THESIS B, 2026:

Summary of thesis aims:

This thesis develops an experimental and modelling framework for understanding how workload structure, architectural topology, and resource constraints jointly shape system behaviour under load. The framework integrates empirical measurement, classical machine learning, and discrete-event simulation, with models parameterised directly from performance traces collected through controlled Docker-based experiments rather than idealised queueing assumptions.

Although evaluated on containerised services, the approach is intentionally application-agnostic and applicable to other performance-critical domains such as DSP pipelines, accelerator-backed services, and AI inference systems, where simple microbenchmarks fail to capture queueing effects, tail latency, and compositional behaviour.

The framework is grounded in a concrete industrial-style case study: predicting latency, saturation thresholds, and tail behaviour under fluctuating load and resource constraints. This addresses a persistent challenge in modern capacity planning and SRE practice, where neither analytical models nor black-box predictors alone are sufficient.

Beyond modelling accuracy, the system is deliberately designed as a teaching instrument for cultivating architectural intuition. Rather than encouraging rote application of performance formulas, it enables students to connect theoretical models with observable system behaviour and to reason critically about design trade-offs.

Rather than treating ML as a full-system predictor, the approach applies ML selectively to capture hard-to-model behaviours (e.g. interference and distributional shifts) and embeds these learned components inside a DES loop.

Iterative Development Plan

Core Principle: Targeted Hybridisation Rather Than Universal Learning

The guiding principle of this thesis is not to replace system modelling with machine learning, but to apply ML only to those system behaviours that are inherently difficult to capture with analytical models or structural simulation. These include behaviours such as nonlinear interference effects, resource contention, distributional shifts under load, and variability introduced by operating system scheduling or runtime effects.

Discrete-event simulation (DES) is used to model structural aspects of the system that are explainable and reusable (e.g. queueing, scheduling, concurrency, and topology). ML components are introduced only to replace specific behavioural components within the simulation where analytical assumptions fail. These learned components are embedded within

the DES loop, forming a hybrid model whose accuracy and generalisability can be evaluated against both ML-only and DES-only baselines.

Iteration 1 — Single-core, single-thread baseline model

The first iteration establishes a validated baseline modelling pipeline for a minimal containerised service. A single-threaded HTTP echo service is deployed within a Docker container constrained to a single CPU core using CPU pinning and quota controls. Load is generated using an open-loop workload generator (Vegeta), while latency, throughput, and utilisation are recorded using the observability stack.

At low arrival rates, measured response times are treated as empirical samples of the underlying service-time distribution. At higher load levels, operational laws are used to estimate mean service demand. These measurements are used to parameterise a discrete-event simulation (DES) implemented in Python, modelling the system as a single-server FIFO queue. The simulator consumes empirical arrival rates and service-time samples to generate predicted response-time distributions.

This iteration is considered successful when the DES reproduces observed behaviour across a sweep of arrival rates, including mean latency trends, utilisation patterns, and response-time distributions. The outcome is a validated experimental methodology and modelling pipeline.

Two additional validation experiments are performed at this stage:

- A comparison between DES driven by raw empirical samples versus DES driven by parametric distribution fits, evaluating generalisation and stability.

- A head-to-head comparison between direct ML-based latency prediction and DES-based prediction, establishing an early baseline for the value contributed by simulation structure.

At this stage, the system remains intentionally simple so that all components are explainable using classical modelling techniques. This establishes a baseline against which later hybridisation can be evaluated. The purpose is not only to validate the simulation pipeline, but to ensure that any later introduction of ML is motivated by genuine modelling limitations rather than convenience.

Iteration 2 — Multi-threaded, multi-core, and multi-replica architectures

The second iteration extends both the experimental system and the simulation model beyond the single-server assumption. The service is modified to use bounded concurrency (e.g. a fixed worker pool), and the DES is extended to model multi-server systems analogous to M/G/c queues. Experiments are conducted across varying numbers of workers and CPU cores.

The system is then extended to multiple container replicas behind a load balancer, and the DES is further extended to represent scale-out behaviour and request routing. This iteration evaluates whether the calibrated modelling pipeline remains predictive as architectural complexity increases, and whether it captures queueing effects, utilisation dynamics, and tail-latency behaviour under more realistic deployment structures.

Iteration 3 — ML-driven modelling of service-time behaviour

Once the modelling pipeline has been validated structurally, machine learning is introduced not as a replacement for the simulator, but as a targeted mechanism for modelling behaviours that resist analytical specification. In practice, this corresponds to modelling how service-time distributions shift under changing resource constraints, interference, or workload characteristics.

Controlled parameters such as CPU quota, arrival rate, concurrency, and workload-specific knobs are systematically varied to generate datasets in which service-time behaviour changes in complex, nonlinear ways. Classical and interpretable ML techniques (e.g. regression models, tree-based models, Gaussian processes, quantile regression, and parametric distribution fitting) are used to learn mappings of the form:

> configuration parameters → service-time distribution characteristics

Rather than predicting latency directly, these learned models are embedded as components within the DES loop, replacing only the service-time generation mechanism while preserving the structural simulation of queueing, scheduling, and topology.

The core research question at this stage is therefore not “can ML predict performance?”, but:

> *Does embedding learned behavioural models inside a structural simulation improve predictive accuracy, generalisability, or interpretability compared to ML-only or DES-only approaches?*

This is evaluated through controlled comparisons between:

- Direct ML-based latency prediction
- Pure DES with manually calibrated parameters
- Hybrid ML-parameterised DES

The expectation is that the hybrid model better captures nonlinear effects under load, generalises more reliably to unseen configurations, and preserves structural interpretability that black-box predictors lack.

