

Functional Evaluation of a CNN-Based Network Intrusion Monitoring Framework Integrating Zeek, Snort, Elasticsearch, and Django

Sarah Yasmin Yuszairulakhbar¹, Ahmad Kamalrulzaman Othman^{2*}, and Yusnita Sokman²

¹Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA Shah Alam, Bangunan Al-Khawarizmi, 40450 Shah Alam, Selangor Darul Ehsan, Malaysia

²Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA Cawangan Johor, Kampus Segamat, 85000 Segamat, Johor, Malaysia

*Corresponding author

DOI: <https://doi.org/10.51244/IJRSI.2026.1313CS025>

Received: 27 August 2026; Accepted: 01 September 2026; Published: 11 September 2026

ABSTRACT

Offline benchmark performance does not establish that an intrusion classifier can operate reliably within a live monitoring workflow. This study evaluates the functional deployment and deployment readiness of a convolutional neural network (CNN)-based network intrusion monitoring prototype integrating Ubuntu virtual machines, Zeek and Snort, Elasticsearch, a saved 1D-CNN with serialized feature-selection and scaling artefacts, and a Django administrator interface. The prototype was examined as an end-to-end system rather than as an isolated classifier. Functional evidence was assessed across model loading, manual prediction, CSV-based testing, Elasticsearch retrieval, periodic live-table refresh, and benign/suspicious visualization. A code-level audit then compared training-time and deployment-time feature schemas, preprocessing, input types, and decision rules. The implementation demonstrates a working monitoring workflow from network telemetry to administrator-facing classification. However, the documented live routine constructs four Zeek-derived model-facing fields before invoking a selector fitted on a 23-feature preselection vector, protocol encoding is not clearly aligned with the numeric representation used during training, manual and CSV routes expose inconsistent feature definitions, and the deployed decision threshold of 0.1 differs from the 0.5 threshold used in offline evaluation. The available records also do not contain labelled live-traffic accuracy, end-to-end latency, throughput, CPU usage, memory usage, dropped-event rate, or alert delay. Consequently, the evidence supports functional implementation and architectural feasibility, but not quantitative real-time effectiveness. The study contributes an auditable deployment architecture, identifies the feature-schema mismatch as the principal technical constraint, and specifies a targeted validation protocol based on labelled replay, stage-level latency instrumentation, throughput stress testing, resource profiling, and consistent inference-path regression tests. These results establish the measurements and corrections required before stronger real-time claims can be justified.

Keywords: deployment validation, network intrusion detection, 1D-CNN, Zeek and Elasticsearch

INTRODUCTION

Machine-learning network intrusion detection research remains heavily benchmark-centric. Models are commonly trained and evaluated on labelled public datasets and then compared using accuracy, precision, recall, F1-score, or ROC-AUC. These measures are necessary for discrimination analysis, but they do not answer whether the trained pipeline can accept live telemetry, preserve the training representation, sustain operational load, or deliver useful alerts. Surveys of data-driven and deep-learning intrusion detection repeatedly identify dataset representativeness, class imbalance, false alarms, and real-world generalization as persistent challenges (Ahmad et al., 2021; Chou & Jiang, 2021). Reliable evaluation therefore requires more than a high offline score.

Deployment introduces a second class of failure that is architectural rather than statistical. Capture tools produce operational fields, preprocessing artefacts expect a fixed schema, the model expects a fixed tensor shape, and the user interface imposes timing and presentation behaviour. If any boundary changes the feature order, type, scale, or decision threshold, the deployed system is no longer technically equivalent to the evaluated classifier. Magan-Carrión et al. (2020) argue for reliable and comparable NIDS evaluation, while Apruzzese et al. (2022) show that cross-dataset evaluation can expose performance that is hidden by a single benchmark. More recent work also demonstrates that leakage-safe partitioning and feature selection materially affect the credibility of intrusion-detection results (Aburomman et al., 2026).

CNN-based NIDS studies confirm that convolutional models can classify flow-derived representations effectively, including 1D-CNN designs for tabular traffic features (Qazi et al., 2022; Mohammadpour et al., 2022; Mo et al., 2024). Yet model architecture alone does not resolve the deployment gap. Practical systems have examined real-time web intrusion detection, end-to-end deployment, FPGA acceleration, edge-oriented detection, and scalable streaming pipelines (Kim et al., 2020; Bertoli et al., 2021; Le Jeune et al., 2021; Racherla et al., 2024; Maasaoui et al., 2025). Shreya et al. (2026), for example, integrate Zeek with a streaming analytics stack and searchable backend. These studies treat intrusion detection as a complete processing system rather than an isolated classifier.

The evaluated prototype follows the same systems direction. It was built around a 1D-CNN trained on CSE-CIC-IDS2018 and deployed with Ubuntu virtual machines, Zeek and Snort for network visibility, Elasticsearch for centralized storage, and a Django web interface. The model package contains a saved Keras model, feature selector, and scaler. The interface exposes manual feature entry, CSV-based testing, and a live page that retrieves recent records and refreshes approximately every three seconds. The resulting architecture moves beyond an offline notebook and creates a concrete path from network telemetry to an administrator-facing classification.

