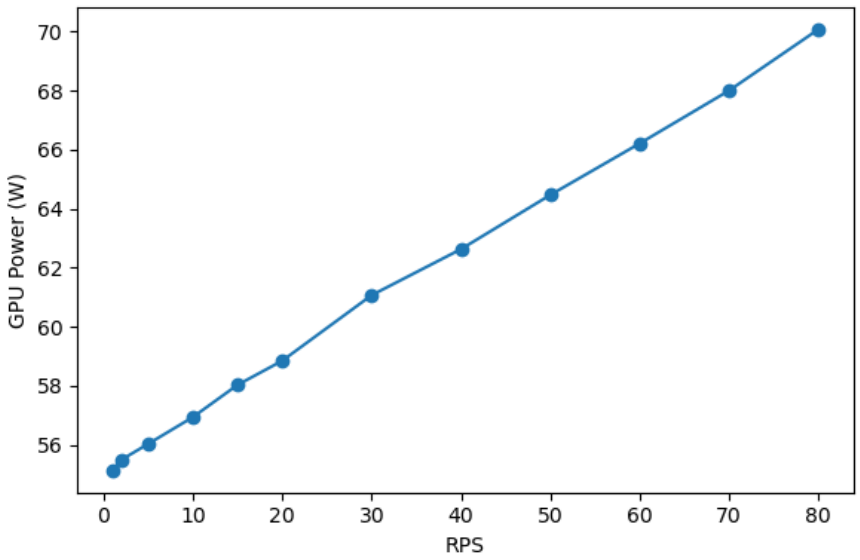
In addition to backlog and energy trends, the experiments also show a clear relationship between workload intensity and GPU idleness. At low request rates, many aggregation windows contain no completed requests and near-zero GPU utilization, indicating frequent idle intervals. As request rate increases, these idle windows become progressively less frequent because the GPU receives work more continuously. This transition explains why energy efficiency improves with load: higher request rates reduce the fraction of time spent idle while maintaining similar baseline power consumption.

Overall, the experimental results demonstrate that energy inefficiency in GPU inference systems is primarily driven by idle time rather than computational inefficiency. Improving efficiency therefore requires reducing idle intervals, either by increasing load or by grouping work more effectively.

7.5 GPU Power Behavior and Limitations

Figure 7.4 presents the measured GPU power consumption as a function of the request rate. Contrary to the expectation that power should scale strongly with workload intensity, the results show only a modest increase in power across the entire range of request rates. While throughput and backlog change significantly with load (Sections 7.2–7.3), power increases gradually from approximately 55 W to 70 W, indicating limited sensitivity to workload.

Figure 7.4: GPU Power vs Request Rate



This empirical observation confirms that GPU power is dominated by a substantial baseline component. Even at very low request rates, where the GPU is frequently idle, power consumption remains high. As load increases and the GPU becomes more active, the additional power drawn by computation is relatively small compared to this baseline. As a result, the difference between idle and active power is much smaller than expected.

This behavior directly explains the lack of energy proportionality observed in Section 7.4. In an energy-proportional system, power would decrease significantly during idle periods and increase proportionally with utilization. Instead, the GPU maintains a persistent power floor, largely independent of workload intensity. This is in contrast to modern CPUs, which employ aggressive low-power states and can significantly reduce power consumption when idle.

Two factors contribute to this weak dependence of power on load. First, architectural components such as memory subsystems, execution contexts, and communication interfaces remain active even in the absence of computation, sustaining a high baseline power. Second,

the temporal structure of GPU execution further reduces observable variation. As shown in Section 7.3, requests are processed in short, batched bursts. These bursts are brief relative to the observation window and are averaged together with idle intervals, smoothing out short-term fluctuations in power.