Iteration 4 — Diverse workload archetypes (generalisability)

Rather than treating workloads as arbitrary applications, each workload archetype is deliberately designed to induce a specific class of service-time behaviour corresponding to concepts taught in COMP9334 (queueing theory, operational laws, simulation modelling, and performance reasoning). The purpose of these workloads is not realism for its own sake, but to act as controlled experimental instruments through which students can observe where modelling assumptions succeed or fail.

Each workload category serves a distinct pedagogical and modelling purpose:

- **CPU-bound deterministic workloads** produce low-variance, near-stationary service times, suitable for validating basic M/G/1 assumptions and DES calibration.

- **Branching and multimodal workloads** generate mixture distributions, allowing students to observe the breakdown of exponential assumptions and the impact of variability on waiting times.

- **Memory/GC-heavy workloads** induce long-tail latency behaviour, illustrating why mean-based reasoning is insufficient for SLO-sensitive systems.

- **I/O-bound and database-backed workloads** demonstrate dependency-induced variability and the impact of external bottlenecks.

- **Contention-driven workloads** (e.g. mutex or threadpool-limited designs) expose nonlinear saturation effects, enabling reasoning about queue growth, bottlenecks, and system stability.

- **Multi-hop and fan-out workloads** demonstrate composition effects such as tail amplification, correlation, and dependency structure.

These archetypes therefore function as teaching artefacts, allowing students to experimentally connect theoretical concepts (e.g. utilisation law, Little's Law, service-time variability, queue discipline, saturation) with observable behaviour in real systems. This design supports both the research goal of evaluating generalisability and the educational goal of bridging analytical models with empirical system behaviour.

Toward a pedagogical and research artefact

As an extension of the core research contribution, the framework can be exposed as an interactive educational artefact, for example through a notebook-based playground or CLI tool that allows users to:

- Select workload archetypes
- Adjust configuration parameters
- Choose modelling mode (analytical, ML, DES, hybrid)
- Visualise latency distributions, tail behaviour, queue length evolution, and utilisation dynamics

Such a tool would transform the research artefact into a reusable pedagogical platform and a benchmarking suite for future work on hybrid modelling techniques.

Type of programs for running in Docker:

1) CPU-bound, deterministic (control knob = iterations)

A1. CPU loop service

- Each request runs `for i in range(N): do_hash()` or similar.
- “Real” analogue: JSON processing, crypto, templating, small inference.
- Best for: clean service-time control and queueing theory assumptions.

A2. CPU + branchy logic service

- Same as A1 but includes branches (if/else) dependent on input.
- “Real” analogue: business logic, routing, feature flags.
- Adds: multimodality in service times.

A3. CPU + vectorised math (SIMD-ish)

- “Real” analogue: image/audio transforms, DSP-like operations.
- Adds: sensitivity to CPU microarchitecture and cache.

2) Memory / allocation / GC (control knob = allocation rate, heap pressure)

Picking one managed runtime (Java/Go/Node) and applying stress.

B1. Allocation-heavy JSON service

- Parse + build objects + respond.
- Knobs: payload size, object count, response size.
- “Real” analogue: API gateway / backend endpoints.

B2. Cache service (memory residency)

- Keeps an in-memory cache; requests hit/miss based on probability.
- Knobs: cache size, hit rate, eviction policy.
- Adds: statefulness and temporal effects (important for “distribution” realism).

B3. GC stress service

- Alternate bursts of allocations with quiet periods.
- Adds: tail latency spikes, multi-timescale behaviour.

3) I/O-bound (control knob = latency injected, queue depth)

C1. DB read-heavy service (Postgres)

- Fixed query shape.
- Knobs: index on/off, row size, concurrency.
- “Real” analogue: most CRUD services.

C2. DB write-heavy service

- Adds: lock contention and fsync variability.

C3. Disk I/O service

- Reads/writes files; optional fsync.
- Adds: cache effects, noisy neighbour effects.

C4. Network I/O service

- Calls an upstream service; configurable delay/error rate.
- Adds: dependency latency distributions.

4) Contention / synchronisation (control knob = lock intensity)

D1. Mutex/lock contention service

- Threads contend on a shared lock or shared counter.
- “Real” analogue: hot locks in real apps, shared caches, logging bottlenecks.
- Adds: non-linear blowups.

D2. Threadpool-limited service

- Fixed worker pool; requests queue when saturated.
- “Real” analogue: web servers, DB connection pools.
- Adds: explicit queueing + tail growth.

5) Multi-service / topology realism (control knob = fanout and dependency graph)

E1. 2-hop chain: frontend → backend

- Adds: tail amplification and correlation.

E2. Fan-out service

- One request calls N downstream services (or N parallel tasks).
- Adds: “max of random variables” behaviour (p99 gets nasty).

E3. Circuit-breaker / retry service

- Adds: feedback loops, bursty traffic, instability.

6) “Production-ish but still controlled”

F1. Nginx static files

- Mostly kernel/network pipeline, can vary file size.

F2. Redis + simple API

- Shows cache-like patterns without full DB overhead.

F3. Small message queue pipeline

- Producer/consumer with backpressure (RabbitMQ/Kafka-lite).
- Adds: queue dynamics close to real event systems.