Research problem, questions, and contributions

The central problem is not whether the offline CNN can classify benchmark records. It is whether the deployed path reproduces the same input representation and operating point, and whether the available evidence justifies the term real-time. The implementation records support three research questions: (RQ1) which end-to-end functions are demonstrably implemented, (RQ2) which offline-to-live mismatches threaten reproducibility and prediction validity, and (RQ3) which measurements are still required before quantitative real-time effectiveness can be claimed.

The study makes three contributions. First, it reconstructs the prototype as an end-to-end architecture linking traffic observation, storage, preprocessing, model inference, and administrator interaction. Second, it audits schema dimensionality, type encoding, preprocessing consistency, inference routing, and decision-threshold equivalence across manual, CSV, and live paths. Third, it defines a targeted operational benchmark covering labelled live or replayed accuracy, latency, throughput, event loss, resource use, and alert delay. The scope is deliberately bounded. Because those operational measurements were not collected in the source experiment, this paper does not infer or fabricate them.

MATERIALS AND METHODS

System architecture and deployment environment

The prototype separates offline model preparation, live data acquisition, centralized storage, inference, and administrator interaction. CSE-CIC-IDS2018 is used for offline model development. In the deployed environment, Zeek generates structured connection telemetry and Snort provides signature-oriented monitoring on Ubuntu virtual machines. Elasticsearch indexes network records and forms the retrieval boundary used by the Django application. The web application loads the serialized preprocessing artefacts and CNN, retrieves recent events, invokes the prediction pipeline, and presents benign or suspicious status to an administrator.

This separation is operationally sensible because traffic capture, model serving, storage, and presentation can evolve independently. The critical systems boundary is the feature adapter between live telemetry and the trained preprocessing pipeline. That adapter must deterministically reproduce the same 23-feature preselection vector, data types, ordering, and semantic definitions used when the selector and scaler were fitted. Without that mapping, architectural integration can be demonstrated, but predictive equivalence cannot.