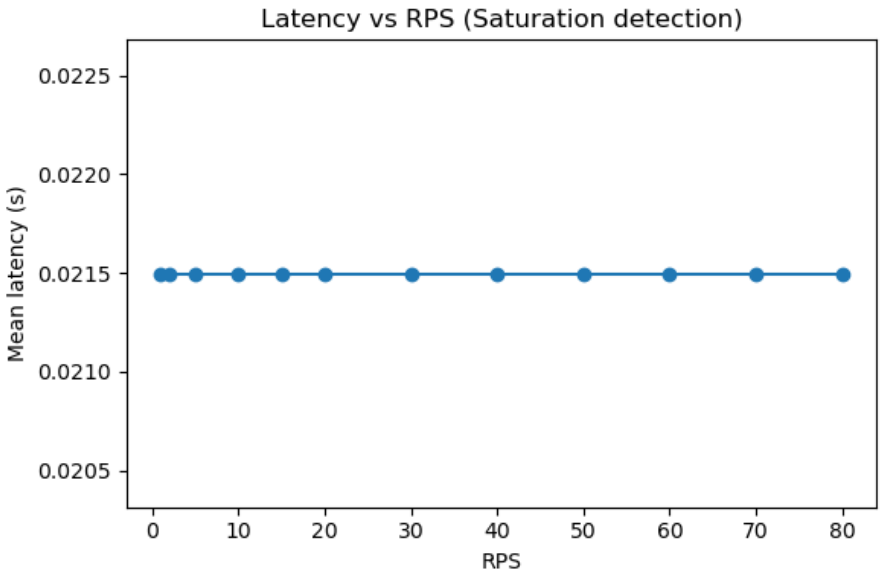
An important consequence of this behavior is that power alone is a poor indicator of system state. Windows with similar average power may correspond to very different execution conditions, including windows with minimal GPU activity and windows containing short bursts of active execution. Because the GPU maintains a relatively high baseline power even during idle periods, average power alone cannot reliably distinguish active computation from short idle intervals. This limits the usefulness of power as a standalone signal for detecting idleness or system load.

Instead, power must be interpreted in conjunction with workload-driven metrics such as backlog and request rates. While power provides useful context about overall energy consumption, it does not capture the temporal dynamics of execution. The results therefore reinforce a central conclusion of this work: energy inefficiency in GPU inference systems is primarily driven by idle time at high baseline power, rather than by large variations in power during computation.

7.6 Latency Behavior and Hidden Saturation

Figure 7.5 shows the mean request latency as a function of the request rate. Contrary to classical queueing expectations, latency remains nearly constant across the evaluated range of request rates, even as backlog increases significantly (Section 7.2). The measured latency remains approximately 21–22 ms with relatively small variation across workloads.

Figure 7.5: Latency vs Request Rate



At first glance, this suggests that the system maintains stable performance under increasing load. However, this interpretation is incomplete. The relatively flat latency curve occurs simultaneously with growing backlog, indicating that the system is experiencing increasing pressure even though latency measurements remain stable.

One reason for this behavior is related to how latency is observed during finite-duration experiments. Latency can only be measured for requests that complete within the observation interval. As workload intensity increases and backlog accumulates, a larger fraction of requests remains queued at the end of the experiment and therefore does not contribute to the measured latency statistics. This introduces an observational bias toward requests that completed earlier or experienced shorter waiting times. Consequently, the measured latency may underestimate the true delay experienced by all issued requests under higher load conditions.

Latency is also inherently a lagging indicator. Backlog responds immediately when arrivals exceed completions, because queued requests accumulate as soon as processing capacity becomes insufficient. In contrast, latency is only observable after requests complete execution. This delayed visibility obscures the onset of congestion and makes latency slower to reflect changes in system pressure.

Batching further contributes to this effect. As backlog increases, more requests become available for grouped execution, improving GPU throughput efficiency. Although requests may spend longer waiting in the queue before execution begins, grouped execution can reduce the processing overhead per request once execution occurs. This partially offsets the effect of increased queueing delay in the measured latency of completed requests.

Together, these effects explain why latency remains relatively stable despite increasing workload intensity. The system enters a regime where backlog steadily increases while latency measurements show only limited change. This can be interpreted as an early-stage saturation effect in which congestion becomes visible first through queue growth rather than through latency escalation.

These results demonstrate that latency alone is insufficient for characterizing the operating state of GPU inference systems. Accurate interpretation requires combining latency measurements with workload-driven metrics such as backlog, request rate, and GPU activity. Among these signals, backlog provides a more immediate and robust indicator of increasing system pressure because it directly captures the imbalance between arrivals and completions.

8. Conclucion, Discussion, Implications, and Future Work

8.1 What the Experiments Actually Show

The experimental analysis in Chapter 7 reveals three key findings.

First, GPU idleness is directly tied to queue dynamics. Idle windows occur when the system has no backlog and no incoming requests within a window. This condition is observable through simple features such as request counts and backlog evolution. The results show that these signals are sufficient to characterize system state, despite the lack of direct access to GPU execution.

Second, classical indicators fail to capture system behavior. In particular, latency remains nearly constant across increasing load (Section 7.6), even when backlog grows significantly. Similarly, GPU power varies only modestly with workload (Section 7.5). These results demonstrate that commonly used metrics do not provide reliable visibility into system state.

