Enterprise AI Design Principles: A Prototype Implementation

DISSERTATION

Submitted in partial fulfillment of the requirements of the

Degree: MTech in Artificial Intelligence and Machine Learning

By

Kartikeya Chauhan
2022AC05022

Under the supervision of

Bharat Shukla, Specialty Presales Manager at Dell Technologies

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
Pilani (Rajasthan) INDIA

February 2025

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Mr. Bharat Shukla, for his critical analysis of the problem statement in its ideation phases, strategic orientations for the project, and continued guidance throughout the development, failures and pivoting to more viable approaches throughout the dissertation. It was tough learning that the goal of this undertaking was always to realize what makes an enterprise monitoring system intelligent, and his reminders have taught me to be detached from the success or perfection of isolated components and use experience to capture the required design philosophy and principles that could bridge the gap between idea and implementation.

I also extend my appreciation to Prof. Selvaraj K, my evaluator, for his objective inspection of my progress, from idea to initial implementation to the prototype implementation; his remarks have kept me grounded as I have tried to navigate each facet of the project.

My colleagues, who have kept their patience as I typed away furiously at my keyboard during lunch breaks, and persevered during the numerous times I showed them my progress and asked for opinions and feedback. To have someone to show even a singular ounce of progress during long and frustrating stagnant periods is truly uplifting—even inertia breaks free from itself.

Finally, I thank my close friends and family, who kept me honest and accountable, constantly asking about my dissertation, so that I could dedicate my focus and efforts even when it was tiring and arduous.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CERTIFICATE

This is to certify that the Dissertation entitled Enterprise AI Design Principles: A Prototype Implementation
and submitted by Mr. Kartikeya Chauhan, ID No. 2022AC05022
in partial fulfilment of the requirements of AIMLCZG628T Dissertation, embodies the work

done by him/her under my supervision.

Bharat Shukla
Manager, Cloud Specialty Domain

Place: Bengaluru

Date: 27/02/25

Abstract

Performance, Resource and Network monitoring for enterprise workloads have several proprietary and open-source software dedicated to each critical aspect. This project aims to bridge the gap between isolated implementations and create a holistic enterprise AI system by leveraging machine learning. We aim to build, experiment and optimise this unified system in an effort to intuit and glean design principles from our prototype, especially should a scalable integration be to exist in the future for enterprise deployment. The task is divided into 3 sub-projects: Infrastructure Prediction (resource utilisation) using time-series data; Network Security Analysis— using GNNs; Business Process Optimisation using reinforcement learning. Publicly available datasets like Alibaba Cloud Traces and the UNSW-NB15 network data are used to train our models. The idea is to implement the backbone of the system in C++ for control and granular optimisations, with ML trained using PyTorch/TensorFlow with inferencing in the C++ layer using Libtorch. The goal is to discover fundamental design principles by prototyping unified solutions and demonstrate the feasibility of potentially scaling methodologies to enterprise-grade AI system design. All in all, this project puts forth a prototype as well as frameworks to implement unified systems that combine functionalities of specialised but isolated monitoring solutions in the industry.

Key Words: Enterprise AI Systems, AI integration, Predictive Analytics, Network Security Analysis, Infrastructure, Time Series Analysis, Graph Neural Networks, Reinforcement Learning, Business Process Optimization, Performance Optimization, Unified Monitoring Architecture, C++ Systems Programming, High-Performance Computing.

List of Symbols & Abbreviations used

UNSW-NB15 – University of New South Wales
GCT – Google Cloud Traces
LSTM – Long Short Term Memory
GNN - Graph Neural Networks
RL – Reinforcement Learning
ONNX – Open Neural Network Exchange
API – Application Programming Interface
AWS Cloudwatch – Amazon Web Services Cloudwatch
HPC – High Performance Computing
GCN – Graph Convolution Network
DQN – Deep Q-Learning Network
KDD99 – Knowledge Discovery and Data Mining 1999 Dataset
SVM – Support Vector Machines
ANN – Artificial Neural Networks
ARIMA – Autoregressive Integrated Moving Average
CEO – Chief Executive Officer
APEX – Dell's As-a-service offering
AIOPS – Artificial Intelligence Operations (Dell Software)
CUDA - Compute Unified Device Architecture
NVIDIA RTX – Nvidia's Ray Tracing Texel eXtreme GPUs
CPU – Central Processing Unit
MAE – Mean Average Error
RMSE – Root Mean Square Error
KNN – K-nearest Neighbor Algorithm
MAB – Multi-Armed Bandit Problem
POST – POST REST API METHOD
PROD – Production environment
JSON – Java Script Object Notation
PIT – Point in Time
CI/CD – Continuous Integration and Continuous Deployment

List of Tables

List of Figures

Table of Contents

Chapter 1 - Introduction

This project intersects 3 very siloed industry solutions concerned with Enteprise Systems Management, ML, and HPC (High Performance Computing). Current solutions have very tight, seemingly isolated features (and for good reason) where monitoring is specialized at each level—performance, network and business processes. While there are Enterprise Solutions (virtually black boxes), there has been an AI approach to central monitoring such as Splunk for Log analysis [1] and Dynatrace for intelligent infrastructure monitoring [2], AWS cloudwatch is built into AWS for resource monitoring/anomaly detection; Open-Source solutions in Prometheus or Grafana for metrics collection + visualization, OpenTelemetry, Apache SPOT and so on; a lot of these systems, focus on their own domains and few solutions combine all 3 aspects (infrastructure, security, business). Even ML implementations use basic statistical analysis and few use advanced ML like GNNs with almost none using Reinforcement Learning.

The approach in this project tries to implement custom ML algorithms for each of the 3 use cases, with correlations across domains in an integrated or unified architecture and maximum control of the software using C++ as the core implementation language with PyTorch for implementing ML models.

1.1 Objectives

The objectives of this project are as follows:

i. Prototype a unified monitoring solution that combines network security analysis, infrastructure prediction and business processes using ML methodologies.
ii. Implement time-series analyses using possibly GCT dataset (Google Cloud Traces) [3], with focus on CPU, IO and memory.
iii. Implement GNNs for network anomaly and threat detection using UNSW-NB15 dataset [4].
iv. Implement Reinforcement Learning models for business process optimization, to find balanced trade-offs between resource allocation and maximum workflow efficiency.
v. Weave up the 3 implementations above in a C++ environment to try and achieve low latency and a novel architecture/design principle.
vi. Test against benchmarks and deploy the prototype with a front-end web interface (if possible) or a local deployment.

1.2 Scope of Work

This project has 3 components:

Backend: A C++ spine for granular control over ML integration from python models using TF/PyTorch; Data collection and preprocessing pipelines—each layer will have separate pipelines; API integration with core C++ backend.
ML components: Time-series LSTM analysis for infrastructure prediction; GNN implementation for network security analysis; RL models for process optimization using DQN algorithms.
Possible Frontend implementation: A web-based monitoring dashboard; Visualizations for all 3 layers: Infrastructure prediction, network analysis and business process optimization.

Chapter 2 – Literature Review

The purpose of this literature review is not to build and iterate on existing enterprise solutions or open-source endeavors, or to review design methodologies for a unified monitor. We will briefly look at the 'current state' of enterprise monitoring and inspect if there are truly any fragmentations in their monitoring approaches. But the core chunk of our reading will relate to Deep Learning approaches to isolated/specialized problems (the solutions to which we are attempting to unify centrally). It would also benefit us to briefly understand the evolution of monitoring systems regarding dominating methodologies of each era.

2.1 Evolution of Infrastructure Monitoring

We stand today at Deep Learning approaches like LSTM networks for time-series predictions, with quite literally monitoring the health of a physical machine by sensory conditions showing tool wear and tear [5]; attention mechanisms combined with RNNs to demonstrate model introspection without sacrificing impressive DL model performances while performing anomaly detection during log analysis [6]; as well as optimizations in computation time by introducing transfer learning for resource allocation (compared to DL methods without transfer learning) [7].

But it was as early as 1993 that time series predictions 'predicted' the possibility of it being used to analyze nonlinear stochastic processes [8], which is similar to economic and business theories combined with behavioral theories to make decisions on cost optimizations based on multiple dynamic parameters. Traditional rule-based monitoring using Nagios [9] or Zabbix [10] moved to Statistical Approaches, such as Time Series Analysis using Autoregressive Integrated Moving Average (ARIMA) models [11] to where we are today.

2.2 Evolution of Network Security Monitoring

We start to see similar trends in the evolution of all these isolated monitoring solutions, because even though the category is forced (by us as the discriminants), technologies share the same mathematical and philosophical backbones of their time. From matching known attack signatures using SNORT ids [12] to packet analysis (manually) using Wireshark [13] to eventually more statistical profiling using anomaly-based methods, where research led us to a statistical analysis of the UNSW-NB15 dataset while comparing it to the KDD99 dataset [14].

Classical ML applications were not far behind as SVMs and ANNs were also trying to help in intrusion detection, with SVMs outclassing ANNs in some cases with separate encoding methods [15]. Since our curriculum covered Graph Neural Networks, we also looked at its applications, in a much broader sense rather than highly specific to network analysis [16].

2.3 Evolution of Business Process Optimizations

This is a much broader topic as there are a lot of business methodologies from memoirs or experiences that successful CEOs or entrepreneurs wrote but we will steer clear of manual process management, even though there is value to localized environments and human behavior. There is a lot of potential for reading here but our goal is to stick to classical approaches as stepping stones for more complex agents to try out eventually.

We stick to the classical Multi-Armed Bandit Problem [17] for our RL agent approach with balancing exploitation and exploration to optimize infrastructure costs and network security monitoring [18].
In the future, we can look at modern libraries for more control and nuance to RL agents, such as OpenAI's educational resource 'Spinning Up in Deep RL' [19] and their Gymnasium library [20].

2.4 Unified Monitoring Approaches

With Monitoring frameworks like Prometheus [21] and Grafana [22], even Splunk has come a long way from log analysis to unified approaches with AI insights, as mentioned earlier [1]. We also focus on Dell APEX AIOPS for AI-driven health, cybersecurity and sustainability insights [23], proving that there exist unified monitoring approaches. Even with proprietary code, a lot of these focus on an API centric approach for ease of usability of metrics and insights.

The goal of our prototype is not to surpass or model the performance of these existing tools, but start from ground up and see where the gaps are compared to enterprise level monitoring tools.

2.5 Implementation Approaches

Due to the comfort of using TensorFlow, we considered using PyTorch, specifically Libtorch for C++ inferencing. Some research shows that PyTorch is more suited for speed and is more flexible with integration options compared to TensorFlow, which prioritizes accuracy [24]. Besides, it will be a great learning experience to familiarize oneself with newer technologies and see how well they perform for our use case.

Chapter 3 - Project Setup and unified architecture skeleton

The initial objective states prototyping a unified monitoring solution that combines network security analysis, infrastructure prediction and business processes using ML methodologies.

The language we use in these chapters will be a real-time thought process with insights and flexible change in directionality of approach. There is no static, singular narrative running across the chapters that tells a smooth story of our methods.

The first goal was to identify the platform for development as there several options available, with TensorFlow and Pytorch using python as separate deployments for all 3 components mentioned above (using LSTM, GNN and RL respectively), with CUDA libraries for local training and testing. This created another issue: how do we integrate the layers? While a dynamic project with a stream of input (essentially a real-time machine learning application) is probably beyond the scope of this project, we can certainly aim to establish some design principles that could scale from a static, smaller deployment to enterprise level with large amounts of input data buffered in real-time.

To achieve this architectural basis, the first sub-goal was to set up the project with a skeleton that we could later fill on. But this skeleton had to support a modular approach that could be tweaked easily without breaking the entire structure, as well as have a sound central spine for data collection from outputs of the infrastructure prediction modules and the network security modules to possibly use as the input for the business process module (using reinforcement learning at this level).

For this reason, we chose PyTorch for C++ using Libtorch [25] for preprocessing, collecting data from outputs of the models trained in PyTorch using python, inferencing as well as being the central API backbone for the entire project with cmake to build the project. With CUDA 12.4 [26] libraries installed on the NVIDIA drivers for the RTX 3060 GPU, the training and testing process could be greatly simplified.

Going with Libtorch introduces its own problems since the documentation is not as extensive as PyTorch using python, but beneath the wrappers of the latter, most of it is C++ code. This could help in designing principles for both isolation and modularity of the project as well as opportunities to scale.

This is the initial directory structure of the project:

```
├──build
├──data
├──docs
├──include
│   ├──data
│   │   data_collector.hpp
│   │
│   ├──ml
│   │   │  model_interface.hpp
```

```
        ├──gnn
        │   network_model.hpp
        │
        ├──lstm
        │   infrastructure_model.hpp
        │
        └──rl
            business_model.hpp
    └──preprocessing
        data_collector.hpp
        data_pipeline.hpp
        monitoring_types.hpp

├──lib
    ├──libtorch
    ├──onnxruntime
    └──tensorflow
├──pytorch models
├──src
    CMakeLists.txt
    main.cpp

    ├──core
        CMakeLists.txt
        core.cpp

    ├──ml
        CMakeLists.txt

        ├──gnn
            CMakeLists.txt
            network_model.cpp

        ├──lstm
            CMakeLists.txt
            infrastructure_model.cpp

        └──rl
            business_model.cpp
            CMakeLists.txt

    └──utils
        CMakeLists.txt
        utils.cpp

├──tests
    CMakeLists.txt
```

.gitignore
CMakeLists.txt

The CMakeLists.txt represents the directory metadata and is used for configuring variables for building the project too. Let us look at the CMakeLists.txt to understand the hierarchy of our projects:

Enterprise AI Monitor > CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)
project(EnterpriseAIMonitor CUDA CXX)

# Set CUDA toolkit first
set(CMAKE_CUDA_COMPILER "C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.4/bin/nvcc.exe")
set(CUDA_TOOLKIT_ROOT_DIR "C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.4")
set(CUDAToolkit_ROOT "C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.4")

# Find CUDA package
find_package(CUDAToolkit REQUIRED)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Add the include directory to the include path
include_directories(${PROJECT_SOURCE_DIR}/include)

# Set Torch path and find package
set(CMAKE_PREFIX_PATH "C:/Users/91965/Downloads/libtorch-win-shared-with-deps-2.5.1+cu124/libtorch")
find_package(Torch REQUIRED)

# Add subdirectories
add_subdirectory(src)
add_subdirectory(tests)
```

Fig 3.1 - root CMakeLists.txt

We first set some environment paths after defining the project title (and instructing it to use CUDA) -- CUDA toolkit directories are linked as well as the separate libtorch package corresponding to the CUDA version. The libtorch path would likely need to change and point to the local lib sub-directory for libtorch. It is also crucial that the libtorch version matches the CUDA version since there were missing dependencies earlier on due to 11.8 and 12.4 version mismatches and yet ironically, installers for both versions are needed for the libtorch libs for 12.4 [27].

The include directory will have hpp files to include in our cpp files inside the src directory. These hpp files will have the basic data structures required as well as abstract methods designed to implement in those cpp files.

For example, the data_collector.hpp file for unifying inputs from the infrastructure prediction and network security modules looks somewhat like this (snippet, not full code) where we can clearly see integrations from our ML models and also the data types + thread safe queues for data integration:

```cpp
class UnifiedDataCollector {
private:
    // Shared pointers to our ML models
    std::shared_ptr<ml::InfrastructureLSTM> infra_model;
    std::shared_ptr<ml::NetworkGNN> network_model;

    // Thread-safe queues for different data types
    std::queue<ml::InfrastructureMetrics> infra_queue;
    std::queue<ml::NetworkGraph> network_queue;

    // Synchronization primitives
    std::mutex infra_mutex;
    std::mutex network_mutex;
    std::condition_variable cv;

    bool running;

    // Configuration parameters
    struct Config {
        size_t queue_size_limit;
        int collection_interval_ms;
        std::string data_directory;
    } config;

public:
    UnifiedDataCollector(const Config& cfg,
                         std::shared_ptr<ml::InfrastructureLSTM> infra,
                         std::shared_ptr<ml::NetworkGNN> network);

    void start();
    void stop();

    void addInfrastructureData(const ml::InfrastructureMetrics& metrics);
    void addNetworkData(const ml::NetworkGraph& graph);

private:
    void processInfrastructureData();
    void processNetworkData();
};
```

Fig 3.2 - data_collector.hpp

Similarly, we have hpp files for all our ML models (GNN, LSTM, RL) to handle those methods and data types (model interface), as well as data integration and preprocessing pipelines.

Our src directory has sub-directories for utils (currently a placeholder for util functions), core (core functionality/spine), and ml sub-directory for lstm, gnn and rl models. These cpp files (for the models) contain loading our pt models, prediction logic, update logic and so on.

For example, here is our LSTM code snippet (in src/models/infrastructure_model.cpp) to handle implementation logic:

```cpp
try {
    // Implementation of prediction logic
    std::vector<float> input_data;
    for (size_t i = metrics_buffer.size() - sequence_length;
        i < metrics_buffer.size(); i++) {
        input_data.push_back(normalize(metrics_buffer[i].cpu_usage, cpu_mean, cpu_std));
        input_data.push_back(normalize(metrics_buffer[i].memory_usage, mem_mean, mem_std));
        input_data.push_back(normalize(metrics_buffer[i].io_rate, io_mean, io_std));
    }

    auto input_tensor = torch::from_blob(input_data.data(),
        {1, sequence_length, 3}).to(device);

    std::vector<torch::jit::IValue> inputs;
    inputs.push_back(input_tensor);
    auto output = model.forward(inputs).toTensor();

    return true;
} catch (const c10::Error& e) {
    std::cerr << "Error during prediction: " << e.msg() << std::endl;
    return false;
}
```

Fig 3.3 - LSTM prediction logic using libtorch

Each directory and sub-directory will have their own CMakeLists.txt file for local metadata and build parameters. But in essence, this completes the basic setup for our project, with individual components left to be solved for - the 3 separate machine learning models, which we will discuss in the following chapters.

As you can see, the first build (using cmake [28]) was successful for our skeleton (files contained in the build directory) with changes tracked using git:

```
  C:/Users/91965/Downloads/libtorch-win-shared-with-deps-2.5.1+cu124/libtorch/share/cmake/Torch/TorchConfig.cmake:68 (find_package)
  CMakeLists.txt:20 (find_package)


-- USE_CUDNN is set to 0. Compiling without cuDNN support
-- USE_CUSPARSELT is set to 0. Compiling without cuSPARSELt support
-- USE_CUDSS is set to 0. Compiling without cuDSS support
-- USE_CUFILE is set to 0. Compiling without cuFile support
-- Autodetected CUDA architecture(s):  8.6
-- Added CUDA NVCC flags for: -gencode;arch=compute_86,code=sm_86
-- Found Torch: C:/Users/91965/Downloads/libtorch-win-shared-with-deps-2.5.1+cu124/libtorch/lib/torch.lib
-- Configuring done (15.0s)
-- Generating done (0.2s)
-- Build files have been written to: C:/Users/91965/Desktop/M.Tech AI ML/Semester 4/Enterprise AI Monitor/build
PS C:\Users\91965\Desktop\M.Tech AI ML\Semester 4\Enterprise AI Monitor\build> cd ..
PS C:\Users\91965\Desktop\M.Tech AI ML\Semester 4\Enterprise AI Monitor> git add .
PS C:\Users\91965\Desktop\M.Tech AI ML\Semester 4\Enterprise AI Monitor> git commit -m "first prelim build successful"
[master daabb6d] first prelim build successful
```

Fig 3.4 - first cmake build

Chapter 4 - Infrastructure Prediction using LSTM

In this chapter we will talk about our LSTM implementation using Pytorch as well as setting up the libtorch components for inferencing and model integration later on.

Initially we planned to use Google Traces output from Google Cloud. Apparently, this is 2.4TiB of uncompressed data and needs to be queried using BigQuery [29]. There is a 2019 cluster dataset on the same repository but that is also hundreds of GB. To address this issue, we looked at first using BigQuery to obtain a sample of the GCT dataset and save some of it locally. There was a much better solution: Alibaba Cloud Trace Data [30]. It requires no sign up and we promptly went with the 2017 dataset for server_usage which was around 18MB (csv) with ~187K inputs.

Here is the schema for this dataset [31]:

dataset	column	feature	format	mandatory
server_usage.csv	1	timestamp	INTEGER	YES
server_usage.csv	2	machine id	INTEGER	YES
server_usage.csv	3	used percent of cpus(%)	FLOAT	YES
server_usage.csv	4	used percent of memory(%)	FLOAT	YES
server_usage.csv	5	used percent of disk space(%)	FLOAT	YES
server_usage.csv	6	linux cpu load average of 1 minute	FLOAT	YES
server_usage.csv	7	linux cpu load average of 5 minute	FLOAT	YES

Table 4.1 - alibaba traces data schema

Initially as mentioned in our objectives, we just wanted to predict CPU utilization using memory and disk space (2 input features). But since the UNSW-NB15 dataset mentions all values as mandatory, why not use all features to predict CPU? So we shifted to using all 6 features instead. This is useful since we were getting very high losses using only 3 features:

```
Epoch 0/150, Train Loss: 778.8547, Test Loss: 493.0964
Epoch 10/150, Train Loss: 671.6022, Test Loss: 390.8331
Epoch 20/150, Train Loss: 497.2652, Test Loss: 278.3938
Epoch 30/150, Train Loss: 415.4488, Test Loss: 225.0544
Epoch 40/150, Train Loss: 357.9089, Test Loss: 187.8149
Epoch 50/150, Train Loss: 308.3628, Test Loss: 157.1373
Epoch 60/150, Train Loss: 265.9868, Test Loss: 132.3478
Epoch 70/150, Train Loss: 229.8139, Test Loss: 112.5374
Epoch 80/150, Train Loss: 198.9916, Test Loss: 96.9474
Epoch 90/150, Train Loss: 172.7785, Test Loss: 84.9210
Epoch 100/150, Train Loss: 150.5441, Test Loss: 75.8995
Epoch 110/150, Train Loss: 131.7533, Test Loss: 69.4049
Epoch 120/150, Train Loss: 115.9463, Test Loss: 65.0224
Epoch 130/150, Train Loss: 102.7219, Test Loss: 62.3885
Epoch 140/150, Train Loss: 91.7257, Test Loss: 61.1822
Model and weights saved
```

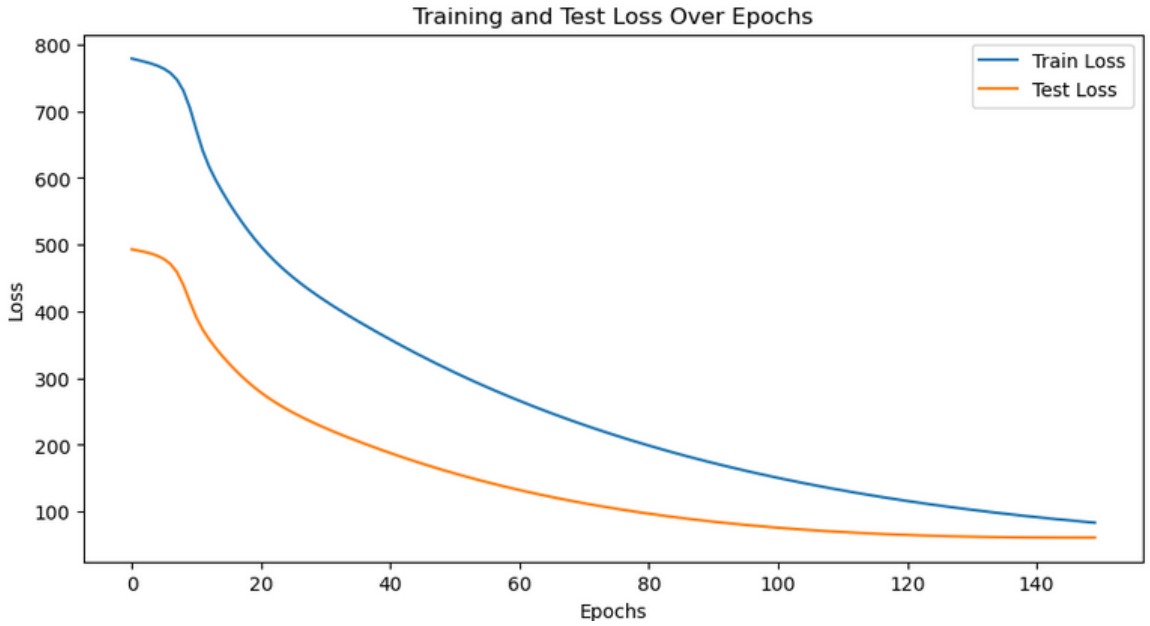


Fig 4.1 - Initial Training Loss with 3 n-dims

So, during preprocessing, we loaded all the features and didn't drop any of them. These were then (both features and target) normalized using MinMaxScaler() and sequences were created for the time series data.

To make the LSTM model more effective, we added a simple attention module as well in addition to the LSTM architecture:

```python
class AttentionLayer(nn.Module):
    def __init__(self, hidden_dim):
        super(AttentionLayer, self).__init__()
        self.attention = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x):
        attention_weights = F.softmax(self.attention(x), dim=1)
        context_vector = torch.sum(attention_weights * x, dim=1)
        return context_vector
```

Fig 4.2 - attention module

The LSTM architecture is straightforward: it has 7 input_dims (as we stated), 128 hidden dims, 2 layers (with 0.3 dropout applied) and an output_dim. Also, to better interpret the results, RMSE and MAE were added to evaluate on the actual scale of the data, which makes more sense.

After multiple reruns and settling on 1e-3 for LR and a sequence length of 10 over 50 epochs, these were our results:

Epoch 0/50, Train Loss: 0.6206, Test Loss: 0.1313
Epoch 10/50, Train Loss: 0.0304, Test Loss: 0.0333
Epoch 20/50, Train Loss: 0.0278, Test Loss: 0.0291
Epoch 30/50, Train Loss: 0.0258, Test Loss: 0.0323
Epoch 40/50, Train Loss: 0.0263, Test Loss: 0.0315

Metrics on the original scale:
RMSE: 6.33
MAE: 5.35

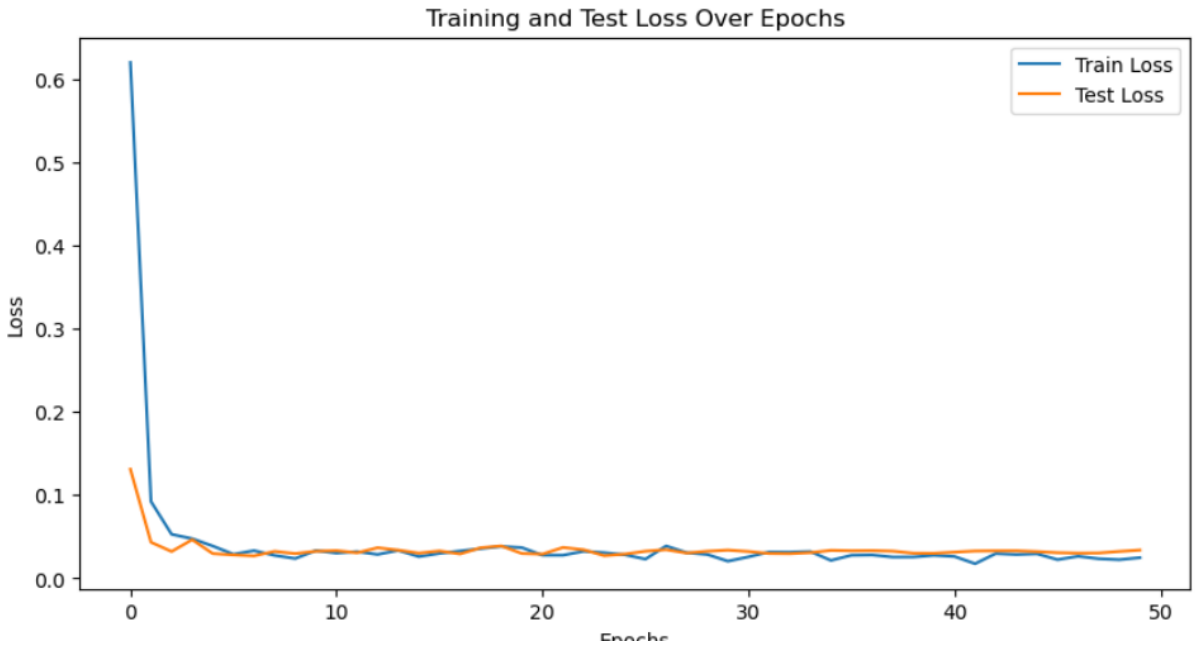
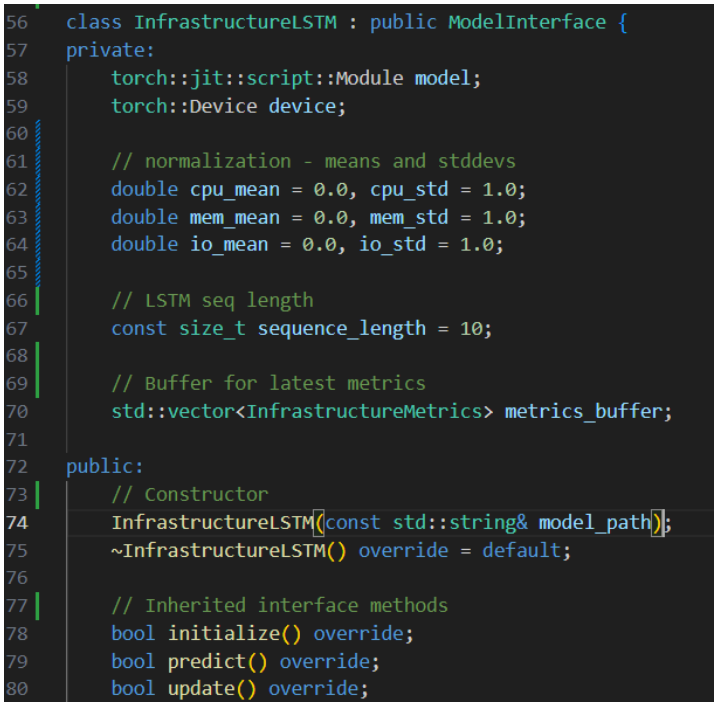


Fig 4.3 - Final Training/Test Loss

As we see, the model converges very quickly and no wonky oscillations, indicating decent generalization. RMSE is 6.33 - this means for average CPU usage, predictions are off by 6.33% which is fairly acceptable. The MAE of 5.35 denotes actual difference of 5.35 points (these would be % points over the FLOAT comparisons for CPU %).

For scalability for real-time infrastructure predictions, actual CPU readings might be super volatile due to server utilization so the error rate might work to our advantage as a safety buffer and counter-intuitively serves as a possible design principle for enterprise monitoring. We can also use this variance number in the future to detect anomalies or any resource constraints.

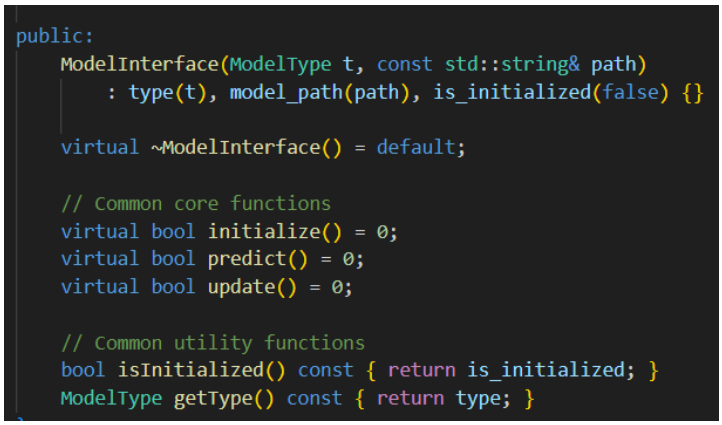
On the libtorch side, we rehauled the code for the infrastructure_model.hpp to include more base classes, for example, our Infrastructure LSTM class:



```
class InfrastructureLSTM : public ModelInterface {
private:
    torch::jit::script::Module model;
    torch::Device device;

    // normalization - means and stddevs
    double cpu_mean = 0.0, cpu_std = 1.0;
    double mem_mean = 0.0, mem_std = 1.0;
    double io_mean = 0.0, io_std = 1.0;

    // LSTM seq length
    const size_t sequence_length = 10;

    // Buffer for latest metrics
    std::vector<InfrastructureMetrics> metrics_buffer;

public:
    // Constructor
    InfrastructureLSTM(const std::string& model_path);
    ~InfrastructureLSTM() override = default;

    // Inherited interface methods
    bool initialize() override;
    bool predict() override;
    bool update() override;
```

Fig 4.4 - LSTM hpp class

We have a separate model_interface bass header file for 3 models as well with common functionality for all of them:



```
public:
    ModelInterface(ModelType t, const std::string& path)
        : type(t), model_path(path), is_initialized(false) {}

    virtual ~ModelInterface() = default;

    // Common core functions
    virtual bool initialize() = 0;
    virtual bool predict() = 0;
    virtual bool update() = 0;

    // Common utility functions
    bool isInitialized() const { return is_initialized; }
    ModelType getType() const { return type; }
```

Fig 4.5 - model_interface base header

So in these libtorch components, we handle the architecture that mirrors the one we created earlier in our jupyter notebook, as well as pipeline and inferencing. We will see something similar for the other 2 components (security and business) as well, outlining key design principles for modularity while having the ability to scale without major alterations in any of the 3 components if we keep the base methods similar structurally.

Chapter 5 - Network Security Module using GNN

For the Network Security Analysis module, it was clear whether the UNSW-NB15 dataset could be the right fit as we had looked at the dataset and understood the components and its structure well enough.

This is the schema of the dataset (it already has a train/test split--257, 673 total rows):

id dur proto service state spkts dpkts sbytes dbytes rate sttl dttl sload dload sloss dloss sinpkt dinpkt sjit djit swin stcpb dtcpb dwin tcprtt synack ackdat smean dmean trans_depth response_body_len ct_srv_src ct_state_ttl ct_dst_ltm ct_src_dport_ltm ct_dst_sport_ltm ct_dst_src_ltm is_ftp_login ct_ftp_cmd ct_flw_http_mthd ct_src_ltm ct_srv_dst is_sm_ips_ports attack_cat label

As you can see, there are a lot of fields reflecting things like response body, tcp port and so on. The dataset predicts 10 attacks ranging from exploits to DoS to backdoors.

We will try 2 approaches for this task. First, construct a KNN graph (n=5) from the data. Now it is obvious that this might not be the most meaningful way to represent the relationships between nodes using KNNs. However, we will focus our 2nd approach based on results from this rudimentary approach. Then, create a simple GCN (2 layers) and run a full batch GCN with an evaluation on test data in the end. For reference, we are using the attack_cat instead of the label for multi-class classifications for more nuance.

This was our train/test split:

Data(x=[175341, 42], edge_index=[2, 752187], y=[175341])
Train data: #nodes = 175341 #edges = 752187 num_features = 42
Data(x=[82332, 42], edge_index=[2, 343691], y=[82332])
Test data: #nodes = 82332 #edges = 343691 num_features = 42

We ran it for 10 epochs and this is what we got:

Using device: cuda
Epoch: 01, Loss: 2.5330
Epoch: 02, Loss: 1.7841
Epoch: 03, Loss: 1.4590
Epoch: 04, Loss: 1.2525
Epoch: 05, Loss: 1.1291
Epoch: 06, Loss: 1.0703
Epoch: 07, Loss: 1.0292
Epoch: 08, Loss: 0.9823
Epoch: 09, Loss: 0.9447
Epoch: 10, Loss: 0.9238

This sounds good but our test accuracy was 56%, which is low. And expected. Since this using a KNN that might not capture the true graph relationships between the fields. This leads us to the next task--building a 'real' graph by connecting some attributes that share the same proto, service, state triplet. Again, this might not be the best approach, but graph theory (in Neural Networks) is still in its experimental stages. We can consider different relations like source or destination IP.

Following the results and the thought process on using an actual graph instead of KNN, we built a simple GCN that improved the accuracy to 70%, a notable increase even if it's not really up there.

Here is the GCN we defined, a fairly simple network with some dropout added:

```
class GCN(torch.nn.Module):
    def __init__(self, num_features, hidden_channels, num_classes, dropout_p=0.5):
        super().__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, num_classes)
        self.dropout_p = dropout_p

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_p, training=self.training)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_p, training=self.training)
        x = self.conv3(x, edge_index)
        return x
```

Fig 5.1 – GCN Model

Training Loss reduced quickly as you can see here:

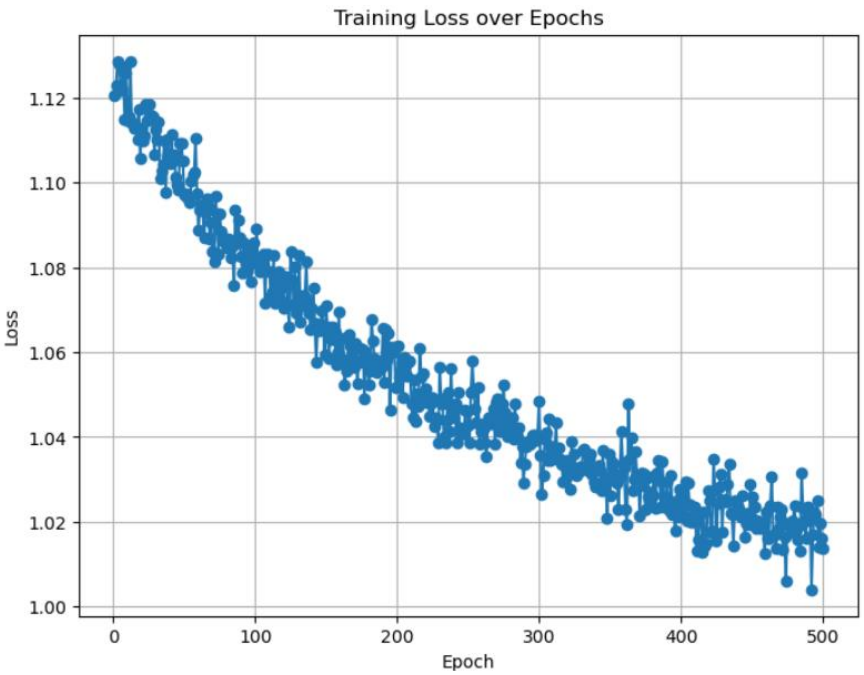


Fig 5.2 – GCN Training Loss

In our classification report, we see that some attacks are completely blank--this is because of the 3 fields we selected as inputs (protocol, state, service) that might not apply to all the attacks:

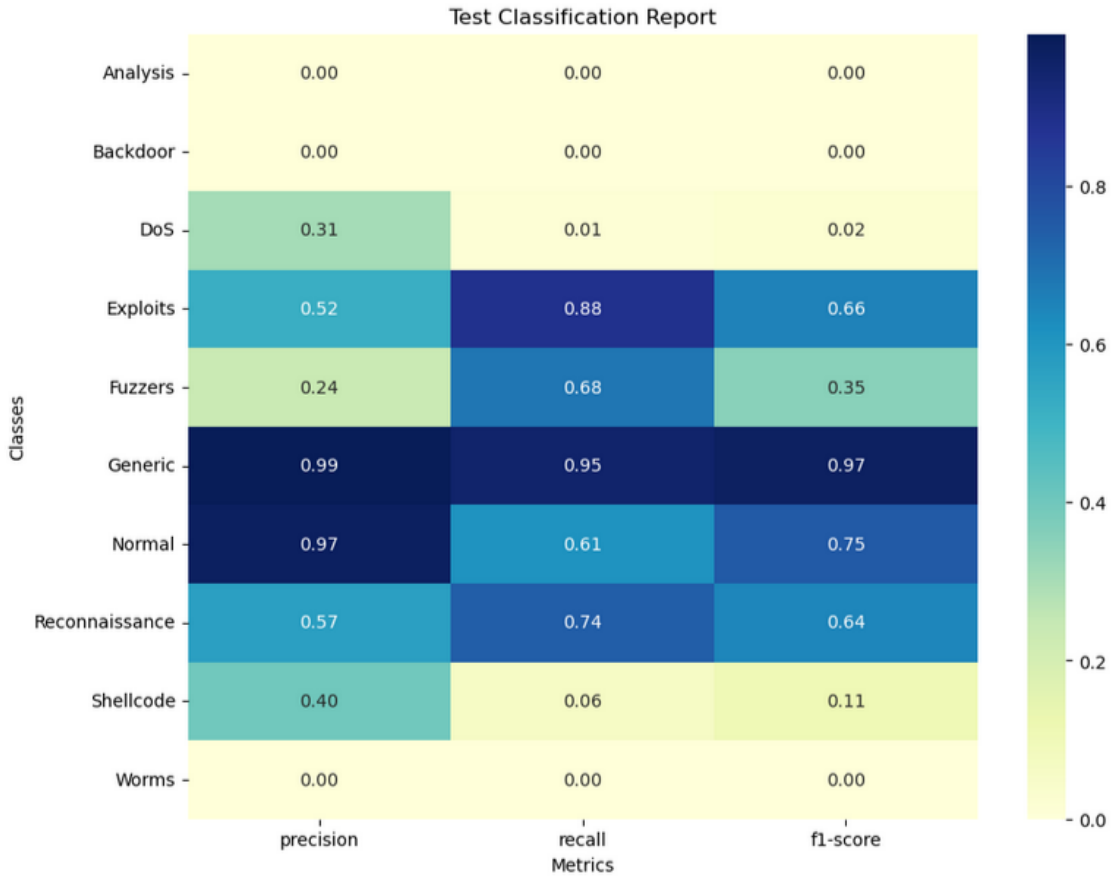


Fig 5.3 – GCN Classification Report

We can certainly take more input vectors and possibly improve test accuracy. Although the point is that we have a 'decent enough' model for practical purposes in order to build the centralized monitoring and recommendation system. However, there are several methods we can use to increase accuracy:

- Include more input vectors (slower training, more resources needed) like packet sizes, port numbers, etc.
- Test different GCN model architectures with nuanced graph connectivity: use domain-specific interconnectivity and relationships between spatially linked concepts instead of random fields.
- Adding attention mechanisms to capture edge weights more accurately.
- Sampling to address the obvious class imbalance that we see above.

Chapter 6 - Business Process Module

The infrastructure predictions and network threat predictions seem to work nominally well within their own isolated inferences. Now the real task begins to take shape--what should be the nature of the reinforcement learning agent that applies business constraints? Is it a static third-variable imposition that controls the tradeoff between resource utilization and network threat monitoring, say, by using a flat $ amount for threats (if successful) and similarly for CPU utilization? Of course, using cloud resources intuitively solves this for us with the projected cost numbers. But even then it is just a statistical modelling insight. What can we use as design principles for a business process module approach?

Let us start simple and apply it, and potentially try to describe problems with the first-approach and use them as stepping stones to refine our design philosophy. Prototyping is a method of learning the practical approach in enterprise scenarios. So we will stick to simple applications and assess what is lacking instead of solving the problem ad infinitum.

Why not leverage the Multi-Armed Bandit Problem for our use case? [32] The arms or actions, in this case, will try to identify the ideal tradeoff between exploration and exploitation. Perhaps the results can be used to intuit the objective nature of business constraints beyond standard or custom business practices, or at least paradigms that businesses can take into account while enforcing their own criteria.

The nature of our learning agent was applied using DQN RN (Deep-Q Network) [33], sampling random prior actions in the experience relay.

In our approach, these are the actions/arms we went with, for simplicity:

```
# Action mapping
self.actions = {
    0: "INCREASE_RESOURCES",
    1: "DECREASE_RESOURCES",
    2: "MAINTAIN_RESOURCES",
    3: "ENHANCE_SECURITY",
    4: "OPTIMIZE_COST"
}
```

Fig 6.1 - MAB Actions

We also set epsilon decay at 0.995 with 1 as starting epsilon and 0.01 as its minimum value. Batch size was 64 with a gamma of 0.99 and Adam as the optimizer.

This is as general as a stock trading RL agent that can either hold, buy or sell. But we add multiple dimensionalities to not simply make it a univariate problem: security and cost. More resources can 'cost' more, but sometimes that 'cost' can be more than just the immediate dollar amount: it can be potentially wasted revenue or missed outcomes that businesses can evaluate at their own discretion.

But let us briefly review our LSTM and GNN models from our previous chapters, as we re-create the model architecture here in our business process module and load the saved best models from previous isolated runs. It is key that we incorporate the same architecture otherwise model loading will fail.

The LSTM model definition with attention layer to add some nuance:

```python
# LSTM Model Definition
class AttentionLayer(nn.Module):
    def __init__(self, hidden_dim):
        super(AttentionLayer, self).__init__()
        self.attention = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x):
        attention_weights = F.softmax(self.attention(x), dim=1)
        context_vector = torch.sum(attention_weights * x, dim=1)
        return context_vector

class InfrastructureLSTM(nn.Module):
    def __init__(self, input_dim=6, hidden_dim=128, num_layers=2, output_dim=1, dropout=0.3):
        super(InfrastructureLSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # Input layer
        self.input_layer = nn.Linear(input_dim, hidden_dim)