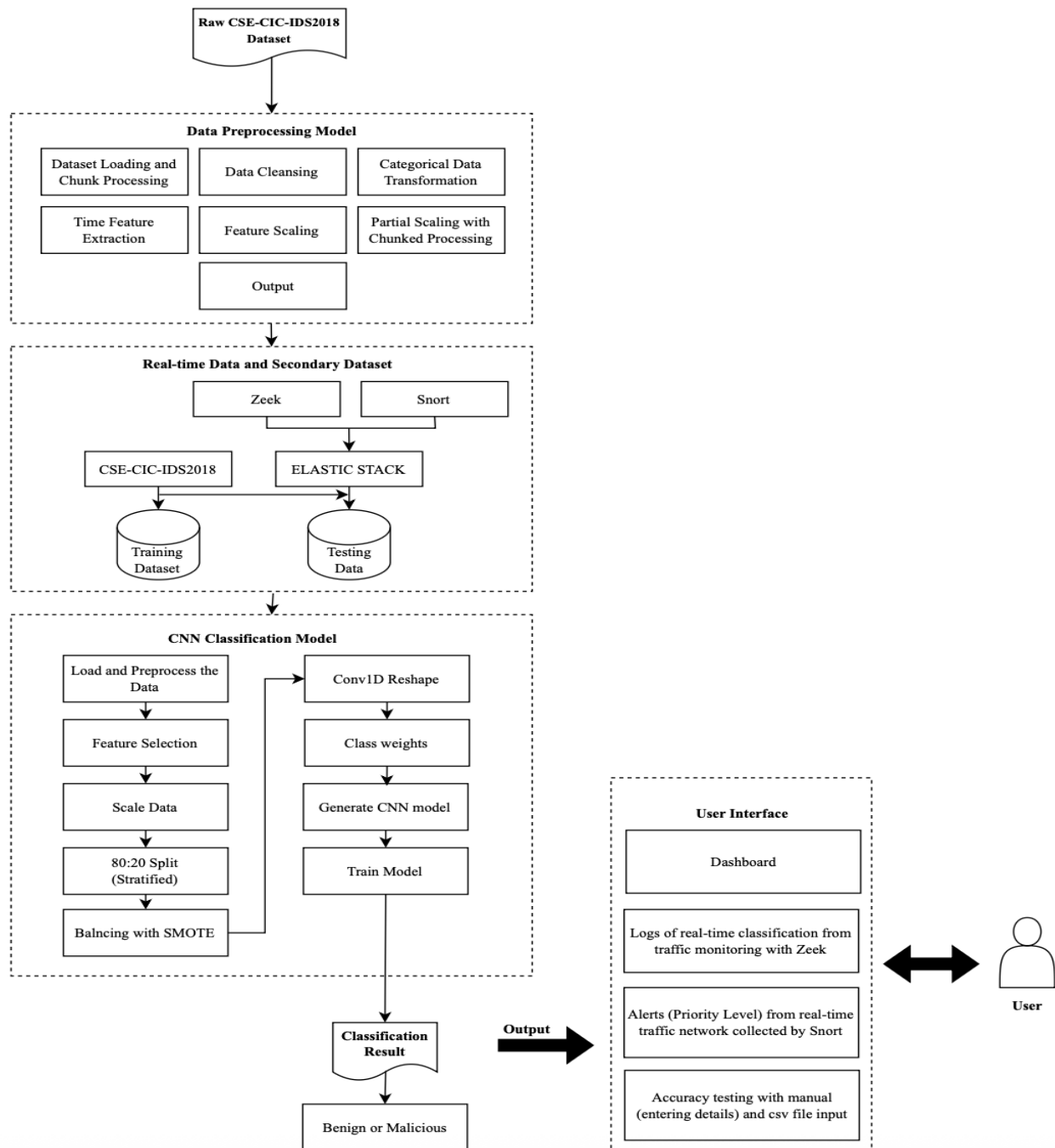


Figure 1. Prototype architecture linking preprocessing, CNN model development, live traffic monitoring, and administrator interaction.

Table 1. Prototype components and operational roles.

Component	Technology	Operational role
Traffic observation	Zeek on Ubuntu	Generate structured network telemetry

Signature monitoring	Snort on Ubuntu	Provide parallel signature-oriented security visibility
Central storage	Elasticsearch / Elastic Stack	Index and retrieve network records
Classifier	Saved Keras 1D-CNN	Produce malicious-class probability
Preprocessing	Serialized selector and scaler	Replicate training-time transformations
Web application	Django + HTML/JavaScript	Support manual, CSV, and live monitoring views
Virtualization	VMware Fusion	Host multiple Ubuntu security components

Offline model artefacts and inference contract

The trained CNN is saved as a Keras model, while the feature selector and scaler are serialized separately. Reusing these fitted artefacts is the correct deployment pattern because refitting them on live observations would alter the learned representation. The model loader defines a 23-feature preselection vector that corresponds to the selected CSE-CIC feature set plus engineered rates and a flag ratio. The selector reduces this vector to the 20 dimensions expected by the CNN, after which the scaler transforms the selected values and the array is reshaped for Conv1D inference.

The deployment contract is therefore strict: every prediction path must create all required numeric inputs in exactly the expected order, call the same selector, apply the same scaler, reshape identically, and use a documented decision rule. A shared inference function is preferable to separate implementations because it reduces silent divergence among manual, file-based, and live predictions.

Live acquisition, feature mapping, and type consistency

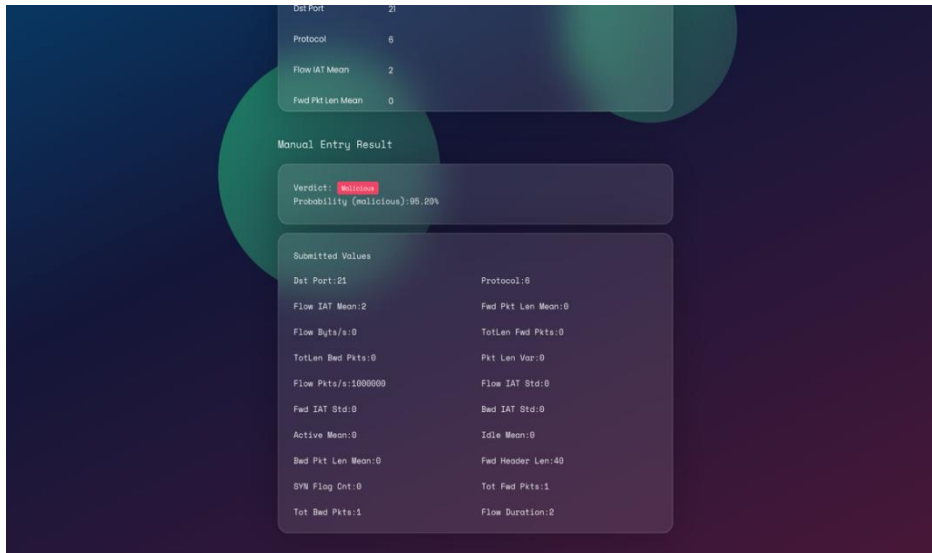
The live routine queries Elastic search for recent documents tagged as Zeek features. The requested source fields include timestamp, origin and responder addresses, responder port, protocol, duration, origin packet count, and responder packet count. The interface later displays source IP, destination IP, destination port, protocol, timestamp, and classification. This retrieval layer is sufficient for monitoring display but does not by itself reproduce the trained feature vector.

The implementation excerpt constructs only four model-facing values, duration, origin packets, responder packets, and protocol, before calling the prediction routine. The saved selector, however, was fitted on 23 preselection inputs. In the documented form, a four-value vector cannot be treated as equivalent to that training representation. Either an omitted feature-construction stage exists or the shown live routine is incomplete. Reproducible deployment requires an explicit feature adapter that derives the required CSE-CIC-style statistics from Zeek or other live sources. Features that cannot be reconstructed reliably should trigger retraining on a live-available feature set rather than silent zero filling.