Third, batching fundamentally alters service behavior. The backlog time series (Section 7.3) shows clear burst patterns, with gradual accumulation followed by sharp drops. This confirms that requests are processed in batches rather than continuously. As a result, the effective service rate is not constant but depends on the availability of work, explaining the deviation from classical M/M/1 behavior observed in Section 7.2.

8.2 Interpreting GPU Idleness

The results indicate that GPU idleness is not a random or purely hardware-level phenomenon. Instead, it emerges from the interaction between stochastic arrivals and the temporal dynamics of request execution. At low load, the system frequently empties, leading to repeated idle windows. At higher load, backlog persists and idle periods become less frequent.

Even at moderate load levels, short idle windows may still occur between execution phases. The observed bursty completion patterns and backlog fluctuations are consistent with grouped or batched execution behavior within the inference pipeline. However, the current study does not explicitly vary batching parameters, and therefore the precise contribution of batching to idleness cannot be isolated experimentally.

An important observation is that idleness is not visible through power measurements alone. As shown in Section 7.5, power remains close to a baseline level even during periods of low

activity. This means that idle windows correspond to continued energy consumption despite limited useful computation.

GPU idleness should therefore be understood as a temporal inefficiency: periods where the system has available processing capability but is not actively executing inference work. This inefficiency arises primarily from the temporal mismatch between request arrivals and execution activity rather than from inefficient computation itself.

8.3 Evidence of GPU Idleness

The experimental results provide multiple indicators of GPU idleness across different workload conditions. First, the fraction of idle windows is highest at low request rates and decreases as workload intensity increases. This indicates that the GPU spends a significant amount of time waiting for work under underloaded conditions.

Second, backlog and idleness exhibit an inverse relationship. When backlog remains close to zero, the system frequently enters idle states because requests are processed faster than they arrive. As backlog increases, idle periods become less frequent because the GPU receives work more continuously.

Third, the temporal traces presented in Section 7.3 reveal clear gaps between execution phases. Backlog increases gradually during accumulation periods and then drops sharply during execution bursts. These transitions indicate alternating periods of waiting and active processing rather than continuous execution.

Finally, GPU power measurements further support this interpretation. As shown in Section 7.5, power consumption remains relatively high even during periods of limited activity. This demonstrates that GPU idleness does not correspond to low energy consumption, but instead represents intervals where energy is consumed without proportional computational work being performed.

Taken together, these observations support the conclusion that GPU idleness is a persistent and measurable property of inference workloads, particularly under low and moderate load conditions.

8.4 Limitations

The conclusions of this work are subject to several limitations. The experimental setup uses a single model on a single GPU. This simplifies analysis but excludes effects such as multi-tenant interference, scheduling contention, and shared-resource behavior. In real systems, these factors may introduce additional variability.

The workload is generated using a Poisson arrival process. While this captures independent arrivals, real workloads may exhibit burstiness, temporal correlations, and non-stationary behavior. These characteristics could affect queue dynamics and observed execution patterns.

The definition of idleness is based on externally observable signals collected through request logs and GPU telemetry. Although these measurements provide useful indicators of system activity, they do not expose low-level GPU scheduling decisions or internal execution states directly. As a result, some hardware-level behavior may not be fully captured.

Another limitation is that batching behavior is inferred indirectly from observable execution patterns rather than controlled experimentally. Although the measured backlog dynamics and bursty completion behavior are consistent with grouped execution, the study does not systematically vary batching parameters such as batch size or batching delay. Consequently, the specific contribution of batching to GPU idleness and queue dynamics cannot be quantified directly.

Finally, all measurements are based on window-level aggregation. This limits temporal resolution and may smooth short-lived events, particularly in power and utilization signals. Very short idle intervals or transient execution events may therefore not be fully visible in the aggregated data.

Despite these limitations, the core observations particularly the relationships between workload intensity, backlog growth, bursty execution behavior, and GPU idleness remain consistent across experiments and are strongly supported by the empirical measurements.

8.5 Future Work

Several directions follow naturally from this study.

A primary extension is online integration. The current analysis is performed offline using collected telemetry and request traces. Integrating the methodology into a live inference

system would enable real-time monitoring, adaptive scheduling, and energy-aware control decisions.

Another important direction is systematic evaluation of batching behavior. Varying parameters such as batch size, batching timeout, and request grouping policies would help isolate the relationship between batching, throughput, latency, backlog formation, and GPU idleness. Such experiments would provide a more precise understanding of how grouped execution affects temporal utilization patterns and energy efficiency.