        # LSTM layers with residual connections
        self.lstm_layers = nn.ModuleList([
            nn.LSTM(hidden_dim, hidden_dim, 1, batch_first=True, dropout=dropout)
            for _ in range(num_layers)
        ])

        # Attention layer
        self.attention = AttentionLayer(hidden_dim)

        # Output layers
        self.fc1 = nn.Linear(hidden_dim, hidden_dim // 2)
        self.dropout = nn.Dropout(dropout)
        self.fc2 = nn.Linear(hidden_dim // 2, output_dim)

        # Layer normalization
        self.layer_norm = nn.LayerNorm(hidden_dim)
```

Fig 6.2 - LSTM model definition

Forward pass for the LSTM model defined thus:

```
def forward(self, x):
    # Input projection
    x = self.input_layer(x)

    # Process LSTM layers with residual connections
    h = x
    for lstm in self.lstm_layers:
        lstm_out, _ = lstm(h)
        h = lstm_out + h  # Residual connection
        h = self.layer_norm(h)  # Layer normalization

    # Apply attention
    context = self.attention(h)

    # Output layers
    out = self.fc1(context)
    out = F.relu(out)
    out = self.dropout(out)
    out = self.fc2(out)

    return out
```

Fig 6.3 - LSTM forward pass

And here is the GNN (GCN) model that we used earlier:

```
# GNN Model Definition
class GCN(torch.nn.Module):
    def __init__(self, num_features, hidden_channels, num_classes, dropout_p=0.5):
        super().__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, num_classes)
        self.dropout_p = dropout_p

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_p, training=self.training)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_p, training=self.training)
        x = self.conv3(x, edge_index)
        return x
```

Fig 6.4 - GNN Model

This is our DQN model, for reference, a simple approach:

```
# DQN Model Definition
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim)
        )

    def forward(self, x):
        return self.network(x)
```

Fig 6.5 - DQN Model

The state space consists of a 4D vector having CPU util, memory util, security threat probability and cost. But here we come to a crossroads--we must think of some thresholds to define optimal/expected resources, security, and so on. Of course, each organization has its own policies but here we present some design principles as baselines for standard practice:

```
# Resource utilization reward (target 60-80% CPU utilization--fair enough?)
if 0.6 <= cpu_util <= 0.8:
    reward += 10
elif cpu_util > 0.9:
    reward -= 20  # Penalize high CPU utilization [>90%]
elif cpu_util < 0.3:
    reward -= 10  # Penalize very low CPU utilization (inefficient)
```

Fig 6.6 - Resource utilization rewards

As you can see, the target is between 60-80% utilization, so we try to increase the reward in order to achieve that sweet spot. Otherwise, either extremes are bad, either wasteful (<30%) or too high (>90%) so we try to elastically bring them to our defined sweet spot.

Let's look at security threat penalties (these are probabilistic outputs). Here we reward reduction in high threats by increasing the reward by 20:

```
# Security threat penalties
if security_threat > 0.7:  # High security threat
    reward -= 30

    # Reward for reducing high security threats
    if next_security_threat < security_threat and action == 3:  # ENHANCE_SECURITY action
        reward += 20
```

Fig 6.7 - Security threat rewards

Finally, based on our policies above, we define the ACTIONS mentioned earlier above in Figure 6.1:

```
# Resource action alignment
if action == 0:  # INCREASE_RESOURCES
    # Reward if CPU was low and is now higher
    if cpu_util < 0.5 and next_cpu_util > cpu_util:
        reward += 10
    # Penalize if CPU was already high
    elif cpu_util > 0.8:
        reward -= 15

elif action == 1:  # DECREASE_RESOURCES
    # Reward if CPU was high and is now lower but still adequate
    if cpu_util > 0.8 and next_cpu_util < cpu_util and next_cpu_util > 0.5:
        reward += 15
    # Penalize if CPU was already low
    elif cpu_util < 0.4:
        reward -= 10

# Cost optimization
cost_diff = next_cost - cost
if cost_diff < 0:  # Cost decreased
    reward += min(15, abs(cost_diff))  # Cap the reward
elif cost_diff > 0:  # Cost increased
    # Only penalize if the action was OPTIMIZE_COST
    if action == 4:  # OPTIMIZE_COST but cost increased
        reward -= min(20, cost_diff)
    # For other actions, smaller penalty for cost increase
    else:
        reward -= min(5, cost_diff)

# Special case: OPTIMIZE_COST action
if action == 4 and cost_diff < 0:
    reward += 10  # Additional reward for successful cost optimization
```

Fig 6.8 - MAB ACTIONS

These are fairly intuitive: increase or decrease resources as recommendations in addition to cost optimizations with some fluctuations in cost as well as CPU, memory and security. These will be important for our input stream as it has to necessarily be a common input stream that both our LSTM and GNN models can capture as a singular structure, and for this reason we are simulating the environment and the input stream. Of course, we have evaluated the LSTM and GNN models on real datasets so a simulation to test it out is within the academic inquiry we are hoping to answer.

Consequently, here are detailed action descriptions:

```
# Base random fluctuations (small)
cpu_fluctuation = np.random.normal(0, 0.02)
memory_fluctuation = np.random.normal(0, 0.02)
security_fluctuation = np.random.normal(0, 0.01)
cost_fluctuation = np.random.normal(0, 1.0)

# Action-specific effects
if action_name == "INCREASE_RESOURCES":
    # Increasing resources boosts performance but costs more
    cpu_effect = -0.15  # Reduce CPU utilization (better performance)
    memory_effect = -0.10  # Reduce memory pressure
    security_effect = -0.02  # Slight security improvement
    cost_effect = 8.0  # Higher cost

elif action_name == "DECREASE_RESOURCES":
    # Decreasing resources saves costs but may impact performance
    cpu_effect = 0.15  # Increase CPU utilization (worse performance)
    memory_effect = 0.10  # Increase memory pressure
    security_effect = 0.02  # Slight security risk
    cost_effect = -6.0  # Lower cost

elif action_name == "MAINTAIN_RESOURCES":
    # Maintain current resource levels with minimal changes
    cpu_effect = 0.0
    memory_effect = 0.0
    security_effect = 0.0
    cost_effect = 0.0

elif action_name == "ENHANCE_SECURITY":
    # Enhance security measures
    cpu_effect = 0.05  # Slight performance impact
    memory_effect = 0.03  # Slight memory impact
    security_effect = -0.15  # Significant security improvement
    cost_effect = 5.0  # Moderate cost increase

else:  # "OPTIMIZE_COST"
    # Optimize costs but with some tradeoffs
    cpu_effect = 0.08  # Some performance impact
    memory_effect = 0.05  # Some memory impact
    security_effect = 0.03  # Slight security impact
    cost_effect = -10.0  # Significant cost reduction

# Calculate next state with both random fluctuations and action effects
next_cpu = np.clip(cpu_util + cpu_fluctuation + cpu_effect, 0.0, 1.0)
next_memory = np.clip(memory_util + memory_fluctuation + memory_effect, 0.0, 1.0)
next_security = np.clip(security_threat + security_fluctuation + security_effect, 0.0, 1.0)
next_cost = max(0, current_cost + cost_fluctuation + cost_effect)
```

Fig 6.9 - Simulation Environment

We had to play around with the reduction in memory pressure and adjust the decrease in resources for memory in the simulation because the penalty had a constant downward trend where memory was falling below 30% constantly without any inflections. But otherwise the change in resources corresponding to action values are fairly arbitrary yet somewhat logical (maintain leads to no change; enhance security leads to all performance impacts; and so on).

Let us now pivot and talk in detail about handling inferences from the LSTM and GNN models. Since this is a pytorch implementation to test out the business process module, our design principles ask for the modularity of each component, so here we will implement mock inferencing BUT use the models we defined, with just the input stream being sampled from noise and some base pattern. The complexity levels up compared to standalone implementations of the infrastructure module and the network security module but this is a good trend of the directionality of increasing complexity.

Here is how we handle the LSTM inferencing, with instantiating the loaded LSTM model and running the sequence tensor from an input sequence and noise for mock inferences (mock in practice but not in essence as the model architecture has proven itself):

```
# LSTM inference function for infrastructure prediction
def lstm_inference(lstm_model, device, sequence_length=10):
    """
    Simulates inference from the LSTM model for CPU utilization prediction.
    In a real scenario, you would use actual time series data.
    """
    # Create a synthetic sequence with realistic server load patterns
    # Libtorch integration layer will handle it in the actual implementation

    # Generate a base pattern with daily cycle
    time_points = np.linspace(0, 2*np.pi, sequence_length)
    base_pattern = 0.6 + 0.2 * np.sin(time_points)

    # Add some noise
    noise = np.random.normal(0, 0.05, sequence_length)
    sequence = base_pattern + noise

    # Ensure values are within [0, 1]
    sequence = np.clip(sequence, 0, 1)

    # Convert to tensor with proper shape (batch, seq_len, features)
    # For an enterprise implementation, I must include all 6 features but for now let's just duplicate CPU resources:
    features = np.tile(sequence.reshape(-1, 1), (1, 6))
    sequence_tensor = torch.FloatTensor(features).unsqueeze(0).to(device)

    # Pass through LSTM model
    with torch.no_grad():
        prediction = lstm_model(sequence_tensor)

    # Return predicted CPU utilization
    return prediction.item()
```

Fig 6.10 - LSTM inferencing

GNN inferencing needs to be tweaked even further than we tweaked our LSTM inferencing above (not including all 6 features for simplicity). Since our GPU VRAM is limited to 6GB, and we are loading multiple models as well as running the RL agent in real-time (ideally), let us try to smoothen the approach by categorizing severity as the closeness to num-classes-1 from a probabilistic standpoint (0-1):

```python
# GNN inference function for security threat detection
def gnn_inference(gnn_model, device, num_classes=10):
    """
    Simulates inference from the GNN model for security threat detection.
    In a real scenario, you would use actual network data.
    """
    try:
        # In an enterprise implementation, this would be constructed from my network data
        # But let's create a synthetic node feature and edge_index
        num_nodes = 10
        num_features = 42  # Based on our UNSW-NB15 dataset

        # Generate synthetic node features
        x = torch.randn(num_nodes, num_features).to(device)

        # Generate a simple edge index (fully connected graph for simplicity)
        edges = []
        for i in range(num_nodes):
            for j in range(num_nodes):
                if i != j:
                    edges.append([i, j])
        edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous().to(device)

        # Forward pass through GNN
        with torch.no_grad():
            logits = gnn_model(x, edge_index)
            probabilities = F.softmax(logits, dim=1)

            # Get the highest threat class probability
            # Classes close to num_classes-1 are considered more severe threats
            # This is a simplification - we'd like to map specific attack threats but our GPU does not have enough VRAM for this
            threat_scores = []
            for i in range(num_classes):
                # Weight higher class indices as more severe
                severity_weight = i / (num_classes - 1)
                threat_scores.append(probabilities[:, i].mean().item() * severity_weight)

            # Normalize to [0, 1]
            threat_level = sum(threat_scores) / sum(severity_weight for i in range(num_classes))
            return max(0.0, min(1.0, threat_level))

    except Exception as e:
        print(f"Error in GNN inference: {e}")
        # Return a random threat level if inference fails
        return random.uniform(0.1, 0.3)
```

Fig 6.11 - GNN inferencing.

Finally, we run training for 100 episodes with 30 steps per episode. For initial values, cpu utilization is set at 70%, memory util at 65%, security threat at 0.2 probability and current cost at 100 (simple numeric value instead of cost per GHz, etc.). Here is our main training loop and it uses all the functionalities we have discussed above with simulation step, reward computation and updating target network (every 5 steps):

```
    for step in range(steps_per_episode):
        # Get predictions from LSTM and GNN models
        try:
            predicted_cpu = lstm_inference(lstm_model, device)
            predicted_threat = gnn_inference(gnn_model, device)
        except Exception as e:
            print(f"Error during model inference: {e}")
            predicted_cpu = cpu_util * (1 + np.random.normal(0, 0.05))
            predicted_threat = security_threat * (1 + np.random.normal(0, 0.05))

        # Blend current state with predictions (smoothing)
        cpu_util = 0.7 * cpu_util + 0.3 * predicted_cpu
        security_threat = 0.7 * security_threat + 0.3 * predicted_threat

        # Ensure values are within valid ranges
        cpu_util = max(0.0, min(1.0, cpu_util))
        security_threat = max(0.0, min(1.0, security_threat))

        # Create state tensor for RL agent
        state = agent.get_state(cpu_util, memory_util, security_threat, current_cost)

        # Select action using epsilon-greedy policy
        action_idx = agent.select_action(state)
        action_name = agent.actions[action_idx]

        # Simulate environment step
        next_cpu, next_memory, next_security, next_cost = simulate_environment_step(
            cpu_util, memory_util, security_threat, current_cost, action_idx, agent.actions
        )

        # Create next state tensor
        next_state = agent.get_state(next_cpu, next_memory, next_security, next_cost)

        # Compute reward
        reward = agent.compute_reward(state, action_idx, next_state)
        episode_reward += reward

        # Store transition and train
        agent.store_transition(state, action_idx, reward, next_state)
        loss = agent.train_step()

        # Update environment state
        cpu_util = next_cpu
        memory_util = next_memory
        security_threat = next_security
        current_cost = next_cost

        # Periodically update target network
        if step % update_target_every == 0:
            agent.update_target_network()

        # Logging
        if step % 5 == 0:
            print(f"Episode {episode+1}/{num_episodes}, Step {step}, " +
                  f"Action: {action_name}, Reward: {reward:.2f}, " +
                  f"CPU: {cpu_util:.2f}, Security: {security_threat:.2f}, " +
                  f"Cost: {current_cost:.2f}, Epsilon: {agent.epsilon:.2f}")

    # Track episode metrics
    episode_rewards.append(episode_reward)
    cpu_utilizations.append(cpu_util)
    security_threats.append(security_threat)
    costs.append(current_cost)

    print(f"Episode {episode+1}/{num_episodes} finished. Total reward: {episode_reward:.2f}")
```

Fig 6.12 - Main Training Loop

Here is a sample of the training loop:

```
Episode 14/100, Step 0, Action: OPTIMIZE_COST, Reward: 0.50, CPU: 1.00, Security: 0.18, Cost: 89.50, Epsilon: 0.19
Episode 14/100, Step 5, Action: OPTIMIZE_COST, Reward: 0.20, CPU: 1.00, Security: 0.17, Cost: 38.63, Epsilon: 0.19
Episode 14/100, Step 10, Action: DECREASE_RESOURCES, Reward: -20.00, CPU: 1.00, Security: 0.16, Cost: 0.00, Epsilon: 0.18
Episode 14/100, Step 15, Action: DECREASE_RESOURCES, Reward: -20.00, CPU: 1.00, Security: 0.06, Cost: 0.00, Epsilon: 0.18
Episode 14/100, Step 20, Action: INCREASE_RESOURCES, Reward: -40.00, CPU: 0.85, Security: 0.08, Cost: 6.64, Epsilon: 0.17
Episode 14/100, Step 25, Action: OPTIMIZE_COST, Reward: -20.00, CPU: 1.00, Security: 0.09, Cost: 0.00, Epsilon: 0.17
Episode 14/100 finished. Total reward: -383.36
```

Fig 6.13 - Training Sample

Ideally, we would like to buffer these episodes as a second for each time slice unit (for example, each second is a day of monitoring data), with the monitor showing predictions alongside recommendations. We saved the model and the best weights, but here are the results:

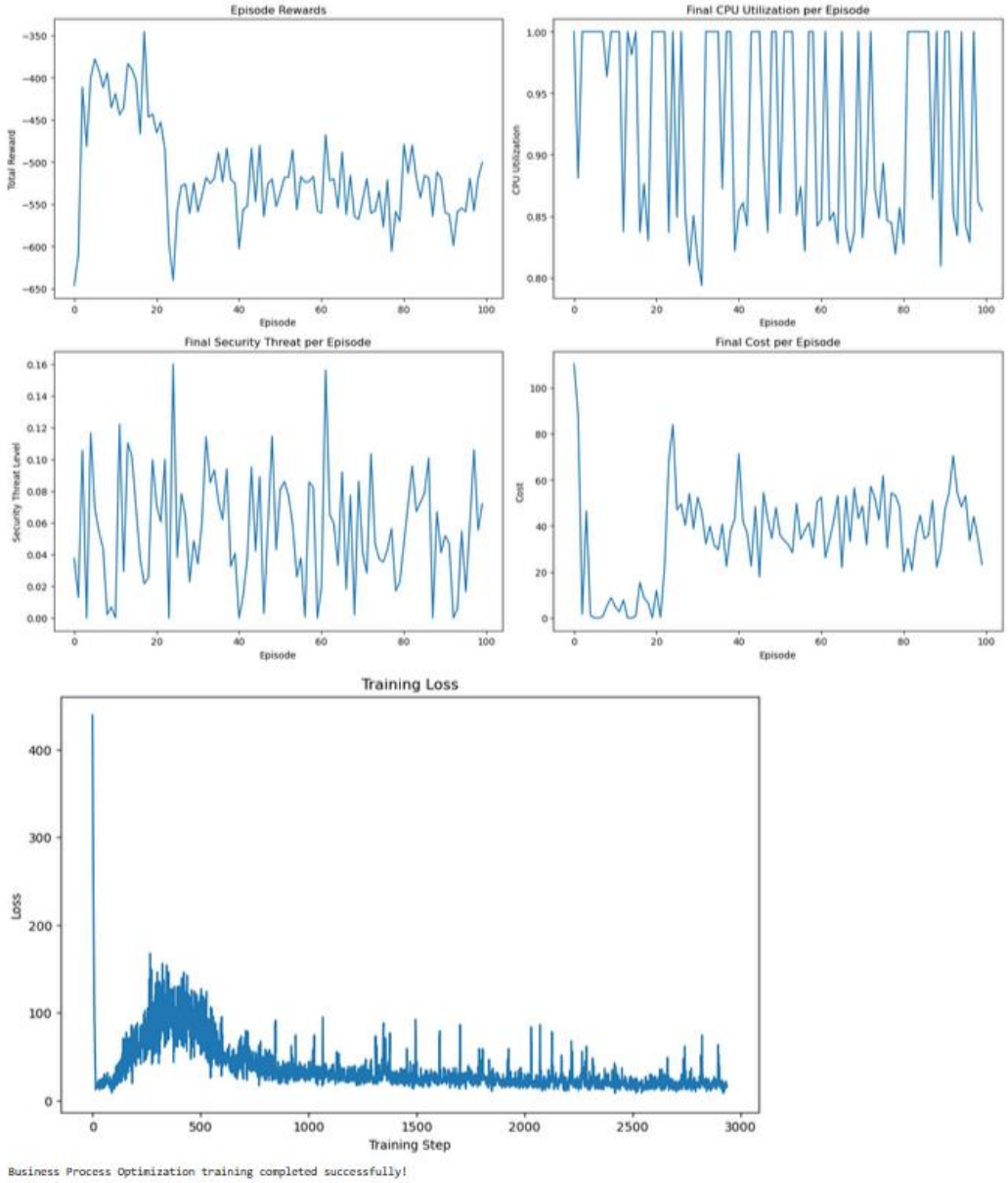


Fig 6.14 – RL Agent Results

If you look at the rewards, they start very low at around 650 and improve in the first 20 episodes before stabilizing mid-way in the graph so we have learnt a good policy but it hasn't been fully optimized so far. Even CPU utilization hovers around 80% and 100% which is better than low utilization at the very least so resource allocation strategies are being heavily exploited. Final costs stabilize between 20-60 which is a good correlation with the reward function if you look at the 2 graphs side by side.

So our agent seems to be modestly decent at maximizing CPU efficiency while also minimizing threats + costs.

But what are some strategies we can outline as part of our design philosophy were we to scale up and improve the performance of our RL agent? We can extrapolate from our graphs that perhaps adding more weight to CPU stability in the reward function can help, given the constraint being consistent required utilization rate. Decision making needs to be completely overhauled and greater temperature as well as nuance can be added to business constraints which are simplistic and arbitrary right now for numerical consistency as complexity might not show logically consistent results. Security threat prediction is the most lacking part since our original GNN module doesn't take all fields into account and only uses 3 -- protocol, state and service. Maybe we can consider a moving average to reduce FPs in our network threat detection.

All in all, actual LSTM predictions and GNN security analyses from our previous chapters can be used to stress test our design philosophy.

Chapter 7 - Integration Layer

At this point we have successfully implemented each module we set out to in our introduction:

1. Infrastructure Prediction using LSTM: metrics -> LSTM -> CPU predictions -> RL Agent
2. Network Security Detection using GNN (GCN): Network data -> GNN -> Threat analysis -> RL Agent
3. Business Process Module using RL (MAB): RL decisions -> system adjustments -> AI recommendations

Our project setup was invariably complex (see Chapter 1), with the focus on inferencing using a libtorch backbone using the C++ infrastructure. Each CMakeLists.txt file for each directory caused cascading build errors where either torch.h could not be found or include errors for hpp files. Besides our include files, we had (and this is but a small snippet of all the directories):

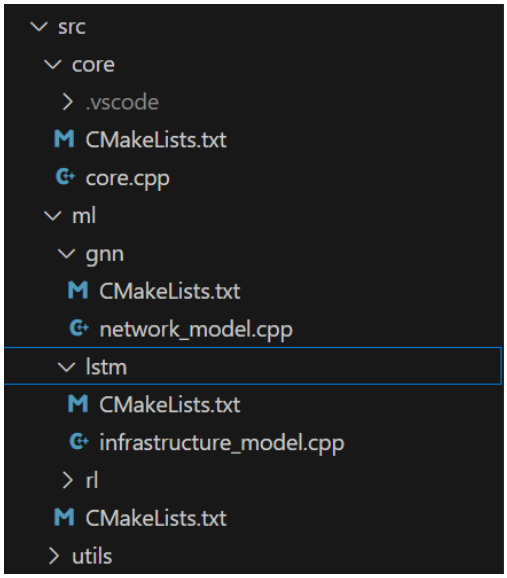


Fig 7.1 - project directory snippet

We are using Cmake to build the project files (https://cmake.org/documentation/) and our prelim build was successful:

```
-- USE_CUDNN is set to 0. Compiling without cuDNN support
-- USE_CUSPARSELT is set to 0. Compiling without cuSPARSELt support
-- USE_CUDSS is set to 0. Compiling without cuDSS support
-- USE_CUFILE is set to 0. Compiling without cuFile support
-- Autodetected CUDA architecture(s):  8.6
-- Added CUDA NVCC flags for: -gencode;arch=compute_86,code=sm_86
-- Found Torch: C:/Users/91965/Downloads/libtorch-win-shared-with-deps-2.5.1+cu124/libtorch/lib/torch.lib
-- Configuring done (13.9s)
-- Generating done (0.2s)
-- Build files have been written to: C:/Users/91965/Downloads/Enterprise-AI-Monitor-daabb6dcd140c5e5ff0fd8549802a9162520e5a3/Enterprise-AI-Monitor/build
PS C:\Users\91965\Downloads\Enterprise-AI-Monitor-daabb6dcd140c5e5ff0fd8549802a9162520e5a3\Enterprise-AI-Monitor\build>
```
Fig 7.2 - build successful

Version control made it possible to rollback to earlier releases in case we ran into errors or builds failed:

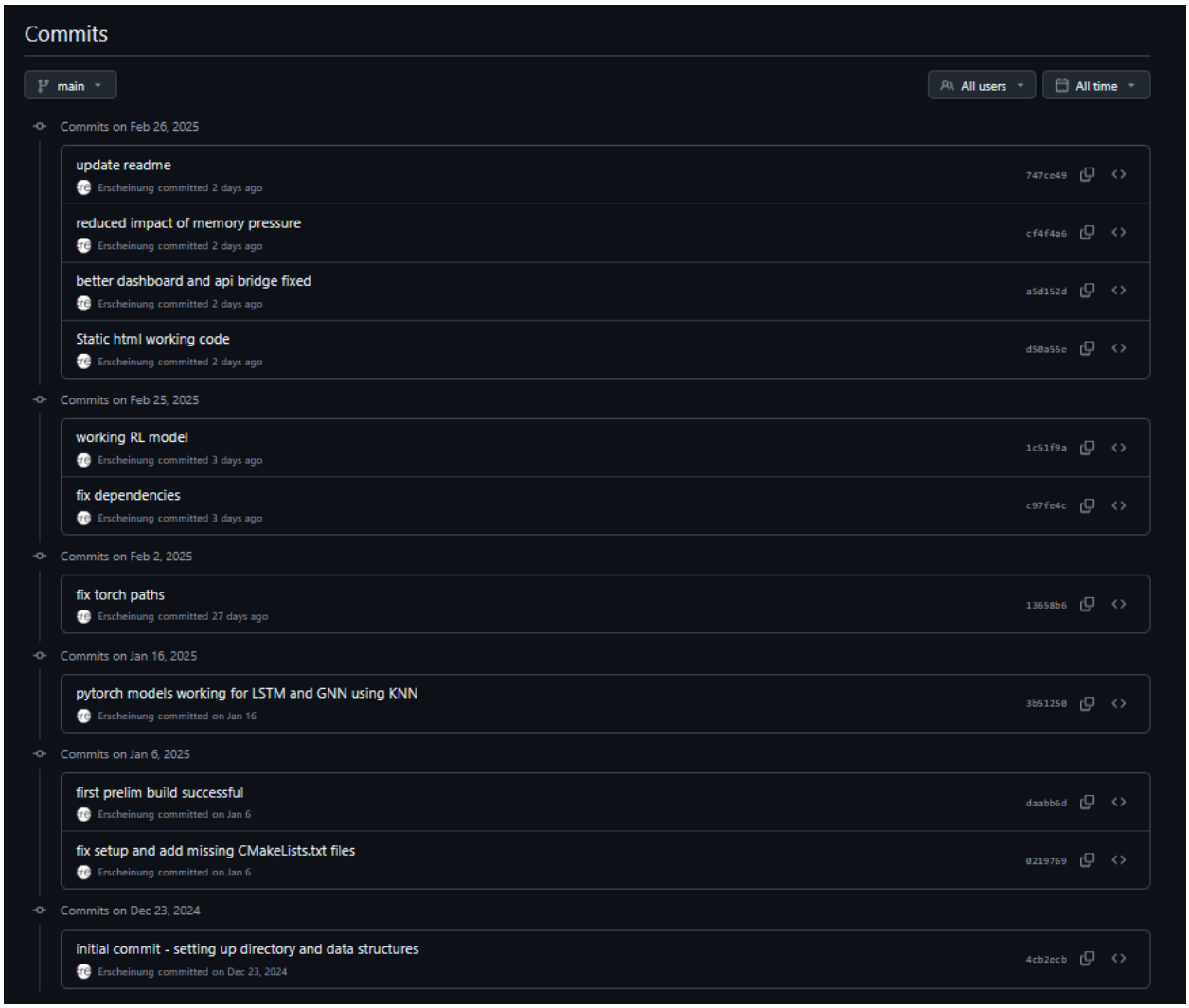


Fig 7.3 - Version control history

Although a new problem was emerging--the project setup had multiple cmake --build . failures due to cpp file dependency problems, and more so at the data ingestion layer for monitoring_types. Path dependency problems and dependency mismatches wreaked havoc and prevented a working prototype to be deployed using our existing setup.

All these problems were addressed by a singular API Bridge that further decouples our concerns: keep ML models in python where they're easier to develop and maintain but remove libtorch integration for ML models and just use C++ as a REST API interface for a clean, agnostic approach. And for the sake of the prototype, this must suffice as we gradually migrate to a C++ backbone as components mature.

So our header file can focus on helper methods and insight generation:

```cpp
// API Bridge class declaration
class EnterpriseAIBridge {
private:
    std::string baseUrl;
    void* curl;
    bool isConnected;

    // Helper methods to perform HTTP requests
    bool performGet(const std::string& url, std::string& response);
    bool performPost(const std::string& url, const std::string& data, std::string& response);
    double calculateAverage(const std::vector<double>& values);

    // Data generation for mock mode for our monitor
    json generateMockData();

    // Insight generation
    void generateInsights(
        const std::vector<double>& cpuData,
        const std::vector<double>& memoryData,
        const std::vector<double>& securityData,
        const std::vector<double>& costData
    );

    // LibTorch integration method
    void analyzeWithLibTorch(
        const std::vector<double>& cpuData,
        const std::vector<double>& memoryData,
        const std::vector<double>& securityData,
        const std::vector<double>& costData
    );
```

Fig 7.4 - API BRIDGE helper functions

While the central focus on C++ backbone lies in its API nature rather than an inferencing backbone (with future libtorch functionality added):

```cpp
// Key components of the API Bridge
class EnterpriseAIBridge {
    private:
        std::string baseUrl;
        CURL* curl;
        bool isConnected;

    public:
        // Constructor establishes connection to Python server
        EnterpriseAIBridge(const std::string& url = "http://localhost:5000/api");

        // API endpoints for monitoring and control
        json getStatus();
        json getSimulationData();
        bool startSimulation();
        bool stopSimulation();

        // Data analysis for C++ integration
        void analyzeData(const json& data);

        // LibTorch integration point for future expansion
        void analyzeWithLibTorch(
            const std::vector<double>& cpuData,
            const std::vector<double>& memoryData,
            const std::vector<double>& securityData,
            const std::vector<double>& costData
        );
    };
```

Fig 7.5 - API BRIDGE key components

Our cpp implementation handles the bridge creation, starts the simulation and stream input generation, handles POST requests, monitoring simulations and so on. Here is where we see an instance of the MAB actions and penalties:

```cpp
// Pick a random action
std::string action_options[] = {
    "INCREASE_RESOURCES", "DECREASE_RESOURCES", "MAINTAIN_RESOURCES",
    "ENHANCE_SECURITY", "OPTIMIZE_COST"
};
std::uniform_int_distribution<> action_dist(0, 4);
std::string action = action_options[action_dist(gen)];
actions.push_back(action);

// Generate a plausible reward based on the action and state
double reward = 0.0;
if (action == "INCREASE_RESOURCES" && cpu > 0.8) {
    reward = 5.0 + std::uniform_real_distribution<>(-2.0, 2.0)(gen);  // Good if CPU was high
} else if (action == "DECREASE_RESOURCES" && cpu < 0.4) {
    reward = 5.0 + std::uniform_real_distribution<>(-2.0, 2.0)(gen);  // Good if CPU was low
} else if (action == "ENHANCE_SECURITY" && security > 0.5) {
    reward = 10.0 + std::uniform_real_distribution<>(-3.0, 3.0)(gen); // Good if security threats high
} else if (action == "OPTIMIZE_COST" && cost > 110.0) {
    reward = 8.0 + std::uniform_real_distribution<>(-2.0, 2.0)(gen);  // Good if costs were high
} else {
    reward = std::uniform_real_distribution<>(-5.0, 5.0)(gen);  // Random for other cases
}
rewards.push_back(reward);
```

Fig 7.6 - MAB actions in bridge API

Our bridge uses libcurl [34] for HTTP requests to the Python API, nlohmann/json for parsing and serialization [35] and Standard C++ Threading for async operations.

Now that we have our API bridge, we can set up a Python Server that applies a PROD ready application architecture--note that this is much more than just a reimplementation of our jupyter notebook for our Business Process Module. Our python server has several layers:

1. Define Models: existing classes for each isolated architecture

```python
# Define model classes (like in our notebook)
class AttentionLayer(nn.Module): ···

class InfrastructureLSTM(nn.Module): ···

class GCN(torch.nn.Module): ···

# Simple DQN for the RL agent
class DQN(nn.Module): ···

# Business RL Agent
class BusinessRLAgent: ···
```

Fig 7.7 - Defining Models

2. Load Models: load saved models from test runs.