Protocol type is a second compatibility requirement. Zeek can emit categorical protocol values such as tcp or udp, while the training pipeline used numeric encoding. The deployed adapter must use the same deterministic encoder and must reject or explicitly handle missing and previously unseen values. Schema validation should occur before model invocation so that incomplete records fail visibly rather than producing confident predictions from semantically invalid inputs.

Administrator interaction and threshold rule

The Django interface exposes three interaction modes. Manual mode accepts one feature vector, CSV mode processes multiple observations and can compare predictions with known labels, and the live page retrieves recent network records asynchronously. The live table refreshes approximately every three seconds. This polling interval describes the user-interface update policy. It is not an inference-latency measurement because capture, feature construction, storage, inference, and display can each contribute separate delay.



Dist Port: 21
Protocol: 6
Flow IAT Mean: 2
Fwd PkLen Mean: 0

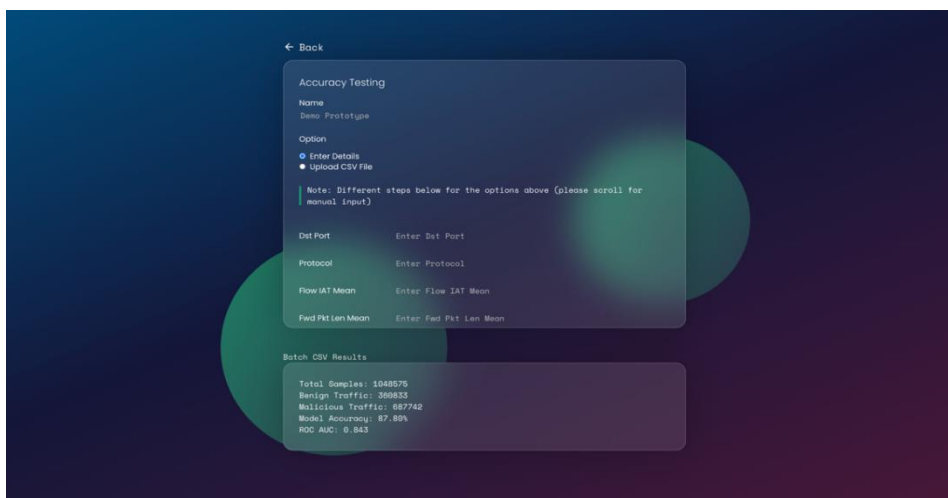
Manual Entry Result

Verdict: **Benign**
Probability (malicious): 95.29%

Submitted Values

Dist Port: 21	Protocol: 6
Flow IAT Mean: 2	Fwd PkLen Mean: 0
Flow Bkts/s: 0	TotLen Fwd Pkts: 0
TotLen Bwd Pkts: 0	PktLen Var: 0
Flow Pkts/s: 1000000	Flow IAT Std: 0
Fwd IAT Std: 0	Bwd IAT Std: 0
Active Mean: 0	Idle Mean: 0
Bwd PkLen Mean: 0	Fwd Header Len: 40
SYN Flag Cnt: 0	Tot Fwd Pkts: 1
Tot Bwd Pkts: 1	Flow Duration: 2

Figure 2. Manual feature-entry interface documented in the implementation.



← Back

Accuracy Testing

Name: Demo Prototype

Option:

- Enter Details
- Upload CSV File

Note: Different steps below for the options above (please scroll for manual input)

Dist Port: Enter Dist Port

Protocol: Enter Protocol

Flow IAT Mean: Enter Flow IAT Mean

Fwd PkLen Mean: Enter Fwd PkLen Mean

Batch CSV Results

Total Samples: 1048576
Benign Traffic: 360833
Malicious Traffic: 687742
Model Accuracy: 97.50%
ROC AUC: 0.843

Figure 3. Example classification output interface documented in the implementation.

The implementation also contains a schema inconsistency in the user-facing paths. One manual view lists 16 features, whereas the narrative states that users enter 23. The interface should not maintain independent feature lists. A single versioned schema should be imported by manual, CSV, and live routes so that every path inherits the same names, order, data types, transformations, and validation rules.

Offline evaluation converts predicted probabilities to labels at a threshold of 0.5, while the deployment loader sets a threshold of 0.1 to increase sensitivity. A lower threshold is operationally plausible, but it changes the operating point and normally trades higher recall for more false positives. Consequently, offline precision, recall, false-positive rate, and confusion-matrix values at 0.5 cannot be transferred to the live system at 0.1. The deployment threshold should be selected on validation data against an explicit objective and then held fixed during live testing.