A further direction is extending the analysis to multi-GPU and distributed environments. Real production systems often involve multiple models, shared accelerators, and distributed scheduling layers. Modeling interactions between multiple queues and shared resources would provide a more realistic view of large-scale inference deployments.

Additional improvements may also be achieved through more advanced temporal modeling techniques. Sequence-based approaches that explicitly incorporate historical context could better capture transition regions and non-stationary workload behavior.

Finally, there is strong potential for energy-aware scheduling policies that directly leverage idleness estimation and workload dynamics. Integrating workload-aware control with mechanisms such as Dynamic Voltage and Frequency Scaling (DVFS), power capping, or workload shaping could significantly improve energy efficiency in GPU inference systems.

8.6 Conclusion

This work demonstrates that GPU inference systems cannot be effectively understood using traditional performance metrics alone. Through systematic experimentation, we showed that backlog provides a reliable and immediate view of system state, while commonly used signals such as latency and power can be misleading. The results highlight that GPU behavior is shaped by the interaction between stochastic arrivals and batched execution, leading to partial observability and non-intuitive dynamics such as hidden saturation and temporal fragmentation. By framing idleness as a queue-driven phenomenon and validating it empirically, this thesis provides both a clearer understanding of GPU inference behavior and a foundation for predictive, energy-aware system design.

BIBLIOGRAPHY

[1] Announcing TorchServe, An Open Source Model Server for PyTorch | AWS News Blog

https://aws.amazon.com/blogs/aws/announcing-torchserve-an-open-source-model-server-for-pytorch/

[2] TorchServe – Hamel's Blog - Hamel Husain

https://hamel.dev/notes/serving/torchserve/

[3] GPU-NEST: Characterizing Energy Efficiency of Multi-GPU Inference Servers

https://www.cs.ucr.edu/~ajaha004/files/GPU-NEST.pdf

[4] TorchBench: Benchmarking PyTorch with High API Surface Coverage

https://arxiv.org/html/2304.14226

[5] iacoma.cs.uiuc.edu

https://iacoma.cs.uiuc.edu/iacoma-papers/hpca25_2.pdf

[6] GPU Overload No More: A Modern Guide to Optimizing AI Hosting and Performance

https://cloudserv.ai/gpu-overload-no-more-a-modern-guide-to-optimizing-ai-hosting-and-performance/

[7] preprints.org

https://www.preprints.org/manuscript/202512.1908/v1/download

[8] GitHub - ml-energy/zeus: Measure and optimize the energy consumption of your AI applications! · GitHub

https://github.com/ml-energy/zeus

[9] Running TorchServe — PyTorch/Serve master documentation

https://docs.pytorch.org/serve/server.html

[10] Getting started — PyTorch/Serve master documentation

https://docs.pytorch.org/serve/getting_started.html?highlight=model+store

[11] serve/examples/README.md at master · pytorch/serve · GitHub

https://github.com/pytorch/serve/blob/master/examples/README.md

[12] NVML API Reference Guide :: GPU Deployment and Management Documentation

https://docs.nvidia.com/deploy/archive/R515/nvml-api/nvml-api-reference.html

[13] GitHub - ml-energy/zeus: Measure and optimize the energy consumption of your AI applications! · GitHub

https://github.com/ml-energy/zeus

[14] usenix.org

https://www.usenix.org/system/files/atc20-shahrad.pdf

[15] GPU (In)efficiency in AI Workloads | Anyscale

https://www.anyscale.com/blog/gpu-in-efficiency-in-ai-workloads

[16] TensorFlow Serving
https://www.tensorflow.org/tfx/guide/serving

[17] NVIDIA Triton Inference Server
https://docs.nvidia.com/deeplearning/triton-inference-server

[18] TorchServe
https://pytorch.org/serve

[19] Queueing theory
https://en.wikipedia.org/wiki/Queueing_theory

[20] M/M/1_queue

https://en.wikipedia.org/wiki/M/M/1_queue

ANNEX A - Source Code Repository

The complete implementation, experimental scripts, workload generator, telemetry collection framework, and analysis pipeline developed for this thesis are publicly available on GitHub:

https://github.com/Constantinos-Hadjieftychiou/ADE.git

The repository contains:

- TorchServe deployment configuration
- Workload generation scripts
- GPU telemetry monitoring utilities
- Dataset construction scripts
- Analysis and visualization tools
- SLURM batch execution scripts for HPC deployment