```
# Initialize Flask app
app = Flask(__name__)
CORS(app)  # Enable CORS for all routes

# Global variables for simulation
> simulation_data = {···

# Model paths
LSTM_MODEL_PATH = "infrastructure_lstm_model.pt"
GNN_MODEL_PATH = "gnn_model_actual_v2.pt"
RL_MODEL_PATH = "business_rl_model_final.pt"

# Model loading functions
> def load_infrastructure_model(lstm_path=LSTM_MODEL_PATH):···

> def load_gnn_model(gnn_path="gnn_model_actual_v2.pt", device='cpu'):···

> def load_rl_model(rl_path=RL_MODEL_PATH):···

> def gnn_inference(input_data=None):···
```

Fig 7.8 - Load Models

3. Handle Inferencing: run predictions on loaded models.

```
> def load_gnn_model(gnn_path="gnn_model_actual_v2.pt", device='cpu'):···

> def load_rl_model(rl_path=RL_MODEL_PATH):···

> def gnn_inference(input_data=None):···

# LSTM inference
> def lstm_inference(model, sequence_data=None):···

# Environment simulation for the RL agent
> def simulate_environment_step(cpu_util, memory_util, security_threat, current_cost, action_idx, action_names):···

# Function to compute reward (simplified version for simulation)
> def compute_reward(state, action, next_state):···

# Load models at startup
lstm_model = None
rl_agent = None
gnn_model = None
```

Fig 7.9 - Handle Inferencing

4. Simulation Engine: a mock engine that buffers real-time input--our API bridge handles this.

```
> def init_models():···

# Simulation thread
simulation_running = False
simulation_thread = None

> def run_simulation():···
```

Fig 7.10 - Simulation Methods

5. API Layer for Flask endpoints: exposing functionality to view, route methods.

```python
# API Routes
@app.route('/')
def index():···

@app.route('/api/start-simulation', methods=['POST'])
def start_simulation():···

@app.route('/api/stop-simulation', methods=['POST'])
def stop_simulation():···

@app.route('/api/simulation-data', methods=['GET'])
def get_simulation_data():···

@app.route('/api/status', methods=['GET'])
def get_status():···

# Initialize models when app starts
@app.before_first_request
def before_first_request():···

if __name__ == '__main__':
    # Initialize models now in case we're using gunicorn or similar
    init_models()
    # Run the Flask app
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Fig 7.11 - API Routes

We rehauled the code structure since a progressive enhancement ties up well into our design philosophy rather than a monolithic project structure. The detailed status of API endpoints lends its internal state for monitoring and debugging, as we can see the JSON structure during live runs:

```
Loaded full LSTM model
Could not load GNN model: No module named 'torch_geometric.inspector'
Failed to load GNN model: No module named 'torch_geometric.inspector'
Created new GNN model as fallback
Model loaded from business_rl_model_final.pt
Models loaded
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.20.10.3:5000
Press CTRL+C to quit
 * Restarting with stat
Loading models...
C:\Users\91965\anaconda3\envs\tf_gpu_env\lib\site-packages\torch\serialization.py:781: UserWarning: 'torch.load' received a zip file that looks like a TorchScript archive dispatching to 'torch.jit.load'
 (call 'torch.jit.load' directly to silence this warning)
  " silence this warning)", UserWarning)
Loaded full LSTM model
Could not load GNN model: No module named 'torch_geometric.inspector'
Failed to load GNN model: No module named 'torch_geometric.inspector'
Created new GNN model as fallback
Model loaded from business_rl_model_final.pt
Models loaded
 * Debugger is active!
 * Debugger PIN: 804-085-725
```

Fig 7.12 - Running python server

Here's our json before beginning simulation:

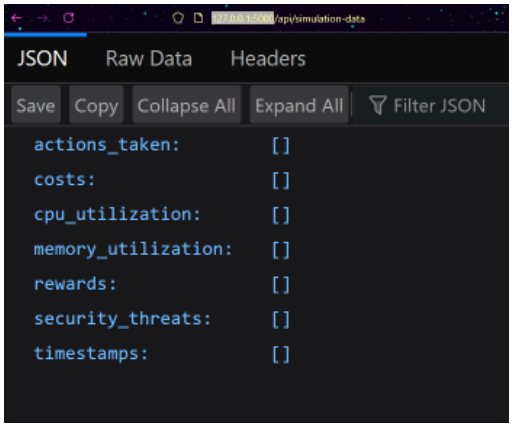


Fig 7.13 - Before Starting Simulation

And it starts filling up as the simulation is started:

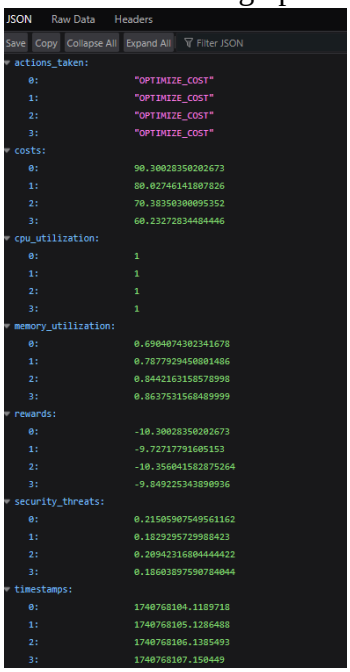


Fig 7.14 - During Simulation

This means our server interface is working as expected, and we can export this json as well in case enterprises require telemetry-collects. The clean and observable nature of our metrics mean that visualization is built into the dashboard and there is no need for specialized functions to parse the metrics from the json as it is the case for raw outputs (like from Flexible I/O Tester [36]) that are rolled into a tarball later on.

Next, we will attempt to visualize in real time or even a static simulation of this RL process as a monitoring dashboard.

Chapter 8 - Central Monitor

The vision for a central monitoring system was inspired by the Live Optics dashboard [37]:

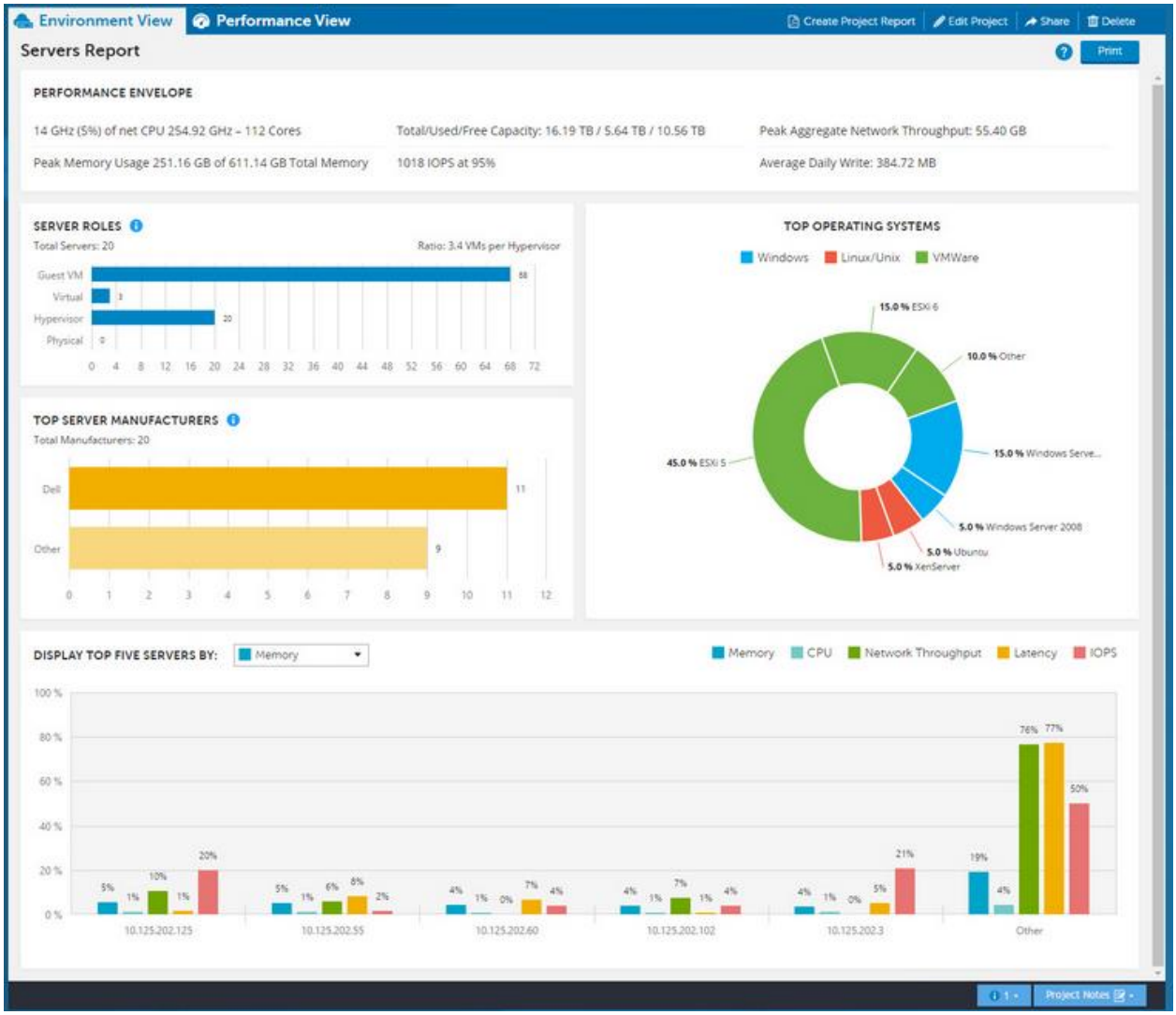


Fig 8.1 - Live Optics Dashboard

Ideally, a graph or a pie chart showing CPU metrics, threat levels and finally business process recommendations. We already have a python server collecting these metrics in a json and we need a way to visualize these metrics in a real time scenario. In that case, we can buffer 100 or so predictions, 1 per second, where each second can represent a day or maybe an actual real-time environment.

It could be confusing to display a prediction chart instead of a real-time monitoring chart. Almost certainly counter-intuitive were we to select the former approach. So the prediction models are running in the background, feeding the RL model which produces insights or recommendations.

The goal was to deploy this as a React web application but since this is a single page, it would suffice to render a static Flask page but we also provide instructions to deploy a React app, as well as include provisions for it in our codebase.

Our first-approach tried to use C++ with external visual libraries but this was not feasible as real-time updates are difficult to implement given the lack of interactive visualization capabilities in C++ as well as integration proving complicated with web technologies.

So our second approach used python visualization with matplotlib as a static result. But this is antithetical to the key idea of a monitoring system: eyes that see the world as it moves; not PIT snapshots.

Finally, we got it right with our third approach--real-time API integration from our previous chapter. We have consistent JSON structures, which makes it easier to create the visualization layer with time-series charts and health indicators alongside AI recommended predictions (action type).

What this means is that we make cause-effect relationships apparent:

Action -> State Change -> Reward -> Next Action

We have created a new subdirectory in our project folder for this task:

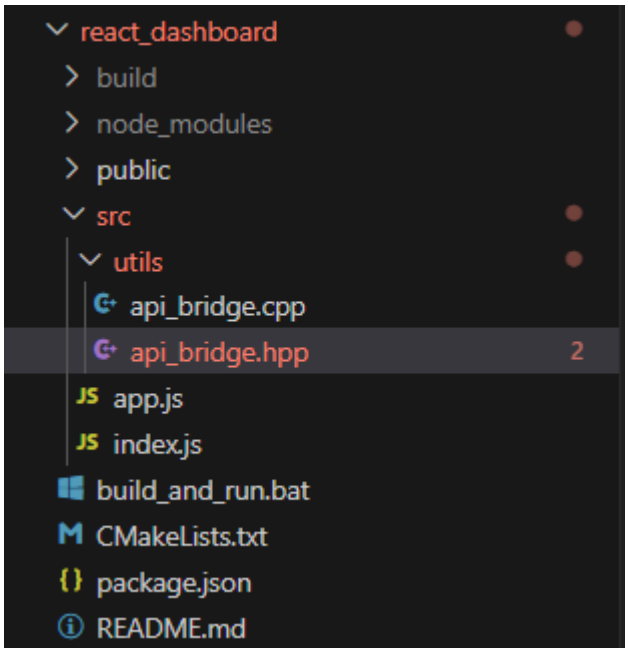


Fig 8.2 - Dashboard directory

The build_and_run batch file handles the dashboard from building the project to running the API bridge:

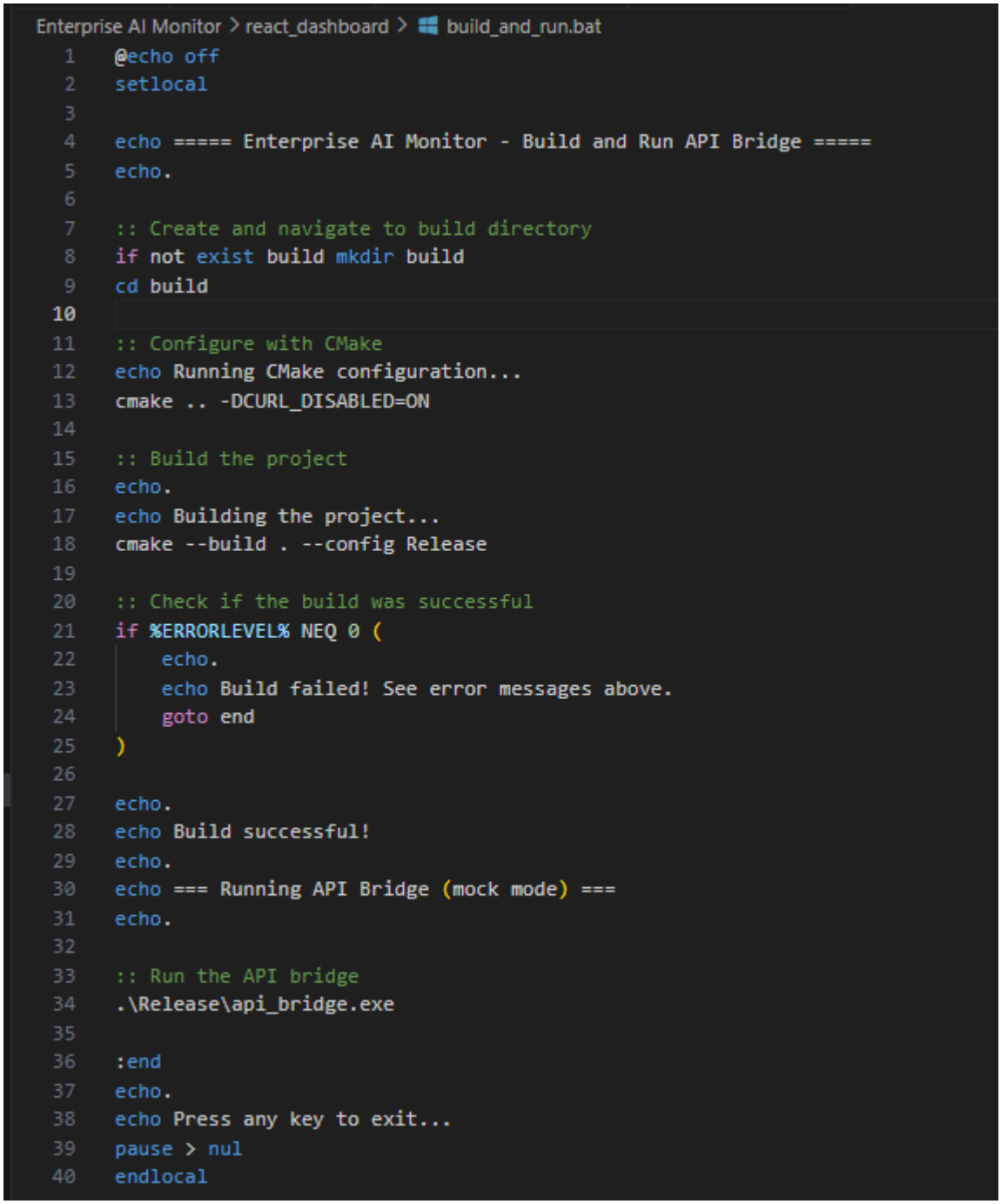


Fig 8.3 - Build and Run Batch File

Note that we can run it in a mock mode for demo purposes by setting the -DCURL_DISABLED flag to ON. We also created a README where you can build it manually:

````
1. Include the header file:
   ```cpp
   #include "api_bridge.hpp"
   ```

2. Create an instance of the bridge:
   ```cpp
   EnterpriseAIBridge bridge("http://localhost:5000/api");

   if (!bridge.isReady()) {
       std::cerr << "Failed to initialize API bridge" << std::endl;
       return 1;
   }
   ```

3. Use the API methods:
   ```cpp
   // Get system status
   json status = bridge.getStatus();

   // Start simulation if not running
   if (status["status"] != "running") {
       bridge.startSimulation();
   }

   // Get simulation data
   json data = bridge.getSimulationData();

   // Analyze the data
   bridge.analyzeData(data);
   ```
````

Fig 8.4 README snippet

This flexibility is very useful as we can create CI/CD pipelines and automate builds and deployments. Right now there is no cloud provider configured as we are not trying to deploy it as an application on a real datacenter environment. That can be a topic of research where the API centric approach could yield good results.

The web dashboard features real-time data visualization with Chart.js and here is what our first iteration looked like:

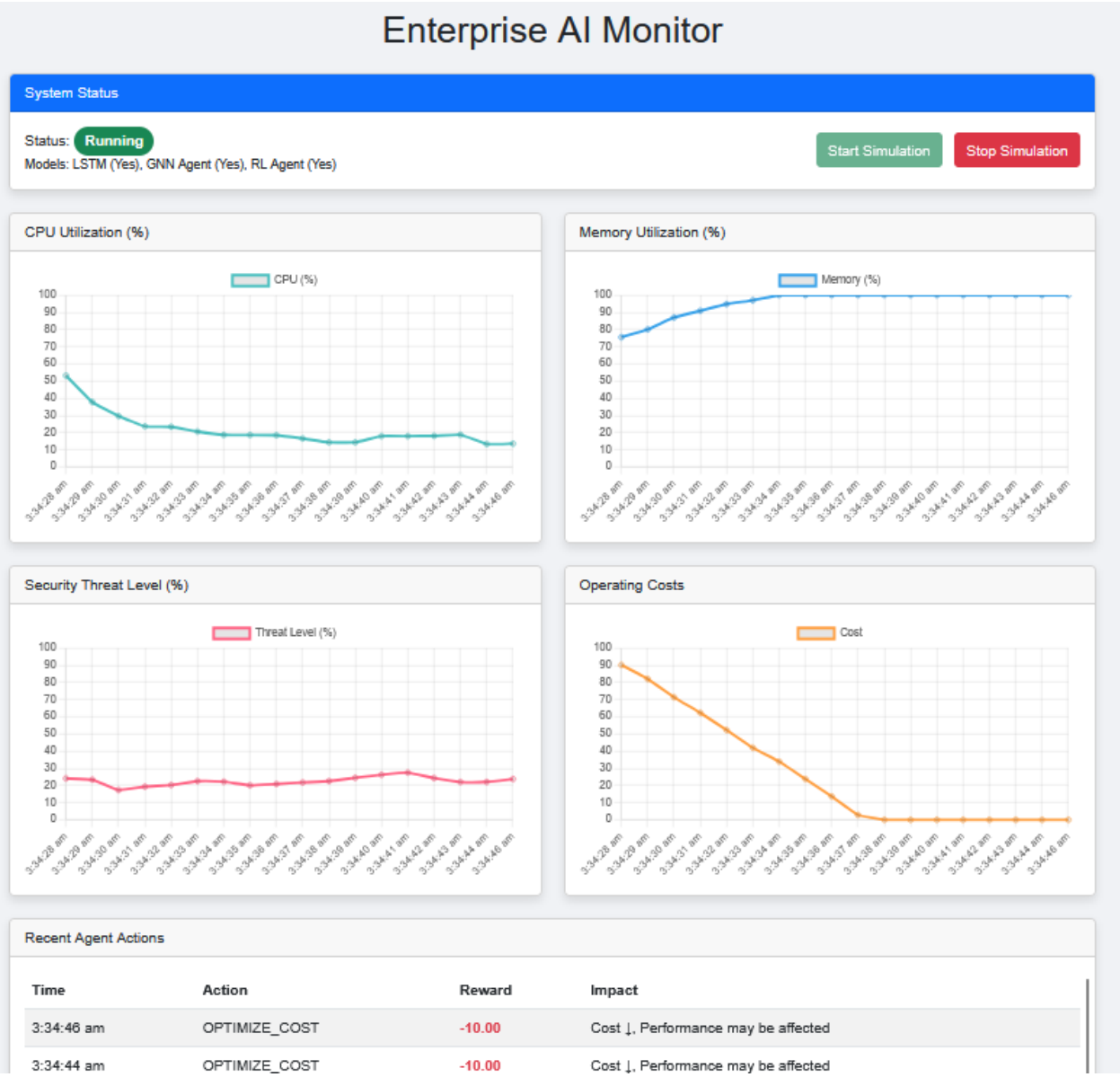


Fig 8.5 - Dashboard v1

As you can see, we have CPU and memory utilization graphs as well as threat level % -- LSTM and GNN models are used in the backend here while the RL model is optimizing all of these and driving down operating costs. Recent Agent Actions are also logged for the user's/admin's convenience. We can start and stop the simulation at any time. We can also curl into the routes or put those routes in the browser to capture the live metric collection in the json file (see previous chapter for flask routes). But there leaves a lot to be desired from this dashboard. It is too 'raw', in a sense. Not very intelligent, which is what our vision is, for an intuitive dashboard.

Keeping these points in mind, we tried to add some structure and smoothen the user experience, with our second iteration being thus:

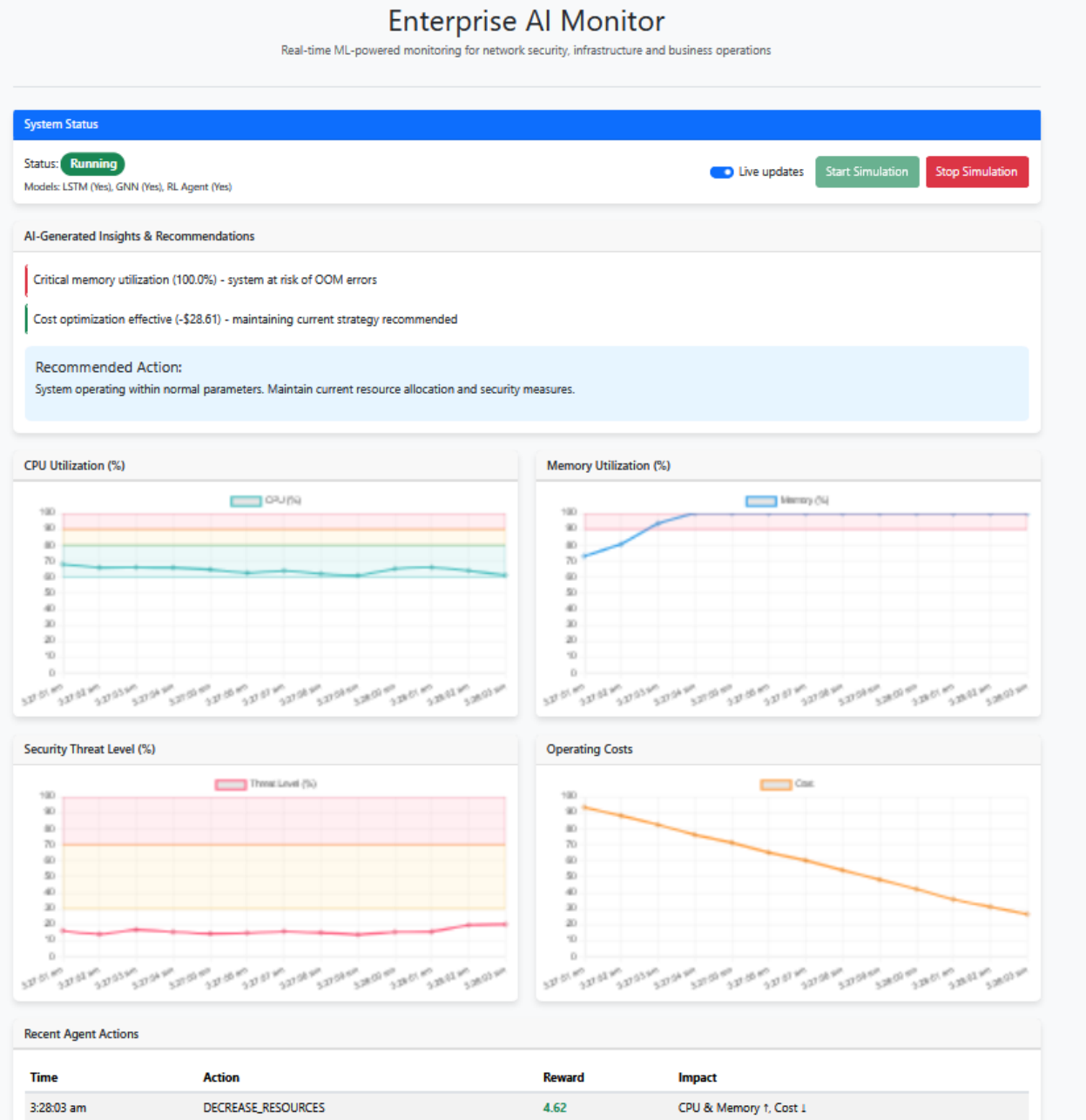


Fig 8.6 - Dashboard v2

Here we can pause the simulations, as well as having a more intelligent recommendation system instead of mechanical and dry agent rewards and actions that we see below. There is a health alert with recommended actions which is what the model predictions really come in handy for. We have also added visual upgrades in the charting for intuitive detection safe/degraded/risky states with the yellow, green and red thresholds. Memory pressure was also decreased as the penalties proved too much.

Finally, here is our third and final iteration with a more professional monitoring dashboard with significantly improved graphs and minor UI changes:

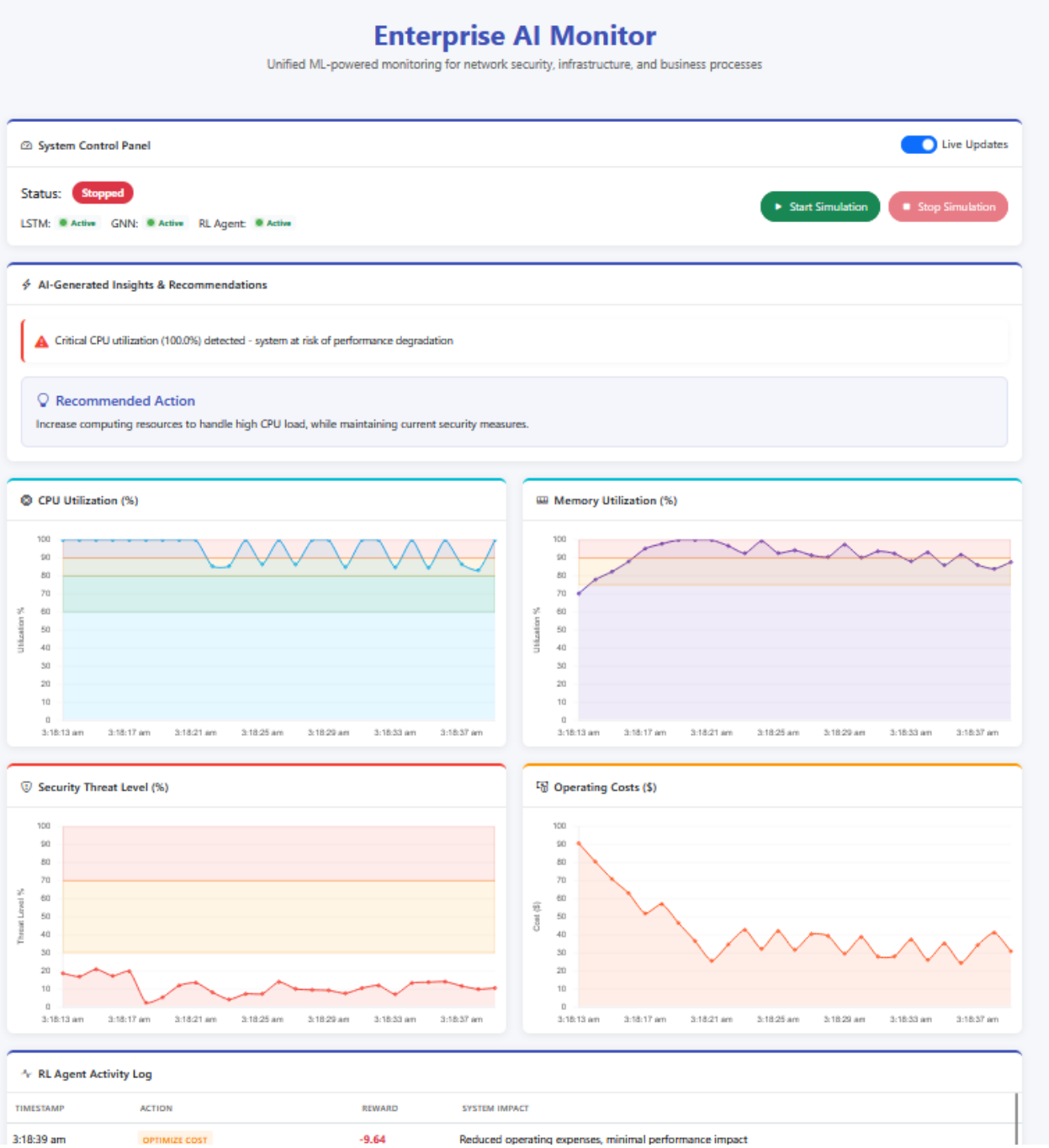


Fig 8.7 - Dashboard v3

Recommended actions can be catalogued and even flagged (thumbs-up/down) for adding bias to the RL agent, or even having one's own business constraints as here we only have basic numerical constraints devoid of any economic practices or business applications.