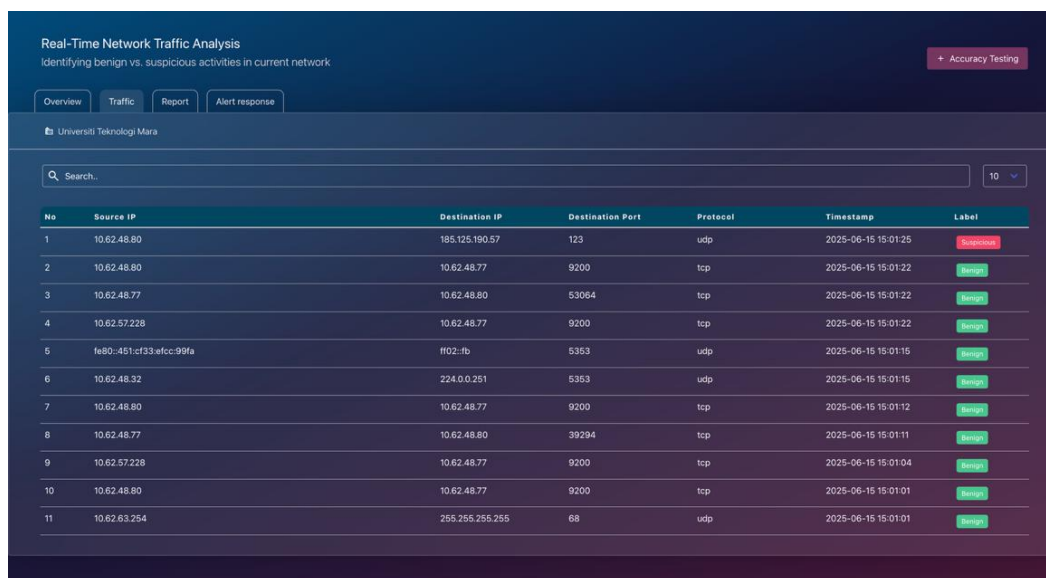
Audit procedure and evidence criteria

The deployment audit treats code, saved artefact usage, interface behaviour, and screenshots as implementation evidence. Each prediction path is checked against five consistency dimensions: input dimensionality and feature order, categorical-to-numeric encoding, reuse of the fitted selector and scaler, probability-to-class threshold, and user-interface routing. Functional claims are accepted only when the implementation records show the corresponding path or rendered output. Quantitative claims are withheld when the source records contain no labelled live ground truth or timing and resource instrumentation.

RESULTS

Functional validation

The implementation demonstrates several end-to-end functions. It loads a saved CNN together with preprocessing artefacts, queries Elasticsearch, exposes a Django endpoint that returns traffic data, renders manual and CSV testing interfaces, and periodically refreshes a live monitoring table. These elements establish that the project progressed beyond offline model training and implemented a multi-component monitoring workflow.



The screenshot shows a web interface titled "Real-Time Network Traffic Analysis" with the subtitle "Identifying benign vs. suspicious activities in current network". It includes tabs for "Overview", "Traffic", "Report", and "Alert response", and a "+ Accuracy Testing" button. The interface displays a table of network traffic records for "Universiti Teknologi Mara". The table has columns for "No", "Source IP", "Destination IP", "Destination Port", "Protocol", "Timestamp", and "Label". The records are numbered 1 through 11, and the "Label" column shows "Suspicious" (in red) for record 1 and "Benign" (in green) for records 2 through 11.

No	Source IP	Destination IP	Destination Port	Protocol	Timestamp	Label
1	10.62.48.80	185.125.190.57	123	udp	2025-06-15 15:01:25	Suspicious
2	10.62.48.80	10.62.48.77	9200	tcp	2025-06-15 15:01:22	Benign
3	10.62.48.77	10.62.48.80	53054	tcp	2025-06-15 15:01:22	Benign
4	10.62.57.228	10.62.48.77	9200	tcp	2025-06-15 15:01:22	Benign
5	fe80::451:c33:efcc:99fa	ff02::fb	5353	udp	2025-06-15 15:01:15	Benign
6	10.62.48.32	224.0.0.251	5353	udp	2025-06-15 15:01:15	Benign
7	10.62.48.80	10.62.48.77	9200	tcp	2025-06-15 15:01:12	Benign
8	10.62.48.77	10.62.48.80	39294	tcp	2025-06-15 15:01:11	Benign
9	10.62.57.228	10.62.48.77	9200	tcp	2025-06-15 15:01:04	Benign
10	10.62.48.80	10.62.48.77	9200	tcp	2025-06-15 15:01:01	Benign
11	10.62.63.254	255.255.255.255	68	udp	2025-06-15 15:01:01	Benign

Figure 4. Live traffic-monitoring table documented in the implementation.

The strongest functional evidence is the live table, which displays multiple network records with source and destination information, protocol, timestamp, and class status. The interface visually differentiates benign and suspicious records and is updated through repeated retrieval. The screenshot verifies the intended monitoring workflow but does not establish ground-truth predictive correctness because the displayed events are not paired with independently verified labels.

Manual and CSV modes provide additional functional routes. Manual mode demonstrates single-record input and rendered output. CSV mode implements batch testing and includes an accuracy calculation when true labels are present. However, the documented CSV function appears capable of calling the model directly on selected columns rather than necessarily routing through the same selector-scaler-model function used elsewhere. Functional presence is therefore supported, while pipeline equivalence must still be verified.

Table 2. Functional capabilities supported by the implementation evidence.

Capability	Implementation evidence	Assessment
Model loading	Keras model, selector, and scaler loaded in Python	Implemented

Manual prediction	Django/HTML form and output screenshots	Functionally demonstrated
CSV evaluation	Upload path and label-based accuracy code	Implemented, preprocessing consistency to verify
Elasticsearch retrieval	Query for recent Zeek feature documents	Implemented in code
Live refresh	Asynchronous polling approximately every 3 s	Functionally demonstrated
Class display	Live table with benign/suspicious status tags	Functionally demonstrated
Labelled live accuracy	No verified live labels reported	Not measured
Latency / throughput	No timed operational benchmark reported	Not measured

Deployment audit findings

Table 3. Offline-to-live mismatches affecting reproducible deployment.

Issue	Observed evidence	Technical risk	Required correction
Feature dimensionality	Four live values passed toward a selector fitted on 23 inputs	Transformation failure or invalid representation	Implement and test a complete 23-feature live mapper, or retrain on live-available features
Protocol encoding	Live protocol may be text, training used numeric codes	Type mismatch and semantic drift	Use one deterministic encoder shared by training and deployment
Manual schema	One view lists 16 features, narrative states 23	Different paths can feed different inputs	Define one versioned schema and import it everywhere
Decision threshold	Offline 0.5, deployment 0.1	Live error rates cannot inherit offline metrics	Select threshold on validation data and report threshold-specific results
CSV routing	Shown code may bypass selector and scaler	Predictions may differ from the trained pipeline	Route all inference through one tested function
Polling vs. latency	Browser refresh fixed near 3 s	UI interval can be mistaken for processing latency	Timestamp capture, feature-ready, inference, storage, and display stages
Snort role	Architecture includes Snort, shown live query uses Zeek	Contribution to ML inference is ambiguous	State whether Snort supplies features, alerts, comparison, or parallel monitoring

The feature-dimensionality mismatch is the binding technical constraint. A selector fitted on 23 inputs cannot be assumed to accept a four-field live representation. This problem sits at the system boundary, not inside the CNN. The correction is an explicit feature-construction layer that maps live Zeek records into the training schema. Where features require aggregation across packets or flows, that aggregation must be specified and tested. If some training variables cannot be reconstructed reliably, a reduced-feature model should be retrained and evaluated rather than relying on arbitrary imputation.

The threshold mismatch is the binding evaluation constraint. A threshold of 0.1 may be defensible when attack misses are more costly than false alerts, but its operational consequences must be measured. A new confusion matrix at the deployed threshold is required, together with the resulting alert burden under benign traffic. Threshold optimization should be treated as part of validation rather than as an ad hoc deployment setting.

DISCUSSION

Interpretation and deployment implications

The results support architectural and functional feasibility, not quantitative real-time effectiveness. That distinction matters because a working interface can coexist with an invalid model-input path. The prototype shows that traffic records can be retrieved and displayed and that model artefacts can be integrated into a web application. It does not yet show that every displayed class label was produced from a representation equivalent to the training data or that the system satisfies any latency, throughput, or reliability target.

The feature adapter should therefore be treated as the primary engineering correction. A practical option is to reconstruct the required CSE-CIC-style flow statistics from Zeek connection records and any additional telemetry required by the selected features. The alternative is to retrain a reduced-feature CNN using only variables that can be generated deterministically in the live environment. This second route can become a useful ablation study. The existing model serves as an offline reference, while the reduced model is evaluated on the same benchmark and then deployed. If predictive degradation is small but live extraction becomes simpler and faster, the deployment contribution becomes stronger.

This trade-off is consistent with current deployment-oriented research. Lightweight architectures have been proposed specifically to reduce operational overhead while preserving detection capability (Anwar et al., 2026), and real-time work increasingly reports system-level behaviour rather than classification accuracy alone (Racherla et al., 2024; Maasaoui et al., 2025; Shreya et al., 2026). The prototype evaluated here is simpler than Kafka- or Spark-based streaming designs. Simplicity can be an advantage for a small institutional network, but only if sustainable load and tail latency are measured rather than assumed.

Snort also requires precise positioning. In the documented live inference path, Zeek-derived records are queried for model-facing processing, while Snort appears as a parallel IDS component. The manuscript should therefore describe Snort as complementary signature-oriented monitoring unless a fusion rule, feature contribution, or comparative experiment is implemented. This avoids attributing model functionality to a component that is not shown in the inference path.

Minimum operational validation campaign

A focused validation campaign can complete the systems evidence without developing a new classifier. The most controlled design is a labelled replay experiment. Known benign and attack traffic is replayed through the same capture, feature-construction, storage, inference, and display path used by the deployment, while ground-truth labels remain external to the model. This tests the feature mapper, threshold, and CNN as one operational pipeline and permits a deployed confusion matrix to be reported.