We have created a proactive monitoring system where dashboard is intuitive and informative, but not too detailed and yet we can see the activity Log of the agent (this can be renamed for professional reasons but we have kept it as it is for academic purposes), as well as recommended actions. Creating APIs for different storage systems and compute from different vendors can be the first-step to creating a holistic AI monitoring system where more complex algorithms or a DIY RL approach with user constraints can come into play to model real and clearly defined business outcomes.

Chapter 9 – Conclusion and Future Work

We sought out to bridge isolated monitoring solutions by creating our unified architecture across infrastructure predictions, network security analysis and business process optimizations. However, the goal was not to build a robust prototype but by trial and error, validate our ideas and intuit fundamental design principles for enterprise AI monitoring solutions. Here's what we accomplished:

1. Infrastructure Prediction Module: LSTM based predictions for CPU utilization, with 5-6% error rates modelling real-time average volatility in datacenters. We predict these CPU usage percentages to create a resource management strategy rather than reactive responses.
2. Network Security Module: We implemented a GNN based threat prediction model (using GCN) to output the probability of a predicted attack pattern. This goes beyond rule-based or database solutions currently in the market.
3. Business Process Optimization: We created an RL agent that makes (experientially 'intelligent') tradeoffs between performance security and cost.

Further yet, we used a modular hybrid architecture to develop this idea--python for model developments, C++ for system integrations and API structure, and a web-based interface to visualize insights and recommendations.

9.1 Key Design Principles

We have discovered some rather valuable design principles during the process of refining, fixing and also sometimes throwing away architectural parts. This has been a learning experience and these design principles are the intended takeaways from this study, rather than the prototype implementation (which should be taken as a Proof of Concept):

- Modularity: There is a reason why monitoring solutions are highly specialized in the market--they work well. We must allow each isolated monitoring solution (in our case, modules) to focus on what it does best to improve maintenance and velocity of development.
- API based integration: A central backbone can, while trying to offload complexity, ironically become monolithic itself. Define clear interfaces, not centralized structures, for information to flow and components to work freely.
- Graceful Degradation: Failback mechanisms can be as simple as loading mock inferences or alternative models. If the monitoring system itself is not resilient, how can it support or help in making enterprise workloads resilient?
- Observability built into the Design: Make monitoring a monitoring solution transparent and simple; a fractal self-similarity must persist within systems that the system is trying to solve for, otherwise complexity arises exponentially. We use API routes that store monitoring info and simulation variables within the monitoring application run, that can be seen in real time. Troubleshooting is key to monitoring well.
- Evaluation of trade-offs: The RL agent might make tradeoffs based on numeric algorithms, but this is the key business constraint that makes an intelligent agent 'intelligent'--the objective function is not a cold, static target but a complex, evolving, organic dynamic influenced (in our world) by economic theories, individualized business practices and policies, proactive cost management, behavioral psychology, statistical modelling, and much more. This must be thought

about deeply and even tested out in simulations to verify the directionality of the approach before applying it ex-ante.

9.2 Future Directions

Alongside design philosophy, our implementation remains, after all, a prototype. It is not an optimized application, it merely 'works'. So here are some ideas to flesh it out into a fully fledged application or mature the stack eventually:

9.2.1 Model Enhancements:

LSTM: Transformer architectures could improve the simple attention mechanism we have implemented currently, probably for more complex workload patterns. Our dataset is from 2017, and technology and workloads have indeed come a long way since then.

GNN: This is a young area of research, but graph structures like temporal GNs might capture the evolution of threats over time [38]. This would be a good middle way between database or dictionary based approaches combined with anomaly detection that AI provides.

RL: We have tested the MAB approach. But algorithms like Proximal Policy Optimization [39] or Soft Actor-Critic [40] could be tested out to see what insights they bring to the table, or how aggressive or rigid their simulations are. This is the module where we have stuck to the most classic or conservative approaches, given the complexity of business constraints.

9.2.2 Scalability

Libtorch implementation can be refined for real-time prediction scenarios instead of buffered input streams that are inherently static in nature and only artificially buffered. An enterprise workload has multiple production or development clusters, with multiple controller nodes. Monitoring could also share this structure for simplicity, and this would require a distributed architecture with a central and local monitoring deployment, where global recommendations can provide strong insights about the cluster itself. However, perhaps these workloads are mission critical, and in this case monitoring overheads will need to be minimized--these optimizations should also be present in the monitoring solution.

9.2.3 Visualization

A React application with component-based architecture would be in line with our modular design philosophy and easier maintenance. Currently, we have the ability to pause live updates, but a 'what if' scenario could be very powerful to understand agent decisions or even simulate real world scenarios in order to take proactive measures. Finally, advanced filtering and specific monitoring dashboards--essentially customizations to configure visualizations and create reports in PDF or PPT formats, with spreadsheet exports.

9.2.4 API-Centric Architecture

The current python server is a monolithic structure. We can further break it down into microservices for scaling and resilience. The dashboard is also too 'noisy', perhaps event-driven updates [41] can add more weight to recommendations rather than a polling system. API definitions will also need to be formalized for simpler maintenance and further development.

9.3 Final Thoughts

This project showed us that you cannot just mash together different monitoring approaches and have another AI model process them in a certain way to apply user constraints and give recommendations. A thoughtful system design with strong basic fundamentals is also not enough. Thorough testing is also just a part of the puzzle, to figure out which algorithms work better. Visualization is not a 'technical hurdle', as it merely shows metrics that already 'work' or are good enough from the application standpoint.

It shows us that an intelligent enterprise monitoring system should itself use the principles that it wishes to apply in its monitoring of enterprise systems. It must act in a multi-faceted manner like businesses do, it must select an input stream that is 'more important' and it must recommend in a way that the recommendation makes sense and is 'required', not merely recommending for the sake of the program, every second or every day, etc. If the user wishes to optimize cost, it must ask why? It must have agency to critically assess the user's constraints and warn them. It must recommend actions even at the pre-monitoring phase. Our Enterprise AI monitor represents a very tiny baby step in this direction, pointing the way towards more adaptable and intelligent monitoring solutions.

Bibliography

[1] "Splunk AI," [Online]. Available: https://www.splunk.com/en_us/solutions/splunk-artificial-intelligence.html.

[2] "Dynatrace vs Splunk," [Online]. Available: https://www.dynatrace.com/monitoring/platform/comparison/dynatrace-vs-splunk/?utm_campaign_id=16272465259&gclsrc=aw.ds&gad_source=1#whitepepar.

[3] "GCT," [Online]. Available: https://github.com/google/cluster-data.

[4] "UNSW-NB15," [Online]. Available: https://research.unsw.edu.au/projects/unsw-nb15-dataset.

[5] R. Zhao, J. Wang, R. Yan and K. Mao, "Machine health monitoring with LSTM networks," 2016.

[6] A. T. B. H. N. N. Andy Brown, "Recurrent Neural Network Attention Mechanisms for Interpretable System Log Anomaly Detection," 2018.

[7] C. Parera, Q. Liao, I. Malanchini, D. Wellington, A. E. C. Redondi and M. Cesana, "Transfer Learning for Multi-Step Resource Utilization Prediction," 2020.

[8] N. A. G. a. A. S. Weigend, "The Future of Time Series," 1993.

[9] "Nagios Wiki Page," [Online]. Available: https://en.wikipedia.org/wiki/Nagios.

[10] "Zabbix Wiki Page," [Online]. Available: https://en.wikipedia.org/wiki/Zabbix.

[11] M. Brian K. Nelson MD, "Time Series Analysis Using Autoregressive Integrated Moving Average (ARIMA) Models," 2008.

[12] "Snort," [Online]. Available: https://www.snort.org/.

[13] "Wireshark Manual," [Online]. Available: http://cet4663c.pbworks.com/w/file/fetch/62450910/4663_Wireshark_manual.pdf.

[14] N. Moustafa and J. Slay, "The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," 2016.

[15] S.-H. H. H.-P. S. Wun-Hwa Chen, "Application of SVM and ANN for intrusion detection," 2005.

[16] G. C. S. H. Z. Z. C. Y. Z. L. L. W. C. L. M. S. Jie Zhou, "Graph Neural Networks: A Review of Methods and Applications," 2018.

[17] M. N. Katehakis, "The Multi-Armed Bandit Problem: Decomposition and Computation," 1987.

[18] S. M. a. Y. M. Eyal Even-Dar, "Action Elimination and Stopping Conditions for the," 2006.

[19] "Spinning Up OpenAI page," [Online]. Available: https://spinningup.openai.com/en/latest/.

[20] "OpenAI Gymnasium," [Online]. Available: https://github.com/Farama-Foundation/Gymnasium.

[21] "Prometheus," [Online]. Available: https://prometheus.io/.

[22] "Grafana," [Online]. Available: https://grafana.com/.

[23] "Dell APEX AIOPS," [Online]. Available: https://www.dell.com/en-in/dt/apex/aiops.htm?gacd=9688261-13116-5761040-271777130-0&dgc=ST&&gad_source=1&gclsrc=ds#scroll=off&tab0=0.

[24] O.-C. Novac, M. Chirodea, C. Novac, N. Bizon, M. Oproescu, O. Stan and C. Gordan, "Analysis of the Application Efficiency of TensorFlow and PyTorch in Convolutional Neural Network," 2022.

[25] "Installing Libtorch," [Online]. Available: https://pytorch.org/cppdocs/installing.html.

[26] "CUDA 12.4 Libraries," [Online]. Available: https://developer.nvidia.com/cuda-12-4-0-download-archive.

[27] "nvcc issues - Github Issue," [Online]. Available: https://github.com/vllm-project/vllm/issues/129.

[28] "cmake about page," [Online]. Available: https://cmake.org/about/.

[29] "Google Cluster Data - 2019," [Online]. Available: https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[30] "Alibaba Cluster Data," [Online]. Available: https://github.com/alibaba/clusterdata.

[31] "Alibaba cluster trace schema," [Online]. Available: https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2017/schema.csv.

[32] "MAB Wiki Page," [Online]. Available: [https://en.wikipedia.org/wiki/Multi-armed_bandit] .

[33] "Deep Q Learning Wiki Page," [Online]. Available: https://en.wikipedia.org/wiki/Q-learning#Deep_Q-learning.

[34] "libcurl docs," [Online]. Available: https://curl.se/libcurl/.

[35] "nlohmann/json docs," [Online]. Available: https://github.com/nlohmann/json.

[36] "FIO docs," [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html.

[37] "Live Optics," [Online]. Available: https://app.liveoptics.com/.

[38] V. L. G. S. M. B. B. L. P. L. F. S. A. P. Antonio Longa, "Graph Neural Networks for temporal graphs: State of the art, open challenges, and opportunities," 2023.

[39] "OpenAI PPO," [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[40] "OpenAI SAC," [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html.

[41] "Event Driven Architecture Wiki Page," [Online]. Available: https://en.wikipedia.org/wiki/Event-driven_architecture.

Appendices

Appendix A: GitHub Repository Documentation

Link to public release version (github repository) of the codebase:
https://github.com/Erscheinung/Enterprise-AI-Monitor-BetaPublicRelease

The README currently has broken links but the React dashboard has its own README section.

Appendix B: Suggested Optional Reading

These are some preliminary readings done based on module specific implementation and research but not referred to or considered during the thesis.

Infrastructure Prediction:

- Resource Management with Deep Reinforcement Learning https://dl.acm.org/doi/10.1145/3005745.3005 750)

- Time Series Forecasting of Cloud Resource Usage (https://ieeexplore.ieee.org/document/9666444)

- CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models (https://ieeexplore.ieee.org/document/8356346)

Network Security:

- Network Intrusion Detection System using Deep Learning(https://www.sciencedirect.com/science/article/pii/S1877050921011078)

- Towards Developing Network forensic mechanism for Botnet Activities in the IoT based on Machine Learning Techniques https://arxiv.org/abs/1711.02825)

Business Process:

- Method towards reconstructing collaborative business processes with cloud services using evolutionary deep Q-learning (https://www.sciencedirect.com/science/article/abs/pii/S2452414X20300649)

- Business Processes Analysis with Resource-Aware Machine Learning Scheduling in Rewriting Logic (https://www.researchgate.net/publication/362336969_Business_Processes_Analysis_with_Resource-Aware_Machine_Learning_Scheduling_in_Rewriting_Logic)

Check list of items for the Final report

a.	Is the Cover page in proper format?	Y
b.	Is the Title page in proper format?	Y
c.	Is the Certificate from the Supervisor in proper format? Has it been signed?	Y
d.	Is Abstract included in the Report? Is it properly written?	Y
e.	Does the Table of Contents page include chapter page numbers?	Y
f.	Does the Report contain a summary of the literature survey?	Y
g.	Are the Pages numbered properly?	Y
h.	Are the Figures numbered properly?	Y
i.	Are the Tables numbered properly?	Y
j.	Are the Captions for the Figures and Tables proper?	Y
k.	Are the Appendices numbered?	Y
l.	Does the Report have Conclusion / Recommendations of the work?	Y
m.	Are References/Bibliography given in the Report?	Y
n.	Have the References been cited in the Report?	Y
o.	Is the citation of References / Bibliography in proper format?	Y

Bharat Shukla (Supervisor)
Manager, Cloud Specialty Domain