Latency should be instrumented by stage. For each event, record capture time, feature-ready time, inference start and finish, persistence or retrieval time where relevant, and the time the result becomes available to the interface. Report the median and high-percentile values because occasional long delays matter operationally even when the mean is small. Throughput should be increased gradually until queue growth becomes unstable, records are dropped, or a predefined latency target is exceeded. The resulting value is a sustainable processing rate, not a best-case model-only benchmark.

Resource profiling should measure CPU and memory for Zeek, Elastic search, the model-serving process, and the Django application. If components run in separate virtual machines, report per-VM utilization and the VM resource allocation. Reliability tests should include malformed records, missing required features, temporary Elastic search unavailability, component restart, and burst traffic. A schema-validation failure should be logged and surfaced rather than converted into a prediction from incomplete data. If the application is intended for concurrent administrators, a modest multi-user response test should also be included.

Table 4. Minimum quantitative evidence required before claiming real-time effectiveness.

Metric	Operational definition	Recommended report
Live/replay accuracy	Correct predictions divided by labelled events	Accuracy plus class-specific precision, recall, and F1
False-positive rate	Benign events predicted malicious / all benign events	Rate and alert volume under benign traffic
End-to-end latency	Traffic observation to available classification	Median, p95, and p99
Inference latency	Model-call duration only	Median, p95, and p99
Throughput	Sustained classified events per second	Maximum stable rate without queue growth or loss
Dropped-event rate	Lost or unprocessed events / generated events	Percentage at each load level
CPU and memory	Resource use by component or VM	Average and peak utilization
Alert delay	Attack event to visible alert	Distribution under realistic load
Concurrent response	Web response under multiple users	Median and p95 by client count

Limitations and validity boundaries

The current evidence has three explicit limits. First, the live records shown in the interface are not accompanied by verified ground-truth labels, so live predictive accuracy cannot be estimated. Second, no instrumentation records inference latency, end-to-end delay, throughput, dropped events, CPU use, memory use, storage pressure, or alert delay. Third, the feature and threshold mismatches mean that the documented live path cannot automatically inherit the offline model metrics. These are validity boundaries, not missing values to be approximated.

The deployment audit is also limited to the available implementation records. It cannot determine whether an undocumented adapter exists outside the shown code. The correct publication response is to make the schema mapper explicit, test all three inference paths against the same reference inputs, and preserve the resulting implementation version with the model and preprocessing artefacts. This would convert a code-level ambiguity into a reproducible systems method.

Reproducibility controls for final deployment

A journal-ready deployment should package the inference path as a versioned contract rather than a collection of loosely synchronized files. The schema definition should specify every input name, order, data type, unit, encoding rule, missing-value policy, and feature-construction formula. The saved selector, scaler, model, and threshold configuration should be tied to the same version identifier. This prevents a later interface or preprocessing edit from silently changing the data presented to the CNN while leaving the model filename unchanged.

The three user-facing routes should also be regression-tested against common reference cases. A small set of fixed input vectors can be passed through manual, CSV, and live-adapter paths. For the same logical flow, all routes should produce the same preselection vector, selected features, scaled tensor, probability, and final class at the configured threshold. Any difference should fail the test. This simple control directly addresses the current risk that the CSV path or manual view may bypass part of the shared preprocessing pipeline.

Operational provenance should be recorded with each prediction where feasible. At minimum, the stored event should retain the model version, schema version, threshold, prediction probability, final class, and timestamps needed for latency reconstruction. These fields make later incident review and threshold-specific re-evaluation

possible. They also separate user-interface behaviour from model behaviour, because a delayed browser refresh can be distinguished from delayed inference or delayed Elasticsearch access.

Finally, the deployment report should state the VM configuration, software versions, Zeek and Snort roles, Elasticsearch indexing pattern, Django serving configuration, and the exact source of each model-facing feature. These details are not administrative overhead. They determine whether another researcher can reconstruct the same experiment and whether performance measurements can be interpreted against the available compute resources. The present prototype already contains most architectural components required for this level of reporting. The remaining task is to make the boundaries explicit and test them systematically.

CONCLUSION

The CNN-integrated prototype demonstrates a transition from offline intrusion classification toward an operational monitoring workflow by integrating Zeek and Snort, Elasticsearch, saved 1D-CNN and preprocessing artefacts, and a Django administrator interface. The implementation confirms functional capabilities including telemetry retrieval, prediction routing, and administrator-facing visualization through manual, CSV, and live monitoring functions.

However, the evidence does not support quantitative claims of real-time predictive effectiveness. The audit identified a feature-schema inconsistency because the documented live routine constructs four model-facing values before invoking a selector fitted on a 23-feature vector. Reproducible deployment requires a complete feature adapter or retraining using operationally available features, together with a shared versioned schema, encoding process, preprocessing pipeline, and inference function. The deployment threshold also differs from offline evaluation, with 0.1 used in deployment and 0.5 used offline. Therefore, offline precision and recall results cannot be transferred to the live system without threshold-specific validation.

The study also lacks verified live-traffic accuracy, end-to-end latency, throughput, resource consumption, dropped-event rate, and alert-delay measurements. The live refresh interval represents interface behaviour rather than processing latency. Therefore, this work contributes a functional validation and deployment-readiness audit rather than a quantitative demonstration of real-time detection performance. Future evaluation should use labelled traffic replay or controlled live testing to measure predictive performance and system-level operational metrics before stronger deployment claims are made.

DECLARATIONS

Data availability. The offline classifier uses the publicly available CSE-CIC-IDS2018 dataset. Prototype evidence is based on the documented implementation and deployment artefacts.

Ethics. Ethical approval was not required because this study did not involve human participants or animals. Any future administrator usability study involving human participants must obtain appropriate institutional ethics approval before data collection.

Competing interests. The authors should confirm this declaration before submission to the selected journal.

REFERENCES

1. Aburomman, A. A., Alakhras, M., & Reaz, M. B. I. (2026). Leakage-safe IoT intrusion detection with White Shark Optimizer-based feature selection. *Array*, 30, 100903. <https://doi.org/10.1016/j.array.2026.100903>
2. Ahmad, Z., Khan, A. S., Shiang, C. W., Abdullah, J., & Ahmad, F. (2021). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, 32(1), e4150. <https://doi.org/10.1002/ett.4150>
3. Anwar, R. W., Abrar, M., Salam, A., & Ullah, F. (2026). A lightweight 1D convolutional neural network for malicious network traffic detection on heterogeneous benchmark datasets. *Array*, 30, 100925. <https://doi.org/10.1016/j.array.2026.100925>

4. Apruzzese, G., Pajola, L., & Conti, M. (2022). The cross-evaluation of machine learning-based network intrusion detection systems. *IEEE Transactions on Network and Service Management*, 19(4), 5152-5169. <https://doi.org/10.1109/TNSM.2022.3157344>
5. Bertoli, G. C., Pereira Junior, L. A., Saotome, O., dos Santos, A. L., Verri, F., Marcondes, C., Barbieri, S., Rodrigues, M. S., & de Oliveira, J. M. P. (2021). An end-to-end framework for machine learning-based network intrusion detection system. *IEEE Access*, 9, 106790-106805. <https://doi.org/10.1109/ACCESS.2021.3101188>
6. Chou, D., & Jiang, M. (2021). A survey on data-driven network intrusion detection. *ACM Computing Surveys*, 54(9), Article 182. <https://doi.org/10.1145/3472753>
7. Kim, A., Park, M., & Lee, D. H. (2020). AI-IDS: Application of deep learning to real-time web intrusion detection. *IEEE Access*, 8, 70245-70261. <https://doi.org/10.1109/ACCESS.2020.2986882>
8. Le Jeune, L., Goedeme, T., & Mentens, N. (2021). Towards real-time deep learning-based network intrusion detection on FPGA. In *Applied Cryptography and Network Security Workshops* (pp. 133-150). Springer. https://doi.org/10.1007/978-3-030-81645-2_9
9. Maasaoui, Z., Merzouki, M., Battou, A., & Lbath, A. (2025). A scalable framework for real-time network security traffic analysis and attack detection using machine and deep learning. *Platforms*, 3(2), 7. <https://doi.org/10.3390/platforms3020007>
10. Magán-Carrión, R., Urda, D., Díaz-Cano, I., & Dorronsoro, B. (2020). Towards a reliable comparison and evaluation of network intrusion detection systems based on machine learning approaches. *Applied Sciences*, 10(5), 1775. <https://doi.org/10.3390/app10051775>
11. Mo, Y., Li, H., Wang, D., & Liu, G. (2024). An intrusion detection system based on convolution neural network. *PeerJ Computer Science*, 10, e2152. <https://doi.org/10.7717/peerj-cs.2152>
12. Mohammadpour, L., Ling, T. C., Liew, C. S., & Aryanfar, A. (2022). A survey of CNN-based network intrusion detection. *Applied Sciences*, 12(16), 8162. <https://doi.org/10.3390/app12168162>
13. Qazi, E. H., Almorjan, A., & Zia, T. A. (2022). A one-dimensional convolutional neural network (1D-CNN) based deep learning system for network intrusion detection. *Applied Sciences*, 12(16), 7986. <https://doi.org/10.3390/app12167986>
14. Racherla, S., Sripathi, P., Faruqui, N., Kabir, M. A., Whaiduzzaman, M., & Shah, S. A. (2024). Deep-IDS: A real-time intrusion detector for IoT nodes using deep learning. *IEEE Access*, 12, 63584-63597. <https://doi.org/10.1109/ACCESS.2024.3396461>
15. Shreya, M., Jain, S., Shetty, S., Kumar, S. A., & Narasimha Swamy, S. (2026). Low-latency streaming-based network intrusion detection system. *International Journal of Networked and Distributed Computing*. Advance online publication. <https://doi.org/10.1007/s44227-026-00112-4>